Operational analysis of sequence diagram specifications

Doctoral Dissertation by

Mass Soldal Lund

Submitted to the Faculty of Mathematics and Natural Sciences at the University of Oslo in partial fulfillment of the requirements for the degree ph.d. in Computer Science

November 2007

Abstract

This thesis is concerned with operational analysis of UML 2.x sequence diagram specifications. By operational analysis we mean analysis based on a characterization of the executions of sequence diagrams, or in other words an operational semantics for sequence diagrams. We define two methods for analysis of sequence diagram specifications – refinement verification and refinement testing – and both are implemented in an analysis tool we have named 'Escalator'. Further, we make the first steps in the direction of extending our approach with support for availability analysis.

In order to facilitate operational analysis, we define an operational semantics for UML 2.x sequence diagrams. The operational semantics is loyal to the intended semantics of UML, and is proven to be sound and complete with respect to the denotational semantics for sequence diagrams defined in STAIRS – a framework for stepwise development based on refinement of sequence diagram specifications.

The operational semantics has a formalized meta-level, on which we define execution strategies. This meta-level allows us to make distinctions between positive and negative behavior, between potential and universal behavior, and between potential and mandatory choice, all of which are inherently difficult in an operational semantics.

Based on the operational semantics and its formalized meta-level, we define trace generation, test generation and test execution. Further, based on a formalization of refinement in STAIRS, the trace generation is used to devise a method for refinement verification, and the test generation and the test execution are used to define a method for refinement testing. Both are methods for investigating whether or not a sequence diagram specification is a correct refinement of another sequence diagram specification.

The operational semantics, the refinement verification and the refinement testing are implemented with the term rewriting language Maude, and these implementations are integrated in the Escalator tool. In addition, Escalator provides a graphical user interface for working with sequence diagram specifications and for running the analyses.

In order to facilitate availability analysis, we define a conceptual model for service availability where the basic properties of availability are identified. Further, we extend the operational semantics with support for one class of these basic properties, namely real-time properties, and outline how the operation semantics extended with time can be applied to make methods for timed analysis of sequence diagram specifications.

Preface

The work of this thesis was conducted within, and funded by, the Basic ICT Research project SARDAS (15295/431) under the Research Council of Norway. During my period as a doctoral student I have been connected to the University of Oslo, to SINTEF, and have had a half year visit to the University of Twente.

I would like to express my greatest thanks to my supervisor, Ketil Stølen, for his capable, persistent and spirited supervision. The quality of his ideas and his feedback has been of great importance for the result.

I also wish to thank the other principal researchers of the SARDAS project, Øystein Haugen, Rolv Bræk and Birger Møller-Pedersen, for their continuous feedback to my work, and our guest scientists Manfred Broy, Ina Schieferdecker and Thomas Weigert for the great insights they brought with them on their visits to Oslo. Thanks also to my fellow doctoral students in the project, Atle Refsdal, Judith Rossebø, Ragnhild Kobro Runde and Knut Eilif Husa, for all the interesting discussions.

Thanks to the people of the Precise Modelling and Analysis group at the Department of Informatics, University of Oslo. The group has, under the leadership of Olaf Owe and Einar Broch Johnsen, been a stimulating environment to be a doctoral student.

Thanks to the people of the Cooperative and Trusted Systems department at SINTEF Information and Communication Technology, where I have been so fortunate to have had my office during the work of my thesis, and its research director Bjørn Skjellaug for arranging thing so that I was able to start, and finish, my doctoral project. Thanks to Fredrik Seehusen and Ida Hogganvik for all the stimulating discussions in the periods we shared office, and also to the rest of the people in the Quality and Security Technologies group for creating an environment in which positive and constructive feedback is a matter of habit.

Thanks to the Formal Methods and Tools group at the University of Twente which, under leadership of Arend Rensink and Ed Brinksma, was so forthcoming in letting me stay there for a six months visit, and to the people of the group. A special thank to Mariëlle Stoelinga for supervising me during my visit, and special thanks to Tomas Krilavičius and Laura Brandán Briones for giving me a social life while in Twente.

Thanks to my parents, Helga Soldal and Svein Lund, for all their support, and to Svein for proof reading my thesis. To Ole Emil Hansen Lund, my grandfather and possibly the person that ever held the highest faith in me: I am writing this in memory of you.

Thanks to all my friends for just being supportive and good friends: Kathrin Greiner (my only friend, except colleagues, that ever read and understood parts of my thesis), Gudrun Kløve Juuhl and Tarjei Vågstøl (and little Dagny), Bendik Steinsland and Liv Kjersti Moen, Aslak Indreeide, Bendik Indreeide, Ingrid Baltzersen, Monika Brückl

(deeply missed), Arnved Soldal Lund (my brother) and Lajla-Margrethe Lindskog (and little Oskar), Nina Reim, Sigurd Jorde, Eirik Nordby and Bodil Hauso, Jokke and Mari Fjeldstad, and all the rest of you.

Finally, my deepest thanks to my girlfriend Eva Marie Meling Mathisen, for all her love, support and patience.

Oslo, November 2007
Mass Soldal Lund

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the current practice of production of computer systems there are great challenges. The size and complexity of software is ever increasing. Keeping track of all parts, interactions and dependencies in these systems is likewise becoming more and more difficult, and construction and management of them takes a conciderable effort.

Our dependency on computer systems and their correct operations is also increasing. Software is now an integrated part of our infrastructure and therefore of our society. In this situation it is vital that computer systems are dependable, available, safe and secure. This thesis focuses in particularly on the property of availability, but the results are also relevant for dependability in general.

Ensuring the desirable properties of large software systems requires good methods and languages for specifying the systems and good methods and tools for analyzing the systems. We believe analysis at the level of specifications could reveal problems of the system design before the systems are built, and should therefore be beneficial for system development. This view is consistent with the underlying ideas of both model driven development[1] (MDD) and formal methods[2]. Our aim is to work within both MDD and formal methods, and to make contributions to both fields.

In this thesis we develop a theory and a tool for analyzing specifications of message interactions. Specifications are made with the notation of sequence diagrams[3], and more precisely the tool analyses the relation between specifications of different levels of abstraction and detail. We show how the specification language and tool may be used to specify and analyse aspects of availability.

In software engineering[4] and MDD the focus is often on practical usability of specification languages, analysis methods and tools. This means that if a method for specification and analysis of a system is to have value, it should be practical. In formal methods the focus has traditionally been the mathematical properties of the methods, such as soundness, completeness and decidability. A formal foundation is necessary if we are to make precise and sound methods and tools. Unfortunately, formal specification languages and methods often require high competence in specific mathematical formalisms and are difficult to use by people who do not possess that competence [50].

[1] Model driven development is software development with models as the artifacts driving the development, as opposed to, e.g. code as the driving artifacts. See, e.g., [10, 42] for references on model driven development.

[2] Formal methods refer to the practice of using mathematical and logical formalisms to specify and analyze computer systems. See, e.g., [26, 76, 184, 189] for more information on formal methods.

[3] Specifically we apply UML 2.x [134] sequence diagrams. See section 1.1 and chapter 4.

[4] Software engineering is the field and practice of building software systems. See, e.g. [170].

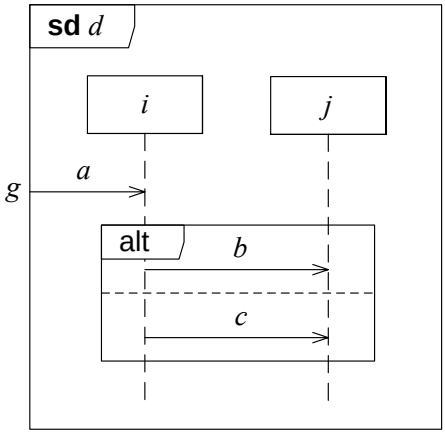


Figure 1.1: Sequence diagram

We adopt sequence diagrams as our specification language and an existing denotational formalization as its formal semantic foundation. By using sequence diagrams we get a specification language that is used in practice, and by adopting a formalized semantics we get the wanted formal foundation.

We implement a tool in which sequence diagrams are automatically analyzed. The tool allows the user to carry out the analysis without deep understanding of its formal basis. This means that the tool may be practical in use also for system developers that are not experts in formal methods. The methods for analysis are based on formal notions of refinement,[5] but apply testing techniques.[6]

1.1 Sequence diagrams

Sequence diagrams is a graphical notation for specification of message interaction between system components. Several notations for sequence diagrams exist. Most prominent are Unified Modeling Language (UML) sequence diagrams [134, 151] and their predecessor Message Sequence Charts (MSC) [85]. Both are are specification languages that have proved themselves to be of great practical value in system development.

We adopt UML sequence diagram as our notation. UML sequence diagrams have wide-spread use and several tools for making sequence diagram specifications are available. The UML standard defines the graphical notation and a well-defined abstract syntax. Figure 1.1 shows an example of a sequence diagram where two components exchange messages. Chapter 4 provides a more detailed introduction to UML sequence diagrams.

1.2 Formal semantics

The semantics of UML sequence diagrams is stated informally in the UML standard. This opens for ambiguities and may lead to different interpretations of the same diagrams. However, when a modeling language such as sequence diagrams is used either

[5] Refinement is a formalization of the concept of a specification becoming less underspecified. See, e.g., [27, 70, 158] for more information on refinement. See also section 5.4.

[6] See e.g. [24, 91, 178, 179, 183], and also chapter 12.

for development of a system or as documentation, it is important that the people involved have a common interpretation of the language. This motivates formal semantics in which the meaning of diagrams are unambiguously stated. This is even more important for tool makers. The only way to make sure that the makers and users of, for example an analysis tool, have same understanding of what results the tool provides, is by basing it on formal theory.

For this purpose we base our work on STAIRS.[7] STAIRS provides a formal trace based denotational semantics for sequence diagrams. This semantics assigns meaning to a sequence diagram by characterizing the execution histories that the diagram specifies. A thorough presentation of STAIRS is found in chapter 5.

1.3 Analysis

In model driven development one will typically start with a high-level model providing a coarse specification of the system to be developed. Over the course of the development the model will be refined into a specification closer to the implementation of the system. Such refinements will consist of decompositions, adding details and possibly functionality, and removing non-determinism. In practice there will always be a certain amount of backtracking as new insights emerges, better solutions are found, requirements evolve etc. It might also be that parts of the system are developed before, or faster than, other parts. However, the general direction will be more and more refined models.

Our focus is that of analyzing the relationship between specifications at different levels of abstraction, more specifically whether a low- level specification is a correct refinement of a high-level specification. We can operate with the expression "correct refinement" because we have a formal foundation on which we base our work, namely STAIRS. In this formal framework, the notion of refinement is defined mathematically in a way that captures our intuitive understanding of refinement of specifications.

Within formal methods there exists several approaches for doing analysis – manual or automated proofs (see e.g. [63]) simulation and bisimulation (see e.g. [125]), and model checking (see e.g. [34, 63]), among others. There are usability issues with these methods similar to those of formal languages – they are often difficult to understand and use for non-experts. A more fundamental issue with several of the formal methods is their lack of scalability. A strive for completeness often leads to state-explosion when the size of the analyzed systems increase (see e.g. [34]).

We aim at developing methods for analyzing sequence diagram specification based on testing techniques. There exists a formal testing theory (see chapter 12). We can apply this to make (formally) sound analysis methods based on testing techniques and also relate them to refinement. Testing has a practical angle where completeness may be sacrificed for the sake of the performance of the testing techniques and in this way give better scalability.

Testing is in its nature operational – when we are doing testing we are studying executions. This means we are in need of an operational semantics, a semantics that precisely characterize executions step by step – instead of characterizing the resulting

[7] STAIRS is a denotational semantics and refinement theory for UML sequence diagrams. STAIRS – with some variations – is defined in a number of publications; works cited in this thesis include [68–70, 144–147, 153–159].

histories as the denotational semantics of STAIRS does. This means we can apply testing techniques to specifications in the same way testing techniques may be applied to implementations An important contribution of this thesis is an operational semantics that is equivalent to the denotational semantics of STAIRS. This operational semantics constitutes the foundation for the development of analysis methods. The operational semantics is implemented in Maude, a formal specification language based on rewrite theory. With use of the operational semantics implemented in Maude and the Maude interpreter, we are able to execute sequence diagrams. Because Maude is a formal language, we may do our implementation without the risk that the meaning of the semantics is lost because the language of the implementation is ambiguous. An introduction to Maude in found in chapter 6.

1.4 Availability

Availability is often defined as the property of being accessible and usable on demand by an authorized entity [78, 81]. Languages and methods for analyzing availability properties require that availability is reduced to properties expressible by primitives in the language. Using the above definition means that properties "accessible" and "usable" must be reduced to something we can specify and analyze. In chapter 18 we present a model where availability is decomposed into basic aspects that may be expressed and analyzed. One of these aspects is timeliness, a property closely related to what is sometimes called soft real-time properties. We show how our specification language and operational semantics can be enhanced to handle time, and hence one aspect of availability.

1.5 Contributions

In this thesis there are five main contributions:

- We define an operational semantics for UML 2.x sequence diagrams. The operational semantics is proved to be sound and complete with respect the denotational semantics of STAIRS, and is implemented given an implementation in rewrite logic.
- Based on the operational semantics a method for refinement verification is defined.
- Based on the operational semantics we define a test generation algorithm for sequence diagrams and characterize test execution with respect to sequence diagrams. Together, the test generation and test execution is used to device a method for doing refinement testing.
- We build a tool for analysis of sequence diagrams based on our operational semantics and our testing theory for sequence diagrams. This tool is based on the implementation of the operational semantics, and implements both the refinement verification method and the refinement testing method.
- We present a conceptual model of availability that shows how the high-level property of availability may be decomposed and reduced into properties that can

be specified and analyzed. In a first step towards using this conceptual model as a basis for analyzing availability, we extend our operational semantics for sequence diagrams to handle one of these properties, namely real-time.

1.6 Structure of thesis

The thesis is structured into six parts, where the main contributions are found in parts III–V. Each of these parts starts with a introduction chapter and ends with a related work chapter. State-of-the-art is spread out through these introduction chapters, in addition to part II which provides background chapters. Related work is spread out through the related works chapters. In detail, the structure of the thesis is as follows:

Part I Research questions and methods provides a characterization of the problem to solve (chapter 2) and research methods (chapter 3).

Part II Background provides background on sequence diagrams (chapter 4), ST-AIRS (chapter 5) and Maude (chapter 6).

Part III Operational semantics contains an introduction (chapter 7), the definition of the operational semantics (chapter 8), definition of meta-strategies for execution of sequence diagram (chapter 9), the implementation of the operational semantics (chapter 10), and related work with respect to operational semantics of sequence diagrams (chapter 11).

Part IV Testing contains an introduction to testing theory (chapter 12), our testing theory for sequence diagrams (chapter 13), a presentation of our tool (chapter 14), a case study on use of the tool (chapter 15), and related work with respect to testing based on sequence diagrams (chapter 16).

Part V Availability contains an introduction (chapter 17), a presentation of our conceptual model for availability (chapter 18), definition of our operational semantics with time (chapter 19), a case study in the use of our time enhanced sequence diagrams (chapter 20), and related work with respect to timed specification and analysis (chapter 21).

Part VI Discussion and conclusions provides discussions (chapter 22) and conclusions (chapter 23).

In appendices A and B we provide detailed proofs, and in appendix C we provide implementation details.

1.7 Relation to other publications by the author

Aspects of this thesis have been published earlier as preliminary results in the following publications:

[A] M. S. Lund and K. Stølen. Deriving tests from UML 2.0 sequence diagrams with neg and assert. In *1st International Workshop on Automation of Software Test (AST'06)*, pages 22–28. ACM Press, 2006. (Same as [113] in the bibliography.)

[B] M. S. Lund and K. Stølen. A fully general operational semantics for UML 2.0 sequence diagrams with potential and mandatory choice. In *14th International Symposium on Formal Methods (FM'06)*, number 4085 in Lecture Notes in Computer Science, pages 380–395. Springer, 2006. (Same as [114] in the bibliography.)

[C] M. S. Lund and K. Stølen. A fully general operational semantics for UML sequence diagrams with potential and mandatory choice. Research report 330, 90 pages, Department of Informatics, University of Oslo, 2007. (Same as [115] in the bibliography.)

[D] J. E. Y. Rossebø, M. S. Lund, A. Refsdal, and K. E. Husa. A conceptual model for service availability. In *Quality of Protection: Security Measurements and Metrics (QoP'05)*, number 23 in Advances in Information Security, pages 107–118. Springer, 2006. (Same as [149] in the bibliography.)

[E] J. E. Y. Rossebø, M. S. Lund, A. Refsdal, and K. E. Husa. A conceptual model for service availability. Research report 337, 28 pages, Department of Informatics, University of Oslo, 2006. (Same as [150] in the bibliography.)

In detail, the relations to these publications are:

- Chapter 4 is based on [C].
- Chapter 5 is based on [A,C].
- Chapter 7 is based on [C].
- Chapter 8 is based on [A,C].
- Chapter 9 is based on [A,C].
- Chapter 11 is based on [C].
- Sections 13.1–13.2 are based on [A].
- Chapter 16 is based on [A].
- Chapter 18 is based on [D,E].
- Appendix A is based on [C].

Part I

Research questions and methods

Chapter 2

Problem statement

In chapter 1 we gave a brief presentation of the problems we seek to address and our solutions to them. In this chapter we elaborate and state our aims more clearly and explicitly. In section 2.1 we present our overall hypothesis and goal, and in section 2.2 we specify success criteria for our solution. The success criteria may be understood as overall requirements to the artifacts resulting from our work.

2.1 Main hypothesis and goal

Stated very loosely our aim is to contribute to improving system development. Based on our views presented in chapter 1, we formulate our overall hypothesis and starting point as follows:

System development will benefit (become more effective and less costly, and produce better software systems) from:

- *Methods and tools for doing analysis of systems at the model or specification*[1] *level. In this we include:*
  - *Analysis of behavior, but also analysis of such properties as availability.*
  - *Analysis of consistency between different levels of abstractions, i.e. that specifications at a lower level are refinements of specification at higher level.*
- *An integration of practical or pragmatic methods with formal methods. Specifically this refers to:*
  - *Integration of an approach based on diagram based specification languages with an approach based on languages with formal semantics.*
  - *Integration of formal refinement theory with practical testing techniques.*

A full investigation of this hypothesis is an enormous task that lies far beyond the scope of this thesis. However, we do seek to contribute to the investigation of the hypothesis by formulating a goal that will, if it is reached, indicate the validity of the hypothesis. We formulate our overall goal as:

[1] In the following we will use the terms *model* and *specification* interchangeably.

Develop a method and tool to analyze refinements and decompositions of specifications expressed in a diagram based language. The methods and tool should:

- *Handle functional behavior as well as aspects of availability.*
- *Have a sound formal foundation.*
- *Apply testing techniques.*
- *Be practical in use also for people without background in formal methods.*

Even though reaching this goal clearly does not validate the hypothesis, we may argue that it supports the hypothesis. If such methods and tools as described in the goal are developed and successfully applied, it would mean that a combination of practical diagram based languages and formal semantics is feasible, as well as a combination of formal refinement theory and testing techniques. Further it would indicate the benefit from having methods and tools that analyze behavior and availability properties at the specification level and in the relation between specifications at different levels of abstraction. Since the feasibility and benefit from such methods and tools are exactly what the hypothesis postulates, successful application of methods and tools as described in the goal would indeed be support of the hypothesis.

2.2 Success criteria

In the following a number of success criteria for our analysis method and tool are formulated. They may be seen partly as subgoals of the overall goal and partly as additional requirements. The success criteria define a number of requirements for the outcome of our project, and in chapter 22 we use these criteria for evaluating our results and to determine to what degree our goal is reached.

Success criterion 1 *The method should support model driven development by offering the capability of analyzing the relation between models or specifications at different levels of abstraction.*

Model driven development is a way of doing system development where models always are at the base of what is being done. Every stage of the development has specific kinds of models associated to it. For every step the models evolve and new models are made based on the models of the previous steps, and over the course of development the models will become more and more refined and closer to code/implementation. In this setting it is important that the models at different levels of abstraction, belonging to different stages of the development, are related in the appropriate way. In this sense a method that analyzes the relation between models will support model driven development by ensuring the correct relation between models.

Success criterion 2 *The method should offer the capability of analyzing aspects of availability.*

What is said above about specifications and models in general, holds equally well for availability. In model driven development there is a need for keeping consistency between specifications and models at different levels of abstraction, and this is also

true for availability specifications. Hence our method should facilitate analyzing the relation between specifications of availability aspects at different levels of abstractions. This is especially important for availability and related properties as they often will be important high-level requirements of software systems, and these requirements ideally should be reflected in lower level specifications.

The high-level definitions of availability we find in standards such as [78,81] cannot be applied directly to specification and analysis. We need to break the definitions down to aspects we may express in a specification language, and that we are able to analyze in a meaningful way. It is not realistic to develop a method to analyze all possible aspects of availability within the scope of this thesis, hence we focus on some selected aspects.

Success criterion 3 *The method should make use of functional models.*

System development is often focused on the behavior of systems, and hence around functional models or specifications in model driven development. Properties such as availability, sometimes referred to as non-functional properties[2], are often specified in separate text-based documents, e.g. stating the required availability. Instead of operating with separate "availability documents" or "availability specifications" we believe that specification of availability properties should be embedded in functional specifications, and that this would increase the chance of these properties getting the attention they deserve.

Success criterion 4 *The method should handle modularity.*

Industrial system development is moving towards component based development. Reuse of components and the use of third party components is increasing. In large projects few or no persons are involved in all parts of the development. In such a situation both development methods and methods for analysis must handle modularity.

This means that our methods should be able to do analysis of components separately. Even more relevant is analysis of whether the composition of components yields the expected results.

Success criterion 5 *The method should be based on practical testing techniques.*

Testing is the chief method for establishing correctness of computer systems and their conformance to requirements and specifications. This means testing is well understood, is well known by practitioners and is supported by several tools. Testing is also pragmatic in the sense that completeness (full coverage) usually is sacrificed for the sake of performance of the testing, or in other words that it is possible to do useful testing within time frames less that what would be needed to do a complete analysis.

We wish to use these characteristics of testing to our benefit. By basing ourselves on testing techniques we should be able to make a method for analysis that can be more easily applied in system development in its current state. We believe we may gain these benefits even though we do testing of specifications rather than implementations.

Success criterion 6 *The method should be formally sound.*

[2]We do not necessarily recognize them as non-functional.

Soundness in this respect is that the outcome of an analysis using our method is valid, and that the outcome follows from the input.

As stated earlier we believe specification languages and analysis methods should be founded on mathematical formalism. This is necessary for knowing what a specification expresses and how results of an analysis should be interpreted. Further it is necessary in tool making because the tool maker should have the same interpretation of the methods as the tool users.

When we base our method on mathematical formalism it also means that we can apply mathematical reasoning and mathematical proofs to ensure that the method is sound.

Success criterion 7 *The output from the method should useful.*

By useful we mean that a software developer using the method should be able to use its output for a practical purpose, such as to, e.g. improve a system design or make a convincing argument that a specification will indeed fulfill specified requirements. This is clearly a necessary criterion when developing methods for improving system development.

In order to fulfill this criterion, output of our method must reflects our intuition about the real world, and the method must be designed in such a way that anomalies do not corrupt the output.

Success criterion 8 *The method should be supported by a fully or partially automatic tool.*

As the above, this is a necessary criterion if our method should ever be applied in practical system development and not only remain an academic exercise. The development of tools of industrial standard is clearly outside the scope of this thesis. We do however think it is important to develop prototypes as "proofs of concept" to demonstrate that our method may be practically applied.

Practical application requires that the tool must be usable also for persons without high competence in the method and its theoretic foundation. Hence, for a tool to be practical, is should automate the method to as high a degree as possible.

Success criterion 9 *Application of the method at early stages of the system development improves the quality of systems with availability requirements.*

We have already stated that we aim at improving system development. Clearly then, one of the success criteria is that the method we develop actually improves system development. It might not be immediately evident what improved system development is, but obviously improved products of the development is one important aspect of what constitute an improvement in the development process.

Success criterion 10 *Application of the method makes development of systems with availability requirements more efficient and less costly.*

This criterion is related to the pervious one, and refers to another aspect of improving system development. While the previous success criterion states that we aim at improvements in the products of system development, this criterion states that these products should be produced using less resources. In other words our success criterion is that application of our method should make it possible to do system development with less effort.

Chapter 3

Research method

In this chapter we discuss and motivate the research methods used in the work of this thesis. We start by giving a general view of computer science as a scientific activity and its relation to other sciences in section 3.1. Then in section 3.2 we present an overall research process, and in section 3.3 we provide an overview of the research methods available. Finally, in section 3.4 we account for the research methods used.

It is common to distinguish between various branches of studies of computer related topics (sometimes informatics is used as a common name). An example is found in [37, p. 10]:

> This distinction can be simply (and perhaps contentiously) made by suggesting that, while *computer science* is about how computers function as hardware and software, and *software engineering* is about building technical systems and ensembles of hardware and software that meet given specifications, *information systems* is about understanding what is or might be done with these technical systems, and their effects in the human/organizational/social world.

In many research projects these different branches are interwoven, so in the following we use the term computer science broadly and loosely to embrace all of the above. Hence we see computer science to be the study of computerized systems in all their aspects.

3.1 Computer science as a science

A distinction may be made between pure science, applied science and technology [48, 166]. In pure science the objective is to investigate laws and properties of things and the relation between them, to increase our knowledge of the world and to formulate theories of the laws governing the world. In contrast to this, applied science is the activity of using the knowledge and theories of pure science in order to fulfill some purpose. The fundamental difference between pure science and applied science is that pure science is about acquiring knowledge for the purpose of knowing, while applied science is about using knowledge for practical purposes.

Technology, which includes engineering, is the invention, development and production of artifacts, where by artifact we mean something human made, created to serve a (not necessarily practical) purpose. Technology and applied science are closely related. What distinguishes the two is that technology may be based on experience in addition

to scientific knowledge and that applied science is not necessarily about creating artifacts. However, it is clear that there is a big overlap, and it is often impossible to place a research project in one, but not the other, of these categories.

Also the distinction between pure science (characterized by knowledge) on the one side and applied science and technology (characterized by purpose) on the other is not always a sharp distinction. Often, advances in pure science may result from advances in applied science and technology, and advances in applied science and technology may be a result of advances in pure science. These advances may be strongly connected and may appear from the same activity.

Computer science as a field belongs to both pure science, applied science and technology. For example the theory of computation [36, 105] may be seen as pure science while compiler construction (which uses elements from the theory of computation) is both applied science and technology. While compiler construction (designing and implementing a compiler) is applied science, the designing and implementing of most other computer programs must be seen as technology, but not as applied science. A computer program is clearly an artifact created to serve a purpose, but seldom based on scientific knowledge. Research in computer science is usually about inventing and developing new artifacts or improving existing artifacts (see e.g. [54, 55]), and therefore lean over to the applied science/technology end of the scale.

What distinguishes artifacts from naturally occurring objects or entities is that they are human creations and that they are created for a human purpose, see e.g. [48, 164, 166]. At the same time, artifacts must obey the same natural laws as any other objects and entities. An important remark here is that "artifact" must not be confused with something necessarily physical. Computer science produces a wide range of artifacts lacking a physical nature. Examples of non-physical artifacts of computer science can be algorithms, programming languages and system development methodologies.

While most computer scientists seldom are interested in the physical workings of a computer they are often interested in the running or execution of a computer program. The running of a computer program is a physical process, and may be studied as a such, see e.g. [127] and [164, pp. 19–21]. This shows that computer science is (partly) an empirical science and may be studied by means of experimental methods inspired by empirical sciences. Several advocates of this view may be found, e.g., [15, 49, 165, 175, 176, 191]. However, we oppose the view that computer science is chiefly an empirical science [127]. Mathematical artifacts such as algorithms and formal models may be studied by mathematical methods such as formal proofs.

The artifacts of computer science are neither purely technical nor technological. They may also be, e.g., organizational. Examples of this include system development methodologies and other computer related processes. Also the more technical or technological artifacts often interact with humans, for example graphical user interfaces. Hence, some parts of computer science, especially what is often called information systems, are closely related to social sciences such as organizational sciences or behavioral sciences. The implication of this is that also research methods from the social sciences are relevant for computer science [37, 72].

In summary we conclude that analytic, as well as empirical research methods, are of great relevance for computer science research, and that research methods can be borrowed from natural sciences and mathematics, as well as from the social sciences. This means we oppose the view presented in e.g. [64, 65] that computer science is

fundamentally different from other sciences and that research methods in computer science must reflect this special nature of computer science rather than being inspired by the research methods of other science.[1]

3.2 Research process

We have argued that computer science may apply methods from mathematics, natural sciences and social sciences, but we still need to see this in relation to the applied nature of computer science. Further, the methods need to be related to the research activity as a processes.

In pure science, the investigation of an object aims to discover its properties or to see whether it can be used to confirm or reject a theory. Either way the purpose is that of gaining knowledge. In applied science and technology, the purpose of an investigation of the properties of an artifact is to investigate the success of the artifact. The success of an artifact is the degree to which it fulfills its purpose, see e.g. [116].

We can formulate hypotheses about, and do hypothesis testing on, artifacts the same way we can formulate hypotheses about, and to hypothesis testing on, natural occurring objects, see e.g. [15]. This means that we can apply the same research methods and the same kind of research process for applied science and technology that we apply for pure science.

Classical research is characterized by an iterative process. In classical research this process may, very simplified, be described as 1) the observation of some phenomena that are not explained by existing theory, 2) the making of a theory to explain the observed phenomena, 3) justifying the new theory by arguing that it better explains the phenomena and/or testing of the theory by experiments, or other methods.

For applied science and technology we can describe a similar process for conducting research: 1) The possibility, need or wish for a new or improved artifact to serve some purpose, 2) development of the artifact, 3) evaluating the success of the artifact by arguing that it fulfills its purpose and/or testing the artifact by experiments, or other methods.

A parallel may be observed. In classical research the iteration of the process will ideally yield better and better theories to explain more and more phenomena. In applied or technological research, iteration of the process will ideally yield better and better artifacts to serve more and more purposes. Views similar to this is found in e.g. [116, 166, 169].

Even though this is a simplified account of research, and may be more normative than descriptive, it is sufficient for our purpose. The research process that we have followed in this work is a practical adaption of the process characterized above:

1. Specification of requirements for the artifacts that are developed in the work of this thesis by means of success criteria (see chapter 2).

2. Development of the artifacts in accordance to the requirements.

3. Evaluation of the artifacts developed with respect to how well they fulfill the success criteria (chapter 22).

[1] We do, however, recognize that computer science has some distinctive characteristics and accepts demonstrations ("proof of concept", prototyping) as a valid method in computer science [64, 65, 116]– but not as its sole method.

3.3 Research methods

In this section we give a brief presentation of various research methods and approaches. We base the presentation of [37, 116, 122, 190, 191]. Taken together we believe they provide a representative picture of the research methods available. In the presentation we focus on the generality, precision and realism of the various methods. This is important since no method can maximize all of these three simultaneously. Methods that give high generality sacrifice precision and realism, and so forth [122].

Based on the above cited literature we may group and summarize a number of research methods:

Demonstration. By *demonstration* we mean the production of an artifact and investigation of its feasibility by observing it working. This is similar to the *instantiation building activity* in [116], *technical development* in [37] and *assertion* in [191], and might also be referred to as *prototyping* and *proof of concept*. This is a simple method that is often applied early in development projects, but to a large degree lacks generality.

Field study. A *field study* is the systematic observation of an artifact (or phenomenon) in its natural (real world) environment. The researcher will usually try not to interact with the object(s)/system(s) during the study and will often observe specific variables to answer specific questions. This is a method which gives high realism, but may lack precision and generality.

There also exists a number of variations of this method. What is called *project monitoring* in [191] is a weak form of field study, where data is collected from the field and analyzed, but without pre-established questions to be answered. Related to this are also what are called *lessons learned* and *legacy data* in [191], where historically recorded observations and data are analyzed.

A *field experiment* in [122] or *replicated experiment* in [191] is a field study where one significant variable is changed and the effect of this is observed. An example of this may be observing the effect of introducing a new development tool in system development projects.

A variant of field study (or field experiment) is *case study* [190]. In a case study one specific case or situation is studied, which may give deeper insight in the case or situation, but less generality. A further variant of case study is *action research* (or *collaborative research*). In action research the researcher ceases to be an outside observer and takes part in the case or situation under study. The researcher is then himself or herself part of the subject and contributes with his or her skills and knowledge to the case or situation. This may give an even deeper understanding of the case or situation, but with the sacrifice of generality and the risk of observations being subjective.

Laboratory experiment. In a *laboratory experiment* (called *synthetic environment experiment* in [191]) an artificial (synthetic) environment is created in which the researcher can control the variables and make sure that conditions remain stable throughout the experiment. A task is performed several times in this artificial environment with or without changing the variables of the environment. Specific variables of the subjects are observed or preferably measured. A typical

laboratory experiment in computer science would be to let a number of users perform specific tasks on a user interface with controlled variations in the tasks they perform and/or the user interface. This method gives high precision because of the controlled environment, but may lack realism for the same reason. The generality may be reasonable if the subjects are carefully chosen.

A variant of laboratory experiment is what in [191] is called *dynamic analysis*. This refers to exposing an artifact (e.g., a computer program) to various (controlled) conditions or situations (e.g., input) and observe or measure the result. An example may be performance, stress or load tests of computer programs. This is also related to *experimental simulation* in [122]. *Benchmarking* is a specific type of dynamic analysis. A benchmarking test usually applies a collection of representative test data (a benchmark suite) that is given as input to several computer programs of the same type in order to observe or measure differences in, e.g. the performance of, the computer programs.

Simulation. A *simulation* refers to the study of a model instead of the actual thing, usually running on a computer. In [191], simulation is closely related to dynamic analysis with the difference being that a model of the environment is used instead of a real environment. Examples may be running a computer program together with a model of user behavior or on a simulated hardware platform while testing its performance. In [122], *computer simulation* refers to observing the behavior of a model of a real system while the model is running on a computer.

Survey. In a *survey* (*sample survey* in [122]) data is collected in form of responses to questions addressed to the subjects of the study. Usually the responses are collected by means of a questionnaire. If the subjects and questions are carefully chosen this method gives high generality, but lacks precision and realism.

A variant is to have the questions answered in *structured (or in-depth) interviews* [37] with the subjects. Often then, there will be fewer subjects, but the questions and answers will typically be more extensive. This may give deeper understanding of the topic under study, but compromises the generality. In another variant, called *judgment study* in [122], responses are gathered from a small group of judges subjected to a specific set of stimuli. This increases precision, but also compromises generality. Gathering expert opinions may be a further variant of survey.

Formal analysis. The archetype of *formal analysis* (*formal theory* in [122] and *theoretical research* or *formal-mathematical analysis* in [37]) is *theorem proving*. From a set of axioms and/or a mathematical model a theorem concerning some properties of the topic under study is formulated and proved using mathematical proof techniques.

We also consider less mathematical, but analytic, methods as formal analysis. Building a *structured argument*, for example a thought experiment, must be considered formal analysis. *Static analysis* in [191] is a related type of formal analysis. Static analysis is concerned with investigating the characteristics of an artifact looking at its structural properties. An example of static analysis is complexity measures of computer programs based on analyzing the code of the programs.

The activity of building or developing concepts and relations between concept (called *conceptual development* in [37]) must also be considered formal analysis. This activity is related to the *constructs and model building* activities in [116]. Forming of concepts and models may be based on empiric observations and is then related to empirical methods, but concept and model building may also be purely analytical.

Formal analysis scores high on generality, but due to its analytic rather than empirical nature, low in realism and precision.

Review. The method called *(literature) review* in [37] and *literature search* in [191] is based on analyzing and comparing results from previously published research. This method is sometimes called *(literature) survey*, but must not be confused with the (sample) survey discussed above. Reviews are seldom the sole method of a research project, but are often used for identifying the current state of, and important results within, the domain of the research project. The purpose of a review may be to gather published empirical results and use these for testing a hypothesis. In this case, a review is sometimes called a *meta-analysis*. A potential problem with reviews is bias in the selection of published works and data in the study.

3.4 Choice of research methods

Our work is to a large degree about developing artifacts, more specifically a method and a tool for analyzing specifications. Artifacts are produced to fulfill a purpose, and the success of an artifact is the degree to which it fulfills this purpose. The success criteria formulated in chapter 2 should be seen as high-level requirements to our method and tool, and hence as statements of the purpose of the artifacts.

The technical work, i.e. the development, have constituted the main effort in the work of this thesis. The aims of the work have been to not do everything from scratch, but build on existing theories, to build a tool as well as a method, and to make a method that is formally sound. The development was iterative in two ways. It has been iterative in the sense that the operational semantics was developed first, then the testing theory, and then the real-time extension. Each of these iterations have been preceded by literature reviews in order to identify the state-of-the-art, and to determine what existing theories could be used and where new development was needed. These literature reviews are reflected in chapters 7, 11, 12, 16, 18 and 21.

The other way the work was iterative is that the implementation of the tool and the proving of theorems have been carried out along with the development of the method. This way, there has been continuously feedback to the method development that has helped improving the method with respect to soundness, feasibility and usability.

When we evaluate our artifacts with respect to the success criteria, our chief research methods are formal analysis and demonstrations. Formal analysis is used in several of its forms to build arguments about the properties of our artifacts. Demonstrations are applied to show the feasibility of our artifacts.

It can be argued that formal analysis and demonstrations are too weak in order to properly evaluate some of the success criteria. In order to obtain a proper evaluation of these success criteria, some variants of field study, laboratory experiment, simulation

or survey should be applied. The main focus of our work has been the development of artifacts and, though unfortunately, there has not been sufficient time within the time-span of the project to conduct such kinds of evaluations in large scale. We have however expose our method and tool two studies that may be considered as small and limited laboratory experiments, in the from of dynamic analyses, and as small and limited case studies.

Success criterion 1 *The method should support model driven development by offering the capability of analyzing the relation between models or specifications at different levels of abstraction.*

This success criterion is evaluated by formal analysis in form of structured arguments, by demonstration in form of the development of a tool, and small and limited laboratory experiments in form of dynamic analysis. The dynamic analysis consists of exposing our tool to example input and comparing the actual output with expected output. Because we use a real specification we can also categorize this analysis as a case study. The tool is documented on chapter 14 and the case study in chapter 15.

Success criterion 2 *The method should offer the capability of analyzing aspects of availability.*

The research method applied for this success criterion is formal analysis in form of conceptual development, in combination with a literature review, in order to identify aspects of availability. In addition, a small and limited case study, in combination with a literature review, was carried out to evaluate our treatment of real-time. The conceptual development is documented in chapter 18, and the case study is documented in chapter 20.

Success criterion 3 *The method should make use of functional models.*

This criterion is evaluated by means of formal analysis in form of structured arguments.

Success criterion 4 *The method should handle modularity.*

The evaluation of this criterion is closely related to evaluation of success criterion 1. The methods applied are formal analysis in form of structured argument based on the tool development documented in chapter 14, and the case study documented in chapter 15.

Success criterion 5 *The method should be based on practical testing techniques.*

This criterion is evaluated by means of formal analysis in the forms of structured argument and static analysis, but also by demonstration in the form of the tool development documented in chapter 14, and the case study documented in chapter 15.

Success criterion 6 *The method should be formally sound.*

This criterion is evaluated by means of formal analysis in the forms of theorem proving (mathematical proof) and structured arguments. The proofs are documented in appendices A and B.

Success criterion 7 *The output from the method should useful.*

This is evaluated by means of the case study documented in chapter 15, together with formal analysis in form of static analysis and structured arguments.

Success criterion 8 *The method should be supported by a fully or partially automatic tool.*

Obviously this criterion is evaluated by the tool development (demonstration). Further, the criterion is evaluated by formal analysis in the form of structured arguments and static analysis, and by the case study documented in chapter 15.

Success criterion 9 *Application of the method at early stages of the system development improves the quality of systems with availability requirements.*

Ideally we should have conducted a larger field study in order to evaluate this success criterion. However, since there has not been sufficient time to conduct such field studies we have to resort formal analysis in the form of structured arguments in order to evaluate this criterion.

Success criterion 10 *Application of the method makes development of systems with availability requirements more efficient and less costly.*

As with success criterion 9, a proper evaluation of this criterion would require a larger field study. Because we have not been able to carry out any field study within the time-span of our project, for the evaluation of this criterion we also have to resort to formal analysis in the form of structured arguments.

Part II

Background

Chapter 4

Sequence diagrams

Sequence diagrams are a graphical specification language defined in the Unified Modeling Language (UML) 2.x[1] standard [134]. Sequence diagrams as defined in the UML 2.x standard are the last of a sequence of languages that have evolved over the last 15 to 20 years. Both UML sequence diagrams and their predecessor Message Sequence Charts (MSC) [85] are specification languages that have proved themselves to be of great practical value in system development.

An early version called Time Sequence Diagrams was standardized in the 1980s (see [47,82]). Better known are MSCs that were first standardized by ITU in 1993 (see e.g. [20]). This standard is usually referred to as MSC-92, and describes what is now called basic MSCs. This means that MSC-92 did not have high-level constructs such as choice, but merely consisted of lifelines and messages. MSC-92 had a lifeline-centric textual syntax[2], and was given a semantics formalized in process algebra.

In 1996, a new MSC standard was defined, called MSC-96 [83]. In this standard, high-level constructs and high-level MSCs were introduced, a kind of diagrams that show how control flows between basic MSCs. Further an event-centric textual syntax[3] and a new semantics were defined [84]. This semantics is also a kind of process algebra, but holds substantial differences from the MSC-92 semantics. Finally, the MSC-96 standard was revised in 1999 and became MSC-2000 [85], but kept the MSC-96 semantics. A further discussion on the MSC semantics is found in chapter 11.

The first versions of the Unified Modeling Language (UML 1.x) [131] included a version of sequence diagrams similar to MSC-92, i.e., consisting of only lifelines and messages and no high-level constructs. An important difference, however, was that the sequence diagrams of UML 1.x did not have the frame around the diagram, which in MSC-92 allowed messages to and from the environment of the specified system.

Sequence diagrams in UML 2.x may be seen as a successor of MSC-2000, since many of the MSC language constructs have been incorporated in the UML 2.x variant

[1]The UML standard exists in versions 1.3, 1.4, 1.4.2, 1.5, 2.0 and 2.1.1. The for us relevant changes occurred in the transition from version 1.5 to version 2.0. Hence, in this thesis we will operate with UML 1.x and UML 2.x with versions 1.4 [131] and 2.1.1 [134] as representatives.

[2]Lifelines represent the timelines of communicating parts or components in a sequence diagram (see section 4.1). In "MSC-terminology", lifelines are called *instances* or *instance lines*. A lifeline-centric syntax means that each lifeline is characterized by itself and a diagram as a collection of lifelines.

[3]In an event-centric syntax events, as opposed to lifelines, are the basic building blocks of a diagram. The event-centric syntax of MSCs is more general than the lifeline centric-syntax in that all diagrams expressed in the lifeline-centric syntax can be expressed in the event-centric syntax, but not the other way around.

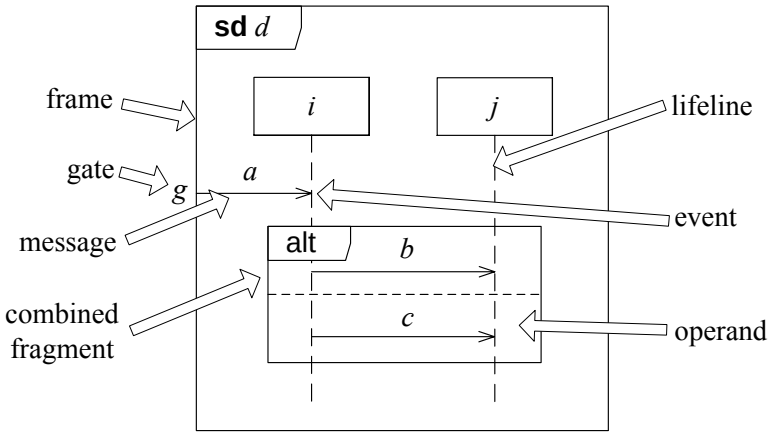


Figure 4.1: Sequence diagram

of sequence diagrams. This means that even though we work with UML 2.x sequence diagrams, much of the related work concerns MSCs. UML 2.x sequence diagrams are, however, neither a subset nor a superset of MSC-2000; there are both similarities and differences between the languages [67]. Most notably MSCs do not have any notion of negative behavior (see below).

4.1 Graphical notation

The UML 2.x standard defines the graphical notation, but also an abstract syntax, for the diagrams. Hence the language has a well-defined syntax.

Figure 4.1 shows a sequence diagram d in the graphical notation. A sequence diagram consists of a *frame*, representing the environment of the specified system, and one or more *lifelines*, representing components of the system. Arrows represent *messages* sent between lifelines or between a lifeline and the environment of the diagram. If the beginning or the end of an arrow is at a lifeline, this represents an *event*. Where an arrow meets the frame, there is a *gate*. *Combined fragments* are operators, like the choice operator alt, and each combined fragment has one or more *operands*.

In the diagram of figure 4.1 we see two components (lifelines) i and j. i receives a message a from gate g and them makes a choice between sending message b or message c to j.

4.2 Basic semantic model

The UML standard defines the semantics of sequence diagrams informally. Most notably, this is a trace-based semantics:

> The semantics of an Interaction[4] is given as a pair of sets of traces. The two trace sets represent valid traces and invalid traces. The union of these two sets need not necessarily cover the whole universe of traces. The traces that are not included are not described by this Interaction at all, and we cannot know whether they are valid or invalid. [...]

[4]In the UML standard, *Interaction* is used as the common name for diagrams specifying interaction by sending and receiving of messages. Sequence diagrams are then one kind of Interaction [our note].

> **Basic trace model:** The semantics of an Interaction is given by a pair $[P, I]$ where P is the set of valid traces and I is the set of invalid traces. $P \cup I$ need not be the whole universe of traces.
>
> A trace is a sequence of event occurrences denoted $\langle e1, e2, ..., en \rangle$. [...]
>
> Two Interactions are equivalent if their pair of trace-sets are equal. [134, pp. 479–480]

The STAIRS semantics is a formalization of sequence diagrams based on an extension of this semantics model. We adopt the semantic model of STAIRS, and thereby the semantic model described in the UML standard. This means we operate with the concepts of positive and negative behavior.[5] Behavior that is neither positive nor negative is referred to as inconclusive. A presentation of the STAIRS semantics is found in chapter 5.

4.3 Operators

In the diagram of figure 4.1 we saw an operator alt. The UML standard defines a number of operators of sequence diagrams that can be used in a similar way. Operators are either unary or they have two or more operands. The standard defines eight unary operators opt, break, loop, critical, neg, assert, ignore and consider, and four operators seq, alt, par and strict that may be viewed as binary or n-ary. Of these we handle the unary operators opt, loop, neg and assert, and all of latter, treated as binary. In addition we introduce two additional operators: the unary operator refuse and the binary operator xalt. We also introduce a constant skip. We consider seq, par and strict as *simple operators* and the remaining as *high-level operators*. Below we give a short description of each of the operators we use:

seq is called the weak sequencing operator, and defines a sequencing of its left argument and its right argument such that events on the left side must come before event on the right side if they belong to the same lifeline. In addition the causality of messages is preserved, but except for these two constraints the result may be any merge of the left side and the right side. See also section 7.3.3.

par is the parallel operator and defines interleaving of the events from each of its arguments. The result is any merge of the left side and the right side that preserves the causality of messages.

strict denotes strict sequencing, which means the left side must happen before the right side.

opt is option. This unary operator represents a choice between its argument and nothing.

neg is the negative operator and defines negative or invalid behavior. This operator represents a choice between this negative behavior or nothing. See also section 7.3.4.

[5] We will use the terms *negative* and *invalid* interchangeably. Further, *positive* and *valid* should be understood to be synonyms when we talk about behaviors.

refuse is introduced to handle an ambiguity in the interpretation of neg (see [155]). While neg defines a choice between negative behavior and nothing, refuse just defines negative behavior.

assert is the assertion operator and defines what may be considered a kind of universal behavior. This means that only the behavior of its arguments may be positive behavior and all other behavior is negative or invalid. See also section 7.3.1.

loop defines iteration of the behavior of its argument. The loop may be infinite or have a specified number of iterations.

alt is the alternative operator and represents a (non-deterministic) choice between its arguments.

xalt is called explicit alternative and is introduced to resolve an ambiguity in the interpretation of alt. While alt represents underspecification, xalt defines a non-deterministic choice representing an actual choice in the specified system. See also section 7.3.5.

skip represents the empty diagram.

Chapter 5

STAIRS

In this chapter we give a presentation of STAIRS. STAIRS may be seen as a development method based on UML 2.x sequence diagrams. This chapter covers the two main components of STAIRS, a denotational semantics for sequence diagrams and a refinement theory based on this denotational semantics. In addition to these two components there exist pragmatics for STAIRS [157], and extensions with time [70], data [158] and probabilities [145, 146]. The time and data extensions are given a short treatment in chapter 21.

In section 5.1 we give a brief introduction to the semantic model of STAIRS. Section 5.2 presents an abstract textual syntax for sequence diagrams, based on [70], that is used in defining the denotational semantics. In section 5.3 the denotational semantics is presented. The presentation is heavily based on [68] but also includes elements from [70, 155], more specifically treatments of infinite loop and the refuse operator which are not found in [68]. Section 5.4 provides a presentation of refinement as defined in [70, 159].

5.1 Introduction

The STAIRS semantics is trace based and uses an extended version of the basic semantic model from the UML standard. Instead of a single pair (p, n) of positive (valid) and negative (invalid) traces, the semantic model of STAIRS is a set of pairs $\{(p_1, n_1), (p_2, n_2), \ldots, (p_m, n_m)\}$. A pair (p_i, n_i) is referred to as an *interaction obligation*. The word "obligation" is used in order to emphasize that an implementation of a specification is required to fulfill every interaction obligation of the specification. This semantic model makes it possible to distinguish between potential and mandatory choice (see section 7.3.5).

A trace is a (finite or infinite) sequence of events $\langle e_1, e_2, \ldots, e_i, \ldots \rangle$. Let $\mathcal{H}$ be the trace universe. For each interaction obligation (p_i, n_i) we have that $p_i \subseteq \mathcal{H}$ is interpreted as positive traces and $n_i \subseteq \mathcal{H}$ as negative traces, and all interaction obligations are independent of each other. The union $p_i \cup n_i$ needs not exhaust the trace universe, so $\mathcal{H} \setminus (p_i \cup n_i)$ may differ from $\emptyset$ and is referred to as inconclusive traces. Further we allow specifications to be inconsistent, i.e., we may have that $p_i \cap n_i \neq \emptyset$.

In the following we assume the semantic model of STAIRS, and regard the STAIRS semantics as a correct formalization of sequence diagrams. This means, e.g., that the correctness of our operational semantics is evaluated with respect to the denotational semantics of STAIRS.

5.2 Syntax

The graphical notation of sequence diagrams is not suited as a basis for defining semantics, and the abstract syntax of the UML standard contains more information than we need for the task. Instead we apply a simpler abstract syntax. This is an event centric syntax in which the weak sequential operator seq is employed as the basic construct for combining diagram fragments.

The atom of a sequence diagram is the *event*. An event consists of a *message* and a *kind* where the kind decides whether it is the *transmit* or the *receive* event of the message. A message is a *signal*, which represents the contents of the message, together with the addresses of the transmitter and the receiver. Formally a signal is a label, and we let $\mathcal{S}$ denote the set of all signals. The transmitters and receivers are lifelines. Let $\mathcal{L}$ denote the set of all lifelines. Further let $\mathcal{G}$ denote the set of all gates. We treat gates as a special kind of lifelines, hence we have $\mathcal{G} \subset \mathcal{L}$. Gates are conceptually different from lifelines, but in this way definitions involving lifelines carry over to gates, and this has some practical advantages.

5.2.1 Messages and events

A message m is defined as a triple

$$(s, t, r) \in \mathcal{S} \times \mathcal{L} \times \mathcal{L}$$

of signal s, transmitter t and receiver r. $\mathcal{M}$ denotes the set of all messages. We sometimes use the signal to denote the message when the lifelines are implicitly given by, or not relevant in the context. On messages we define a transmitter function $tr.\_ \in \mathcal{M} \to \mathcal{L}$ and a receiver function $re.\_ \in \mathcal{M} \to \mathcal{L}$:

$$tr.(s, t, r) \stackrel{\mathsf{def}}{=} t \qquad re.(s, t, r) \stackrel{\mathsf{def}}{=} r$$

We let $\mathcal{K} = \{!, ?\}$ be the set of kinds, where ! represents transmit and ? represents receive. An event e is then a pair of a kind and a message:

$$(k, m) \in \mathcal{K} \times \mathcal{M}$$

$\mathcal{E}$ denotes the set of all events. We sometimes write $!s$ and $?s$ as shorthand for the events $(!, (s, t, r))$ and $(?, (s, t, r))$. On events we define a kind function $k.\_ \in \mathcal{E} \to \mathcal{K}$ and a message function $m.\_ \in \mathcal{E} \to \mathcal{M}$:

$$k.(k, m) \stackrel{\mathsf{def}}{=} k \qquad m.(k, m) \stackrel{\mathsf{def}}{=} m$$

We let the transmitter and receiver functions also range over events, $tr.\_, re.\_ \in \mathcal{E} \to \mathcal{L}$:

$$tr.(k, m) \stackrel{\mathsf{def}}{=} tr.m \qquad re.(k, m) \stackrel{\mathsf{def}}{=} re.m$$

Further we define a lifeline function $l.\_ \in \mathcal{E} \to \mathcal{L}$ that returns the lifeline of an event and a function $l^{-1}.\_ \in \mathcal{E} \to \mathcal{L}$ that returns the inverse lifeline of an event (i.e. the receiver of its message if its kind is transmit and the transmitter of its message if its kind is receive), which may be a gate (since, as explained above, gates are defined as a special kind of lifelines):

$$l.e \stackrel{\mathsf{def}}{=} \begin{cases} tr.e & \textbf{if } k.e = ! \\ re.e & \textbf{if } k.e = ? \end{cases} \qquad l^{-1}.e \stackrel{\mathsf{def}}{=} \begin{cases} tr.e & \textbf{if } k.e = ? \\ re.e & \textbf{if } k.e = ! \end{cases}$$

5.2.2 Diagrams

A sequence diagram is built up of events, the binary operators seq, par, strict, alt and xalt, and the unary operators opt, refuse, neg, assert and loop. Related to the terminology of the graphical syntax, the operators represent combined fragments and their arguments the operands. In addition we let skip represent the empty sequence diagram. Let $\mathcal{D}$ be the set of all syntactically correct sequence diagrams. $\mathcal{D}$ is defined recursively as the least set such that

$$\begin{array}{rcl}
 & & \mathsf{skip} \in \mathcal{D} \\
e \in \mathcal{E} & \Rightarrow & e \in \mathcal{D} \\
d_1, d_2 \in \mathcal{D} & \Rightarrow & d_1 \ \mathsf{seq}\ d_2 \in \mathcal{D} \wedge d_1 \ \mathsf{par}\ d_2 \in \mathcal{D} \wedge \\
 & & d_1 \ \mathsf{strict}\ d_2 \in \mathcal{D} \ \wedge \ d_1 \ \mathsf{alt}\ d_2 \in \mathcal{D} \wedge \\
 & & d_1 \ \mathsf{xalt}\ d_2 \in \mathcal{D} \\
d \in \mathcal{D} & \Rightarrow & \mathsf{neg}\ d \in \mathcal{D} \wedge \mathsf{refuse}\ d \in \mathcal{D} \wedge \\
 & & \mathsf{opt}\ d \in \mathcal{D} \wedge \mathsf{assert}\ d \in \mathcal{D} \\
I \subseteq (\mathbb{N} \cup \{0, \infty\}) \wedge d_i \in \mathcal{D} \textbf{ for } i \in I & \Rightarrow & \mathsf{alt}_{i \in I}\, d_i \in \mathcal{D} \\
d \in \mathcal{D} \wedge I \subseteq (\mathbb{N} \cup \{0, \infty\}) & \Rightarrow & \mathsf{loop}\ I\ d \in \mathcal{D} \\
d \in \mathcal{D} \wedge n \in (\mathbb{N} \cup \{0, \infty\}) & \Rightarrow & \mathsf{loop}\langle n \rangle\ d \in \mathcal{D}
\end{array}$$

with some additional constraints defined below.

Composition of diagrams by matching and elimination of gates are outside the scope of this presentation. Note therefore that this is a grammar for sequence diagrams and not composition as such.

The operator $\mathsf{alt}_{i \in I}$ is referred to as generalized alt. In the definition of this operator and the two loops we have that $\mathbb{N}$ is the set of non-zero natural numbers and ∞ is a number greater than all other numbers and has the property $\infty - 1 = \infty$. The intention behind $\mathsf{loop}\ I\ d$ is that d should be looped any number $n \in I$ times. The operator $\mathsf{loop}\langle n \rangle$ is referred to as loop with counter as the intention of $\mathsf{loop}\langle n \rangle\ d$ is that d should be looped n times.

As can be expected, we have associativity of seq, par, strict, alt and xalt. We also have commutativity of par, alt and xalt. Proofs with respect to the denotational semantics can be found in [70]. Furthermore, the empty sequence diagram skip is the identity element of seq, par and strict. The combination of skip and loop is discussed in section 8.3.1. We assume the following precedence of the operators: refuse, assert, loop, neg, opt, seq, strict, par, alt, xalt such that refuse binds stronger than assert, etc.

Note that an event at a gate cannot be part of a diagram. Hence, the representation of diagram d in figure 5.1 is:

$$d = (?, (a, g, i))\ \mathsf{seq}\ ((!, (b, i, j))\ \mathsf{seq}\ (?, (b, i, j))\ \mathsf{alt}\ (!, (c, i, j))\ \mathsf{seq}\ (?, (c, i, j)))$$

A number of functions that we will need later is defined over the syntax of diagrams. The function $ll.\_ \in \mathcal{D} \to \mathbb{P}(\mathcal{L})$ returns the set of lifelines present in a diagram. For $e \in \mathcal{E}$ and $d, d_1, d_2, d_i \in \mathcal{D}$ the function is defined recursively:

$$\begin{array}{rcl}
ll.e & \stackrel{\mathsf{def}}{=} & \{l.e\} \\
ll.\mathsf{skip} & \stackrel{\mathsf{def}}{=} & \emptyset \\
ll.(\mathsf{op}\ d) & \stackrel{\mathsf{def}}{=} & ll.d, \ \mathsf{op} \in \{\mathsf{neg}, \mathsf{refuse}, \mathsf{opt}, \mathsf{assert}, \mathsf{loop}\} \\
ll.(d_1\ \mathsf{op}\ d_2) & \stackrel{\mathsf{def}}{=} & ll.d_1 \cup ll.d_2, \ \mathsf{op} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{strict}, \mathsf{alt}, \mathsf{xalt}\} \\
ll.(\mathsf{alt}_{i \in I}\, d_i) & \stackrel{\mathsf{def}}{=} & \bigcup_{i \in I} ll.d_i
\end{array}$$

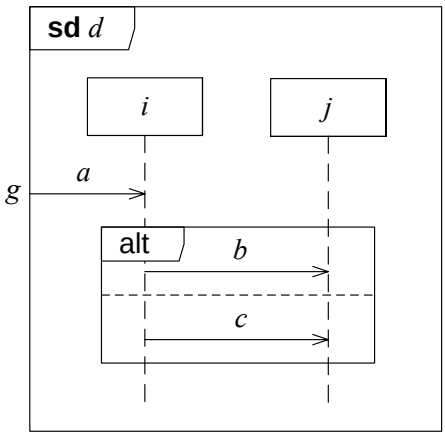


Figure 5.1: Sequence diagram

The function $gates.\_ \in \mathcal{D} \to \mathbb{P}(\mathcal{G})$ returns the gates of a diagram:

$$
\begin{aligned}
gates.e &\stackrel{\text{def}}{=} \begin{cases} \{l^{-1}.e\} & \textbf{if } l^{-1}.e \in \mathcal{G} \\ \emptyset & \textbf{if } l^{-1}.e \notin \mathcal{G} \end{cases} \\
gates.\mathsf{skip} &\stackrel{\text{def}}{=} \emptyset \\
gates.(\mathsf{op}\ d) &\stackrel{\text{def}}{=} gates.d,\ \mathsf{op} \in \{\mathsf{neg}, \mathsf{refuse}, \mathsf{opt}, \mathsf{assert}, \mathsf{loop}\} \\
gates.(d_1\ \mathsf{op}\ d_2) &\stackrel{\text{def}}{=} gates.d_1 \cup gates.d_2,\ \mathsf{op} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{strict}, \mathsf{alt}, \mathsf{xalt}\} \\
gates.(\mathsf{alt}_{i \in I}\, d_i) &\stackrel{\text{def}}{=} \bigcup_{i \in I} gates.d_i
\end{aligned}
$$

The function $msg.\_ \in \mathcal{D} \to \mathbb{P}(\mathcal{M})$ returns the messages of a diagram:

$$
\begin{aligned}
msg.e &\stackrel{\text{def}}{=} \{m.e\} \\
msg.\mathsf{skip} &\stackrel{\text{def}}{=} \emptyset \\
msg.(\mathsf{op}\ d) &\stackrel{\text{def}}{=} msg.d,\ \mathsf{op} \in \{\mathsf{neg}, \mathsf{refuse}, \mathsf{opt}, \mathsf{assert}, \mathsf{loop}\} \\
msg.(d_1\ \mathsf{op}\ d_2) &\stackrel{\text{def}}{=} msg.d_1 \cup msg.d_2,\ \mathsf{op} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{strict}, \mathsf{alt}, \mathsf{xalt}\} \\
msg.(\mathsf{alt}_{i \in I}\, d_i) &\stackrel{\text{def}}{=} \bigcup_{i \in I} msg.d_i
\end{aligned}
$$

The function $ev.\_ \in \mathcal{D} \to \mathbb{P}(\mathcal{E})$ returns the events of a diagram:

$$
\begin{aligned}
ev.e &\stackrel{\text{def}}{=} \{e\} \\
ev.\mathsf{skip} &\stackrel{\text{def}}{=} \emptyset \\
ev.(\mathsf{op}\ d) &\stackrel{\text{def}}{=} ev.d,\ \mathsf{op} \in \{\mathsf{neg}, \mathsf{refuse}, \mathsf{opt}, \mathsf{assert}, \mathsf{loop}\} \\
ev.(d_1\ \mathsf{op}\ d_2) &\stackrel{\text{def}}{=} ev.d_1 \cup ev.d_2,\ \mathsf{op} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{strict}, \mathsf{alt}, \mathsf{xalt}\} \\
ev.(\mathsf{alt}_{i \in I}\, d_i) &\stackrel{\text{def}}{=} \bigcup_{i \in I} ev.d_i
\end{aligned}
$$

5.2.3 Syntax constraints

We impose some restrictions on the set of syntactically correct sequence diagrams $\mathcal{D}$. First, we assert that a given event should syntactically occur only once in a diagram:

$$d_1\ \mathsf{op}\ d_2 \in \mathcal{D} \Rightarrow ev.d_1 \cap ev.d_2 = \emptyset,\ \mathsf{op} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{strict}, \mathsf{alt}, \mathsf{xalt}\}$$

Secondly, if both the transmitter and the receiver lifelines of a message are present in a diagram, then both the transmit event and receive event of that message must be in the diagram:

$$d \in \mathcal{D} \wedge m \in msg.d \wedge tr.m \in ll.d \wedge re.m \in ll.d \Rightarrow (!, m) \in ev.d \wedge (?, m) \in ev.d$$

Thirdly, if both the transmit event and the receive event of a message are present in a diagram, they have to be inside the same argument of the same high-level operator.

$$\begin{array}{l} \mathsf{op}\ d \in \mathcal{D} \wedge m \in msg.d \\ \quad \Rightarrow (tr.m \notin \mathcal{G} \Rightarrow (!, m) \in ev.d) \wedge (re.m \notin \mathcal{G} \Rightarrow (?, m) \in ev.d), \\ \quad \mathsf{op} \in \{\mathsf{refuse}, \mathsf{assert}, \mathsf{loop}\} \end{array}$$

$$\begin{array}{l} d_1\ \mathsf{op}\ d_2 \in \mathcal{D} \wedge m \in msg.d_i \\ \quad \Rightarrow (tr.m \notin \mathcal{G} \Rightarrow (!, m) \in ev.d_i) \wedge (re.m \notin \mathcal{G} \Rightarrow (?, m) \in ev.d_i), \\ \quad \mathsf{op} \in \{\mathsf{alt}, \mathsf{xalt}\},\ i \in \{1, 2\} \end{array}$$

A way of phrasing this constraint is that in the graphical notation, messages are not allowed to cross the frame of a high-level operator or the dividing line between the arguments of a high-level operator.

Fourthly, we assume there is no syntactical repetition of gates in a diagram:

$$d_1\ \mathsf{op}\ d_2 \in \mathcal{D} \Rightarrow gates.d_1 \cap gates.d_2 = \emptyset,\ \mathsf{op} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{strict}, \mathsf{alt}, \mathsf{xalt}\}$$

Finally, we do not allow the operators refuse and assert to be empty:

$$\begin{array}{c} \mathsf{refuse\ skip} \notin \mathcal{D} \\ \mathsf{assert\ skip} \notin \mathcal{D} \end{array}$$

5.3 Denotational semantics

In this section we give the denotational semantics of STAIRS as presented in [68, 70, 155]. In [70] the semantics is given as a timed three event version. In our work, and hence in the following presentation we apply the untimed two event version of [68,155].

The operators strict, generalized alt and loop with counter, defined in sections 5.3.5, 5.3.6 and 5.3.9, respectively, are additions we have made to STAIRS and are hence not found in the above cited works. In addition we have given a new, however semantically equivalent, definition of loop I.[1]

5.3.1 Well formed traces

STAIRS is a trace based semantics. A trace is a sequence of events, used to represent a system run. In each trace, a transmit event should always be ordered before the corresponding receive event. We let $\mathcal{H}$ denote the set of all traces that complies with this requirement. Formally this is handled by a constraint on traces.

We use $\langle\rangle$ to denote the empty trace, and $\_|\_$ and $\#\_$ to denote, respectively, truncation and length of traces. For concatenation of traces, filtering of traces, and filtering of pairs of traces, we have the functions $\_^\frown\_$, $\_\textcircled{S}\_$, and $\_\textcircled{T}\_$, respectively. $t_1{}^\frown t_2$ denotes a

[1] The proof of this equivalence may be found as lemma 16 in appendix B.3 of [115].

trace that equals t_1 if t_1 is infinite or otherwise is the trace consisting of t_1 immediately followed by t_2. The filtering function $\_⊛\_$ is used to filter away elements from a trace. By $A⊛t$ we denote the trace obtained from the trace t by removing all elements in t that are not in the set of elements A. For example, we have that:

$$\{1,3\}⊛\langle 1,1,2,1,3,2\rangle = \langle 1,1,1,3\rangle$$

The function $\_⊚\_$ filters pairs of traces with respect to pairs of elements in the same way as $\_⊛\_$ filters traces with respect to elements. For any set of pairs of elements P and pair of traces u, by $P⊚u$ we denote the pair of traces obtained from u by 1) truncating the longest trace in u at the length of the shortest traces in u if the two sequences are of unequal length; 2) for each $j \in [1 \ldots k]$, where k is the length of the shortest trace in u, selecting or deleting the two elements at index j in the two traces, depending on whether the pair of these elements is in the set P or not. For example, we have that:

$$\{(1,f),(1,g)\}⊚(\langle 1,1,2,1,2\rangle, \langle f,f,f,g,g\rangle) = (\langle 1,1,1\rangle, \langle f,f,g\rangle)$$

Definitions of $\_⊛\_$ and $\_⊚\_$ are found as definition 4 on page 267 and definition 7 on page 275 in appendix A.2.

The semantic constraint on traces may now be formalized. It asserts that for all traces $h \in \mathcal{H}$, if at a point in a trace we have an receive event of a message, then up to that point we must have had at least as many transmits of that message as receives.

$$\forall i \in [1..\#h] : k.h[i] = ! \Rightarrow \#(\{(!, m.h[i])\}⊛h|_i) > \#(\{(?, m.h[i])\}⊛h|_i) \tag{5.1}$$

5.3.2 The denotation function

As explained above, the semantic model of STAIRS is a set of interaction obligations, where an interaction obligation is a pair (p, n) of sets of traces where the first set is interpreted as a set of positive traces and the second set is a set of negative traces. In the following we let $\mathcal{O}$ be the set of all interaction obligations. The semantics of sequence diagrams is defined by a function

$$[\![\_]\!] \in \mathcal{D} \to \mathbb{P}(\mathcal{O}) \tag{5.2}$$

that for any sequence diagram d yields a set $[\![d]\!]$ of interaction obligations.

The semantics of the empty sequence diagram skip is defined as only the empty positive trace:

$$[\![\mathsf{skip}]\!] \stackrel{\mathsf{def}}{=} \{(\{\langle\rangle\}, \emptyset)\} \tag{5.3}$$

For a sequence diagram consisting of a single event e, its semantics is given by:

$$[\![e]\!] \stackrel{\mathsf{def}}{=} \{(\{\langle e\rangle\}, \emptyset)\} \tag{5.4}$$

5.3.3 Weak sequencing

Weak sequencing is the implicit composition mechanism combining constructs of a sequence diagram, and is defined by the seq operator. First, we define weak sequencing of trace sets:

$$s_1 \gtrsim s_2 \stackrel{\mathsf{def}}{=} \{h \in \mathcal{H} \mid \exists h_1 \in s_1, h_2 \in s_2 : \forall l \in \mathcal{L} : e.l⊛h = e.l⊛h_1 \frown e.l⊛h_2\} \tag{5.5}$$

where $e.l$ denotes the set of events that may take place on the lifeline l:

$$e.l \stackrel{\text{def}}{=} \{e \in \mathcal{E} \mid l.e = l\}$$

Weak sequencing of interaction obligations is defined as:

$$(p_1, n_1) \gtrsim (p_2, n_2) \stackrel{\text{def}}{=} (p_1 \gtrsim p_2, (n_1 \gtrsim p_2) \cup (n_1 \gtrsim n_2) \cup (p_1 \gtrsim n_2)) \tag{5.6}$$

Using this definition, the semantics of the seq operator is defined as:

$$[\![\, d_1 \,\mathsf{seq}\, d_2 \,]\!] \stackrel{\text{def}}{=} \{o_1 \gtrsim o_2 \mid o_1 \in [\![\, d_1 \,]\!] \wedge o_2 \in [\![\, d_2 \,]\!]\} \tag{5.7}$$

5.3.4 Interleaving

The par operator represents a parallel merge of traces such that the order of the events in the merged traces is preserved in the resulting traces. In order to define par, we first define parallel execution on trace sets:

$$\begin{aligned} s_1 \parallel s_2 \stackrel{\text{def}}{=} \quad & \{h \in \mathcal{H} \mid \exists p \in \{1,2\}^\infty : \\ & \quad \pi_2((\{1\} \times \mathcal{E})^{\circledcirc}(p, h)) \in s_1 \wedge \\ & \quad \pi_2((\{2\} \times \mathcal{E})^{\circledcirc}(p, h)) \in s_2\} \end{aligned} \tag{5.8}$$

In this definition, we make use of an oracle, the infinite sequence p, to resolve the non-determinism in the interleaving. It determines the order in which events from traces in s_1 and s_2 are sequenced. π_2 is a projection operator returning the second element of a pair. Formally:

$$\pi_i(s_1, s_2) \stackrel{\text{def}}{=} s_i,\ i \in \{1, 2\}$$

Parallel merge of interaction obligations is defined as:

$$(p_1, n_1) \parallel (p_2, n_2) \stackrel{\text{def}}{=} (p_1 \parallel p_2, (n_1 \parallel p_2) \cup (n_1 \parallel n_2) \cup (p_1 \parallel n_2)) \tag{5.9}$$

The semantics of the par operator is then defined as:

$$[\![\, d_1 \,\mathsf{par}\, d_2 \,]\!] \stackrel{\text{def}}{=} \{o_1 \parallel o_2 \mid o_1 \in [\![\, d_1 \,]\!] \wedge o_2 \in [\![\, d_2 \,]\!]\} \tag{5.10}$$

5.3.5 Strict sequencing

The strict operator specifies strict sequencing. Strict sequencing of trace sets is defined as:

$$s_1 \succ s_2 \stackrel{\text{def}}{=} \{h \in \mathcal{H} \mid \exists h_1 \in s_1, h_2 \in s_2 : h = h_1 \frown h_2\} \tag{5.11}$$

Strict sequencing of interaction obligations is defined as:

$$(p_1, n_1) \succ (p_2, n_2) \stackrel{\text{def}}{=} (p_1 \succ p_2, (n_1 \succ p_2) \cup (n_1 \succ n_2) \cup (p_1 \succ n_2)) \tag{5.12}$$

The semantics of the strict operator is defined as:

$$[\![\, d_1 \ \mathsf{strict}\ d_2 \,]\!] \stackrel{\text{def}}{=} \{o_1 \succ o_2 \mid o_1 \in [\![\, d_1 \,]\!] \wedge o_2 \in [\![\, d_2 \,]\!]\} \tag{5.13}$$

5.3.6 Choice

The alt operator defines potential choice. The semantics is the inner union of the interaction obligations:

$$[\![d_1 \,\mathsf{alt}\, d_2]\!] \stackrel{\mathsf{def}}{=} \{(p_1 \cup p_2, n_1 \cup n_2) \mid (p_1, n_1) \in [\![d_1]\!] \wedge (p_2, n_2) \in [\![d_2]\!]\} \tag{5.14}$$

The unary operator opt is defined as a derived operator from alt:

$$\mathsf{opt}\ d \stackrel{\mathsf{def}}{=} \mathsf{skip}\ \mathsf{alt}\ d \tag{5.15}$$

Let $\biguplus$ be a generalization inner union of interaction obligations such that

$$\biguplus_{i \in I} O_i \stackrel{\mathsf{def}}{=} \{\Big(\bigcup_{i \in I} p_i, \bigcup_{i \in I} n_i\Big) \mid \forall i \in I : (p_i, n_i) \in O_i\} \tag{5.16}$$

where O_i are sets of interaction obligations. We define the generalized alt in the following fashion:

$$[\![\underset{i \in I}{\mathsf{alt}}\, d_i]\!] \stackrel{\mathsf{def}}{=} \biguplus_{i \in I} [\![d_i]\!] \tag{5.17}$$

The xalt operator defines mandatory choice. All implementations must be able to handle every interaction obligation, so the the semantics of xalt is defined as the union of interaction obligations defined by its arguments:

$$[\![d_1 \,\mathsf{xalt}\, d_2]\!] \stackrel{\mathsf{def}}{=} [\![d_1]\!] \cup [\![d_2]\!] \tag{5.18}$$

5.3.7 Negative behavior

To specify negative behavior we use the operator refuse. The reason for this is that the UML operator neg is ambiguous and has (at least) two distinct interpretations. Following [155] we define refuse as one of the interpretations and let neg represent the other as a derived operator. refuse is defined as:

$$[\![\mathsf{refuse}\ d]\!] \stackrel{\mathsf{def}}{=} \{(\emptyset, p \cup n) \mid (p, n) \in [\![d]\!]\} \tag{5.19}$$

Note that refuse defines all positive and negative behavior as negative, so nested refuse does not yield positive behavior. The neg operator is defined as a derived operator from refuse:

$$\mathsf{neg}\ d \stackrel{\mathsf{def}}{=} \mathsf{skip}\ \mathsf{alt}\ \mathsf{refuse}\ d \tag{5.20}$$

5.3.8 Universal behavior

Universal behavior is specified by the operator assert:

$$[\![\mathsf{assert}\ d]\!] \stackrel{\mathsf{def}}{=} \{(p, n \cup (\mathcal{H} \setminus p)) \mid (p, n) \in [\![d]\!]\} \tag{5.21}$$

Note that assert defines all behavior that is not positive as negative and hence empties the set of inconclusive traces.

5.3.9 Iteration

The semantics of loop is defined by a semantic loop construct μ_n, where n is the number of times the loop should be iterated. For $n \in \mathbb{N} \cup \{0\}$ (finite loop), μ_n is defined as:

$$\begin{array}{lcl} \mu_0\ O & \stackrel{\mathsf{def}}{=} & \{(\{\langle\rangle\}, \emptyset)\} \\ \mu_1\ O & \stackrel{\mathsf{def}}{=} & O \\ \mu_n\ O & \stackrel{\mathsf{def}}{=} & O \gtrsim \mu_{n-1}\ O \quad \textbf{if } n > 1 \end{array} \tag{5.22}$$

With this loop construct, $\mathsf{loop}\langle n \rangle$ is defined as:

$$[\![\ \mathsf{loop}\langle n \rangle\ d\ ]\!] \stackrel{\mathsf{def}}{=} \mu_n\ [\![\ d\ ]\!] \tag{5.23}$$

$\mathsf{loop}\ I$ is defined a derived operator:

$$\mathsf{loop}\ I\ d \stackrel{\mathsf{def}}{=} \mathop{\mathsf{alt}}_{i \in I} \mathsf{loop}\langle i \rangle\ d \tag{5.24}$$

In order to define infinite loop we need some auxiliary definitions. We define the chains of interaction obligation as:

$$\begin{array}{l} chains(O) \stackrel{\mathsf{def}}{=} \{\bar{o} \in \mathcal{O}^\infty \mid \bar{o}[1] \in O \wedge \\ \qquad\qquad \forall j \in \mathbb{N} : \exists o \in O : \bar{o}[j+1] = \bar{o}[j] \gtrsim o\} \end{array} \tag{5.25}$$

From a chain of interaction obligations we define its chains of positive and negative traces:

$$\begin{array}{l} pos(\bar{o}) \stackrel{\mathsf{def}}{=} \{\bar{t} \in \mathcal{H}^\infty \mid \forall j \in \mathbb{N} : \bar{t}[j] \in \pi_1(\bar{o}[j]) \wedge \\ \qquad\qquad \exists t \in \mathcal{H} : \bar{t}[j+1] \in \{\bar{t}[j]\} \gtrsim \{t\}\} \end{array} \tag{5.26}$$

$$\begin{array}{l} negs(\bar{o}) \stackrel{\mathsf{def}}{=} \{\bar{t} \in \mathcal{H}^\infty \mid \exists i \in \mathbb{N} : \forall j \in \mathbb{N} : \bar{t}[j] \in \pi_2(\bar{o}[j+i-1]) \wedge \\ \qquad\qquad \exists t \in \mathcal{H} : \bar{t}[j+1] \in \{\bar{t}[j]\} \gtrsim \{t\}\} \end{array} \tag{5.27}$$

For every chain of traces $\bar{t}$ we have that for all $l \in \mathcal{L}$, the sequence

$$e.l \circledS \bar{t}[1], e.l \circledS \bar{t}[2], e.l \circledS \bar{t}[3], \ldots$$

is a chain ordered by the prefix relation $\sqsubseteq$. We let $\sqcup_l \bar{t}$ denote the least upper bound of this chain and define the approximations of $\bar{t}$ as:

$$\sqcup \bar{t} \stackrel{\mathsf{def}}{=} \{h \in \mathcal{H} \mid \forall l \in \mathcal{L} : e.l \circledS h = \sqcup_l \bar{t}\} \tag{5.28}$$

For a chain of interaction obligations we define:

$$\sqcup \bar{o} \stackrel{\mathsf{def}}{=} \left(\bigcup_{\bar{t} \in pos(\bar{o})} \sqcup \bar{t}, \bigcup_{\bar{t} \in negs(\bar{o})} \sqcup \bar{t} \right) \tag{5.29}$$

and let infinite loop be defined as:

$$\mu_\infty\ O \stackrel{\mathsf{def}}{=} \{\sqcup \bar{o} \mid \bar{o} \in chains(O)\} \tag{5.30}$$

The UML standard defines two loops $\mathsf{loop}(n)$ and $\mathsf{loop}(n, m)$, where n is the minimum number and m the maximum number of iterations. We may define these as derived operators:

$$\begin{array}{rcl} \mathsf{loop}(n)\ d & \stackrel{\mathsf{def}}{=} & \mathsf{loop}\ [n..\infty]\ d \\ \mathsf{loop}(n, m)\ d & \stackrel{\mathsf{def}}{=} & \mathsf{loop}\ [n..m]\ d \end{array}$$

5.4 Refinement

Refinement is to make a specification more concrete and closer to implementation by removing underspecification. For sequence diagrams, which are partial specifications, this means two things:

1. Making the specification more specific by removing inconclusive behavior, i.e., making inconclusive behavior positive or negative.

2. Making the specification more concrete by removing non-determinism, i.e., making positive behavior negative.

In the following we formalize a number of refinement relations, based on [70, 159], to capture various aspects of the intuitive understanding of refinement. The refinement relations are relations $\_ \rightsquigarrow_x \_ \in \mathcal{D} \times \mathcal{D} \to \mathbb{B}$ between sequence diagrams, where x represents decorations we apply for distinguishing the different relations. For diagrams d, d', the relation $d \rightsquigarrow_x d'$ is to be read "d' is an x-refinement of d" or "d' x-refines d."

5.4.1 Refinement of interaction obligations

In order to define the refinement relations for diagrams we first need to define refinement of interaction obligations. The most general refinement, which captures both the above items, is the relation $\_ \rightsquigarrow_r \_ \in \mathcal{O} \times \mathcal{O} \to \mathbb{B}$:

$$(p, n) \rightsquigarrow_r (p', n') \stackrel{\mathsf{def}}{=} p \subseteq p' \cup n' \wedge n \subseteq n'$$

Each of the above items give rises to a special case of this general refinement. Item 1 is referred to as (proper) supplementing and is formalized by a relation $\_ \rightsquigarrow_s \_ \in \mathcal{O} \times \mathcal{O} \to \mathbb{B}$:

$$(p, n) \rightsquigarrow_s (p', n') \stackrel{\mathsf{def}}{=} (n \subset n' \wedge p \subseteq p') \vee (n \subseteq n' \wedge p \subset p')$$

Item 2 is called (proper) narrowing and is formalized by the relation $\_ \rightsquigarrow_n \_ \in \mathcal{O} \times \mathcal{O} \to \mathbb{B}$:

$$(p, n) \rightsquigarrow_n (p', n') \stackrel{\mathsf{def}}{=} p' \subset p \wedge n' = n \cup (p \setminus p')$$

In some situations it may by useful to assert that no new functionality should be added to the specification, or in other words that the set of positive behavior is not allowed to increase. This is formalized in a relation $\_ \rightsquigarrow_{rr} \_ \in \mathcal{O} \times \mathcal{O} \to \mathbb{B}$ called restricted refinement:

$$(p, n) \rightsquigarrow_{rr} (p', n') \stackrel{\mathsf{def}}{=} (p, n) \rightsquigarrow_r (p', n') \wedge p' \subseteq p$$

5.4.2 Refinement of sequence diagrams

The denotation $[\![d]\!]$ of a diagram d characterizes a set of interaction obligations. In the most general case we say that a diagram d' refines diagram d if and only if all interaction obligations of $[\![d]\!]$ are refined by an interaction obligation in $[\![d']\!]$. This is formalized by the general refinement relation $\_ \rightsquigarrow_g \_ \in \mathcal{D} \times \mathcal{D} \to \mathbb{B}$:

$$d \rightsquigarrow_g d' \stackrel{\mathsf{def}}{=} \forall o \in [\![d]\!] : \exists d' \in [\![o']\!] : o \rightsquigarrow_r o'$$

Likewise we define a restricted general refinement relation $\_ \rightsquigarrow_{rg} \_ \in \mathcal{D} \times \mathcal{D} \to \mathbb{B}$ based on restricted refinement of interaction obligations:

$$d \rightsquigarrow_{rg} d' \stackrel{\mathsf{def}}{=} \forall o \in [\![d]\!] : \exists o' \in [\![d']\!] : o \rightsquigarrow_{rr} o'$$

Both general refinement and restricted general refinement allow a diagram d', which refines a diagram d, to include new interaction obligations, i.e. interaction obligations that are not refinements of interaction obligations of d. In some situations it may be useful to disallow this option of adding interaction obligations in a refinement. Limited refinement $\_ \rightsquigarrow_{l} \_ \in \mathcal{D} \times \mathcal{D} \to \mathbb{B}$ and restricted limited refinement $\_ \rightsquigarrow_{rl} \_ \in \mathcal{D} \times \mathcal{D} \to \mathbb{B}$ are refinement relations that formalize this limitation with respect to general refinement and restricted general refinement:

$$\begin{aligned} d \rightsquigarrow_{l} d' &\stackrel{\mathsf{def}}{=} d \rightsquigarrow_{g} d' \wedge \forall o' \in [\![d']\!] : \exists o \in [\![d]\!] : o \rightsquigarrow_{r} o' \\ d \rightsquigarrow_{rl} d' &\stackrel{\mathsf{def}}{=} d \rightsquigarrow_{rg} d' \wedge \forall o' \in [\![d']\!] : \exists o \in [\![d]\!] : o \rightsquigarrow_{rr} o' \end{aligned}$$

Chapter 6

Maude

As stated earlier, one of our main contributions is an operational semantics and its implementation in Maude. In this chapter we give a brief introduction to Maude, in order to present the capabilities of Maude and to help the reader better understand our Maude implementation.

Maude is two things; a specification language and an interpreter of this language. We start by presenting the language in a example driven fashion (section 6.1), and then introducing the main functionality of the interpreter (section 6.2). Also we present some of the reflexive properties of Maude (section 6.3). We do not apply the full language, and only present the parts that we use. For a more thorough introduction we refer to [35, 121, 139]. For more formal treatments of rewrite logic we refer to [28, 123].

6.1 Specification language

Maude specifications are organized in modules, and modules contain definitions of sorts, operators, equations and rules. A module may be a *functional module* or a *system module*. The only real difference is that system modules may contain rewrite rules, while functional modules may not. As an example we define a module for lists of natural numbers:

```
mod LIST is

endm
```

This is a system module. A functional module would start with `fmod` and end with `endfm`. A module may include other models. Maude contains a number of predefined modules for common data types like the natural numbers or the Booleans, so we may add the line

```
include NAT .
```

to the module above to include the predefined definitions of the natural numbers.

Data types in Maude are called sorts. In our example we add the following lines to our module:

```
sort List .
subsort Nat < List .
```

This means we define a sort `List`, and define the sort `Nat` from the included module `NAT` as a subsort of `List`. This way a single natural number is also regarded as a singleton list.

Operators are functions from a Cartesian product of sorts to a single sort. Special operators called constructors are used to define the values of sorts. The members of a sort are those terms (called ground terms) that are built from the constructor operators. We define the constructors of our lists as:

```
op nil : -> List [ctor] .
op _,_ : List List -> List [ctor assoc id: nil] .
```

Here `nil` is meant to represent the empty list, and since it has zero arguments it is a constant. The keyword `ctor` is used to assert that it a constructor. As list concatenation operator we use comma in mixfix notation. This is also a constructor, and the statement `assoc id: nil` defines it as associative and with `nil` as the identity element. In addition to these two constructors, the constructors of the sort `Nat` become constructors of the sort `List` because `Nat` is a subsort of `List`. Now we have that, e.g.,

```
2 , 4 , 3, 1
```

and

```
0 , nil , nil , 1
```

are ground terms of the sort `List`. Operators that are not constructors may be seen as functions that operate on sorts. If we want a length function of the lists we define its signature as:

```
op length : List -> Nat .
```

A operator is defined by a set of equations. The operator `length` may for example be defined as

```
eq length(nil) = 0 .
eq length(N , L) = 1 + length(L) .
```

if we first have defined `N` as a variable of type `Nat` and `L` as a variable of type `List`. Equations may be conditional. An alternative definition the upper equation above could be:

```
ceq length(L) = 0 if L == nil .
```

Condition may also be of the form `L :: List`, which checks for sort membership, or `N := M * K`, which assigns a value to a variable. Also the condition may be a conjunction of several sub-conditions. This is called the functional part of Maude, and as we can see, it is quite similar to a functional programming language.

In addition to functional definitions we may define rewrite rules, which also may be conditional or unconditional. As an example we define a rule for sorting a list of natural numbers:

```
crl [sort] : L , N , N' , L' => L , N' , N , L' if N' < N .
```

In the condition of a rule, we may also have rewrites as sub-conditions.

Put together the whole module becomes:

```
mod LIST is
  including NAT .

  sort List .
  subsort Nat < List .

  op nil : -> List [ctor] .
  op _,_ : List List -> List [ctor assoc id: nil] .
  op length : List -> Nat .

  vars N N' : Nat .
  vars L L' : List .

  eq length(nil) = 0 .
  eq length(N , L) = 1 + length(L) .

  crl [sort] : L , N , N' , L' => L , N' , N , L' if N' < N .
endm
```

This module defines lists of natural numbers, a length function for these lists, and a rule for sorting the lists.

6.2 Interpreter

The Maude interpreter has three main commands; `reduce`, `rewrite` and `search`. The command `reduce` applies the equations from left to right on the provided term and reduces it until it becomes a ground term. If, for example, we issue the command

```
reduce in LIST : length(2 , 4 , 3 , 1) .
```

the Maude interpreter will produce the result:

```
rewrites: 9 in 2283767000ms cpu (0ms real) (0 rewrites/second)
result NzNat: 4
```

Because we have defined the list concatenation operator to be associative and have `nil` as the identity element, all reductions will happen modulo associativity and identity.

The `rewrite` command applies the rules of a module, from left to right, on a given term, until no more rewrites are possible. Giving the command

```
rewrite in LIST : 2 , 4 , 3 , 1 .
```

will result in:

```
rewrites: 31 in 2283764000ms cpu (1ms real) (0 rewrites/second)
result List: 1,2,3,4
```

Between each rewrite, the interpreter will reduce as much as possible, and also rewrites are applied modulo associativity, commutativity and identity if specified in the constructors. If the left side of more than one rule matches the term, the rules will be applied semi non-deterministically. (By semi non-deterministically we mean that the

interpreter does not have real non-determinism in application of the rules, but simulates non-determinism by chosing which rule to apply based on the, usually arbitrary, order in which the rules appear in the Maude code.)

`serach` makes a breadth first search on all possible sequences of rewrites and returns the terms matching the search criteria. Let us say we want to know what the result may be after only one rewrite. We specify the search

```
search in LIST : 2 , 4 , 3 , 1 =>1 L:List .
```

and the Maude interpreter returns two solutions:

```
Solution 1 (state 1)
states: 2  rewrites: 5 in 24958464280ms cpu (1ms real)
   (0 rewrites/second)
L --> 2,3,4,1

Solution 2 (state 2)
states: 3  rewrites: 13 in 24958464280ms cpu (1ms real)
   (0 rewrites/second)
L --> 2,4,1,3

No more solutions.
states: 3  rewrites: 17 in 24958464280ms cpu (1ms real)
   (0 rewrites/second)
```

If we specify the search

```
search in LIST : 2 , 4 , 3 , 1 =>! N:Nat , L:List , N':Nat
   such that N:Nat > N':Nat .
```

we ask the interpreter to do as many rewrites as possible and look for solutions where the first number of the list is greater than the last number. Luckily, the interpreter gives us this result:

```
No solution.
states: 8  rewrites: 111 in 4767057229ms cpu (2ms real)
   (0 rewrites/second)
```

In addition we may specify searches using `=>*` and `=>+`, in which the interpreter will look for solutions after, respectively, zero or more rewrites and one or more rewrites.

6.3 Reflexivity

Maude is a reflexive language. This means that modules, equations, etc. have meta-representations and that these meta-representations may be manipulated as any other terms. To access the meta-level the predefined module `META-LEVEL` must be included. In the example we define a module `LIST-META` in which we can perform manipulations of the meta-level of the module `LIST`:

```
mod LIST-META is
  including LIST .
  including META-LEVEL .
endm
```

We can now get a meta-representation of a term by using the predefined function `upTerm`:

```
reduce in LIST-META : upTerm(0,1) .
rewrites: 1 in -17206397862ms cpu (0ms real) (~ rewrites/second)
result GroundTerm: '_`,_['0.Zero,'s_['0.Zero]]
```

The result is a term of type `GroundTerm`. `'0.Zero` is the meta-representation of `0`, `'s_['0.Zero]` is the meta-representation of `1` (actually `s 0`, the successor function `s_` applied to `0`) and `'_`,_[...]` is the meta-representation of the list concatenation operator `_,_`. The command

```
reduce in LIST-META : upModule('LIST, false) .
```

gives us a meta-representation of the module `LIST`. The parameter `false` asserts that included modules should not be included in the meta-representation, whereas using the parameter `true` instead would give a meta-representation that included the meta-representation of the included modules. The result is as a term of type `Module`:

```
result Module: mod 'LIST is
  including 'BOOL .
  including 'NAT .
  sorts 'List .
  subsort 'Nat < 'List .
  op '_`,_ : 'List 'List -> 'List [assoc ctor id('nil.List)] .
  op 'length : 'List -> 'Nat [none] .
  op 'nil : nil -> 'List [ctor] .
  none
  eq 'length['nil.List] = '0.Zero [none] .
  eq 'length['_`,_['N:Nat,'L:List]]
     = '_+_['s_['0.Zero],'length['L:List]] [none] .
  crl '_`,_['L:List,'N:Nat,'N':Nat,'L':List]
     => '_`,_['L:List,'_`,_['N':Nat,'_`,_['N:Nat,'L':List]]]
     if '_<_['N':Nat,'N:Nat] = 'true.Bool [label('sort)] .
endm
```

There is a number of functions that may be applied at the meta-level. `metaReduce` and `metaRewrite` work as the commands `reduce` and `rewrite`, but at the meta-level. `metaApply` and `metaXapply` make one rewrite on a term with a specified rule, the former at the top level of the term and the latter somewhere within the term. `metaSearch` does search at the meta-level. In the following we will look closer at `metaApply`. The arguments of this function is the meta-representation of a module, the meta-representation of a term to be rewritten, the name of a rule, a possible empty set of substitutions of values for the variables in the rule and a counter used to differentiate between different possible applications of the rule. The function returns a meta-representation of the term after the rewrite, the type of the resulting term and the substitutions of the values to variables used in the rewrite. By applying the `sort` rule to the list `3, 2, 1` and varying the counter we will see that there are two ways to apply the rule on the list:

```
reduce in LIST-META : metaApply(upModule('LIST, false), upTerm(3,2,1),
    'sort, none, 0) .
```

```
rewrites: 5 in -17329373866ms cpu (6ms real) (~ rewrites/second)
result ResultTriple:
       {'_‘,_['s_^2['0.Zero],'s_^3['0.Zero],'s_['0.Zero]],'List,
        'L':List <- 's_['0.Zero] ;
        'L:List <- 'nil.List ;
        'N':Nat <- 's_^2['0.Zero] ;
        'N:Nat <- 's_^3['0.Zero]}

reduce in LIST-META : metaApply(upModule('LIST, false), upTerm(3,2,1),
    'sort, none, 1) .
rewrites: 5 in -13449941426ms cpu (0ms real) (~ rewrites/second)
result ResultTriple:
       {'_‘,_['s_^3['0.Zero],'s_['0.Zero],'s_^2['0.Zero]],'List,
        'L':List <- 'nil.List ;
        'L:List <- 's_^3['0.Zero] ;
        'N':Nat <- 's_['0.Zero] ;
        'N:Nat <- 's_^2['0.Zero]}

reduce in LIST-META : metaApply(upModule('LIST, false), upTerm(3,2,1),
    'sort, none, 2) .
rewrites: 3 in 6603855571ms cpu (1ms real) (0 rewrites/second)
result ResultTriple?: (failure).ResultTriple?
```

The Maude interpreter will typically use the first possible rewrite, which in this case is swapping 3 and 2 in the list. If we for some reason would like to sort the list in such a way that the sorting starts at the end of the list instead of the beginning, we could use `metaApply` and define a rewriting strategy that always uses the rewrite that does the rightmost possible swap of numbers in the list. One way this can be done is by adding the following definitions to the module `LIST-META`:

```
sort ResultList .
subsort ResultTriple < ResultList .

op empty : -> ResultList [ctor] .
op _&_ : ResultList ResultList -> ResultList [ctor assoc id: empty] .

op mySort : List -> List .
op sorted : List -> Bool .
op getPossibilities : List Nat -> ResultList .
op findRightmost : ResultList -> ResultTriple .
op getTail : ResultTriple -> List .
op findTail : Substitution -> List .

op error : -> [List] .

var L : List .
vars N N' : Nat .
var RT? : ResultTriple? .
vars RT RT' : ResultTriple .
var RL : ResultList .
var V : Variable .
var T : Term .
```

```
var S : Substitution .

ceq mySort(L) = L if sorted(L) .
ceq mySort(L)
    = mySort(downTerm(getTerm(findRightmost(getPossibilities(L,0))),
                      error))
    if not sorted(L) .

eq sorted(nil) = true .
eq sorted(N) = true .
eq sorted(N,N',L) = N <= N' and sorted(N',L) .

ceq getPossibilities(L,N) =
   if RT? :: ResultTriple then
      RT? & getPossibilities(L,s N)
   else
      empty
   fi
   if RT? := metaApply(upModule('LIST,false),upTerm(L),'sort,none,N) .

eq findRightmost(RT) = RT .
ceq findRightmost(RT & RL) =
      if length(getTail(RT)) < length(getTail(RT')) then
         RT
      else
         RT'
      fi
      if RT' := findRightmost(RL) .

eq getTail(RT) = findTail(getSubstitution(RT)) .

eq findTail(none) = nil .
eq findTail(V <- T ; S) =
  if V == 'L':List then
     downTerm(T,error)
  else
     findTail(S)
  fi .
```

The function `mySort` will sort a list of numbers by using the `sort` rule, but in such a way that the swapping of numbers always occur as the rightmost possible swap in the list.

Part III

Operational semantics

Chapter 7

Introduction to operational semantics

One of the main contributions of this thesis is an operational semantics for UML sequence diagrams. In this chapter we motivate this (section 7.1) and have a look at the differences between denotational semantics and operational semantics (section 7.2). Further we discuss some challenges specific for defining operational semantics for UML sequence diagrams (section 7.3).

The operational semantics itself is defined in chapter 8. After that we present meta-strategies for execution of the operational semantics in chapter 9, implementation of the operational semantics in chapter 10 and other approaches to defining operational semantics for sequence diagrams in chapter 11.

7.1 Motivation

When sequence diagrams are used to get a better understanding of the system through modeling, as system documentation or as means of communication between stakeholders of the system, it is important that the precise meaning of the diagrams is understood; in other words, there is need for a well-defined semantics. Sequence diagrams may also be put to further applications, such as simulation, testing and other kinds of automated analysis. This further increases the need for a formalized semantics; not only must the people who make and read diagrams have a common understanding of their meaning, but also the makers of methods and tools for analyzing the diagrams must share this understanding.

Methods of analysis like simulation and testing are in their nature operational; they are used for investigating what will happen when a system is executing. When developing techniques for such analysis, not only do we need to understand the precise meaning of a specification, we also need to understand precisely the executions that are specified. This motivates formalization of semantics in an operational style.

In our work we have aimed at defining an operational semantics that is close to the UML standard in both syntax and semantics. Further we have aimed at facilitating ease of extension and modification when adapting the semantics to different interpretations and applications of sequence diagrams.

7.2 Definition

STAIRS provides a denotational semantics for sequence diagrams, and we have motivated the need for operational semantics. Before we proceed, let us have a look at the distinction between operational semantics and denotational semantics. David A. Schmidt [161, p. 3] suggests the following:

> The *operational semantics* method uses an interpreter to define a language. The meaning of a program in the language is the evaluation history that the interpreter produces when it interprets the program. The evaluation history is a sequence of internal configurations [...]
>
> The *denotational semantics* method maps a program directly to its meaning, called its *denotation*. The denotation is usually a mathematical value, such as a number or function. No interpreters are used; a *valuation function* maps a program directly to its meaning.

As a methodology for language development he suggests that "a denotational semantics is defined to give the meaning of the language" and that "the denotational definition is implemented using an operational definition" [161, p. 4]. Hoare and He [75, p. 258] describe more explicitly the notion of an operational semantics:

> An *operational* semantics of a programming language is one that defines not the observable overall effect of a program but rather suggests a complete set of possible individual steps which may be taken in its execution. The observable effect can then be obtained by embedding the steps into an iterative loop [...]

Taken together, these two descriptions suggest that formalizing an operational semantics of a language is to define an interpreter for the language. The formal definition of the interpreter describes every step that can be made in the execution of the language in such a way that the executions are in conformance with the meaning of the language as defined by a denotational semantics. In our case the input to the interpreter is a sequence diagram represented in an abstract syntax. The output of the interpreter is a trace of events representing an execution.

7.3 Challenges

There are a number of challenges connected with making semantics for sequence diagrams, and obviously, choices must be made where the UML standard is ambiguous. This section is dedicated to looking into these challenges in more detail.

7.3.1 Partial vs. complete specification

An old discussion related to sequence diagrams and MSCs is whether a specification consisting of a set of diagrams represents the full behavior of the specified system or just examples of the behavior. In the former case we say that the set of sequence diagrams is a complete specification, and in the latter case a partial specification.

The UML standard states clearly that sequence diagrams are partial specifications since "[the union of valid and invalid traces] need not be the whole universe of traces" [134, p. 480]. However, the operator assert can be used for specifying universal behavior

in the sense that all behavior other than the behavior which is enclosed by the assert is made negative.

This makes defining an operational semantics somewhat more complicated than in the case of complete specifications. It rules out solutions such as just viewing sequence diagrams as, or translating them to, other formalisms for making complete specifications, such as transition systems. In effect, the trace universe becomes a quantity that must be considered in the definition of the semantics. This excludes a solution where the negative traces are considered as just the complement of the positive traces.

7.3.2 Global vs. local view

In a certain sense, a sequence diagram has a global and a local view at the same time. The lifelines of a sequence diagram do not synchronize and represent processes that execute independently. This is the local view of the sequence diagram. At the same time, the syntax allows operators that cover several lifelines, and hence provide a global view. The best example is the alternative operator alt without guards, i.e. a non-deterministic choice. Since the lifelines are independent and do not synchronize, one of the lifelines may start executing the arguments of the operator before the other lifelines. The choice, however, is global in the sense that all lifelines must choose the same argument when resolving the choice. In [89, p. 369], Jonsson and Padilla make the same observation:

> The transition system will [...] maintain a local state for each instance [...] The execution [...] may follow either of two paths, depending on which alternative is chosen in the [alt]. The crucial point here is that all three instances must choose the same alternative even if they do not arrive simultaneously to the [alt] in their execution. [...] Thus, [one instance's] entry into the first alternative has global consequences in that it "forces" the other instances to also choose the first alternative.

The result of this is that the semantics must reflect this duality; the semantics must provide both the local and the global view of the specifications. A discussion of this problem is also found in [94].

7.3.3 Weak sequencing

The weak sequencing operator seq is the implicit operator for composing sequence diagrams. The operator defines a partial order of the events in a diagram, such that the order along each lifeline and the causal relationship between the transmission and the reception of messages are preserved while any other ordering of events is arbitrary. Further there is no implicit synchronization between the lifelines in a sequence diagram.

An operational semantics must characterize the step by step execution specified by the diagram. For each step, all enabled events must be selectable. This poses a challenge, since due to the weak sequencing, there may be enabled events at arbitrary depths in the syntactical term representing the diagram.

7.3.4 Negative behavior

UML 2.x allows negative behavior to be specified by the neg operator in sequence diagrams. In the semantic model, these behaviors end up in the set of negative traces (see section 4.2). In denotational semantics like STAIRS or the trace semantics of Harald Störrle [173,174] this is easily handled by manipulation of the traces and trace sets (even though these two formalizations do not agree on the interpretation of neg).

It is however not clear what negative behavior means in an operational semantics. If an operational semantics should describe a step by step the execution of a sequence diagram it is not clear how we should distinguish a positive execution from a negative execution. Immediate abortion, which is the naive solution, is not satisfactory for two reasons: 1) We may be interested in complete negative executions, and 2) we need to know that the execution was stopped because it reached a state in which it became negative. The alternative is some way of waving a flag. With this solution a meta-level at which the flag is seen and the behavior interpreted as negative is needed in order to assign meaning to negative behavior.

We cannot know at the execution level that we are in a negative execution, and it seems obvious that the only way of assigning meaning to negative behavior is by adding a meta-level at which we may interpret behavior as positive or negative.

Even though he does not need it for his denotational semantics, Störrle [174] mentions the option:

> One might also consider both the assert- and negate-operators as being meta-logical in the sense that they express properties of traces rather that defining or modifying traces.

The solution for Live Sequence Charts (LSC) (for more on LSC see section 11) is described in [62, p. 88]:

> A hot condition [...] must always be true. If an execution reaches a hot condition that evaluates to false this is a violation of the requirements, and the system should abort. For example, if we form an LSC from a prechart *Ch* and a main chart consisting of a single *false* hot condition, the semantics is that *Ch* can never occur. In other words, it is forbidden, an *anti-scenario*.

Even though they choose to abort a negative execution, this is guided by a meta-variable *Violating* in their operational semantics [61].

Cengarle and Knapp offer two ways of interpreting negative behavior. The first [30], is based on *satisfies* relations. Two such relations are provided: *positively satisfies*, $t \models_p S$, meaning that t is a positive trace of the diagram S, and *negatively satisfies*, $t \models_n S$, meaning that t is an negative trace of S. The second [31] is an operational semantics. Two transition relations $S \xrightarrow{e}_p S'$ and $S \xrightarrow{e}_n S'$ are provided, where the former means that specification S may produce event e in a positive execution and the latter that S may produce e in a negative execution. Their exact interpretation of negative behavior is not relevant in this discussion, but we notice that also their solutions resort to use of a meta-level to distinguish between positive and negative traces.

Whether it is obtained by waving a flag, like in the LSC semantics, or by defining an extra structure, as in the approach of Cengarle and Knapp, an extra meta-level is needed for giving an interpretation to neg and the related refuse operator.

7.3.5 Potential vs. mandatory choice

A final question is the ambiguity of the alternative operator alt, whether the non-determinism introduced by the alternative operator represents underspecification or real choices in the specified system.

It can be argued that the alternative operator is useful for underspecification, i.e. that it represents a design decision to be made later, but practitioners often interpret it as a choice between two alternatives that should both be present in the implementation. Both interpretations have some problems attached to them.

The first interpretation fits well with the observation that sequence diagrams are used for high-level specification and that several steps of refinement are needed on the way to implementation. From this viewpoint it makes sense to interpret an alternative as underspecification. A problem with this is that if a sequence diagram is interpreted as a partial specification, the specification becomes very weak; in effect every behavior is allowed if no negative behavior is specified. Another drawback is that it is not clear how to represent non-deterministic choices that should be preserved in the implementation. Such choices are essential in, e.g., specification of security properties [87].

It may be argued that the second interpretation is more intuitive because the choice operator is used for representing choices in the system and not design choices. The drawback is of course that the possibility of underspecification is then restricted.

In [68, 70] this ambiguity is resolved by interpreting alt in the first sense, as underspecification, and by introducing xalt (explicit alternative) as a choice operator in the second sense; a choice that represents a choice in the system.

It is however not possible to distinguish these two kinds of choices at the execution level; for a single execution it is irrelevant whether a choice is specified by an alt or an xalt. But the distinction is relevant with respect to, e.g., refinement and implementation, which means at the meta-level. As with the operator neg, an extra meta-level is needed for giving an interpretation of xalt and to distinguish between the two kinds of choice.

Chapter 8

Definition of operational semantics

In this chapter we present operational semantics for UML sequence diagrams. As far as possible the semantics is faithful to the standard and is sound and complete with respect to the denotational semantics of STAIRS. The semantics is defined so as to be easy to extend and modify. This allows us to give a "default" or "standard" interpretation, but also to experiment with the semantics and make variations on points unspecified by the standard. Specifically it has a formalized meta-level which allows definition of different execution strategies. It is not based on transformations to other formalisms, which makes it easy to work with.

Our solution to the challenges identified in section 7.3 is the combination of two transition systems, which we refer to as the *execution system* and the *projection system*. The execution system is a transition system over

$$[\_, \_] \in \mathcal{B} \times \mathcal{D} \tag{8.1}$$

where $\mathcal{B}$ represents the set of all states of the communication medium and $\mathcal{D}$ the set of all syntactically correct sequence diagrams. We let $\mathcal{EX} \stackrel{\mathsf{def}}{=} \mathcal{B} \times \mathcal{D}$, and refer to the elements of $\mathcal{EX}$ as execution states.

The projection system is a transition system over

$$\Pi(\_, \_, \_) \in \mathbb{P}(\mathcal{L}) \times \mathcal{B} \times \mathcal{D} \tag{8.2}$$

where $\mathbb{P}(\mathcal{L})$ is the powerset of the set of all lifelines. The projection system is used for finding enabled events at each stage of the execution and is defined recursively. This system handles the challenges related to weak sequencing and related to the global vs. the local view in sequence diagrams.

These two systems work together in such a way that for each step in the execution, the execution system updates the projection system by passing on the current state of the communication medium, and the projection system updates the execution system by selecting the event to execute and returning the state of the diagram after the execution of the event.

In section 8.1 we define the execution system. Section 8.2 is devoted to the communication medium, and in section 8.3 the projection system is defined. In section 8.4 we define fairness, and in section 8.5 we present soundness and completeness results. The meta-level is dealt with in chapter 9.

8.1 The execution system

The execution system has two rules. The first rule represents the execution of a single event and uses the projection system to find an enabled event to execute. It is defined as:

$$[\beta, d] \xrightarrow{e} [update(\beta, e), d'] \textbf{ if } \Pi(ll.d, \beta, d) \xrightarrow{e} \Pi(ll.d, \beta, d') \wedge e \in \mathcal{E} \tag{8.3}$$

In general we assume the structure of the communication medium, i.e. the means of communication, to be underspecified. The only requirement is that the following functions are defined:

- $add \in \mathcal{B} \times \mathcal{M} \to \mathcal{B}$: Adds a message.
- $rm \in \mathcal{B} \times \mathcal{M} \to \mathcal{B}$: Removes a message.
- $ready \in \mathcal{B} \times \mathcal{M} \to \mathbb{B}$: Returns **true** if the communication medium is in a state where it can deliver the message and **false** otherwise.

The function $update \in \mathcal{B} \times \mathcal{E} \to \mathcal{B}$ is defined as:

$$update(\beta, e) \stackrel{\text{def}}{=} \begin{cases} add(\beta, m.e) & \textbf{if } k.e = ! \\ rm(\beta, m.e) & \textbf{if } k.e = ? \end{cases} \tag{8.4}$$

Since transmitter and receiver information is embedded into the messages, these functions are sufficient. Usually we assume the most general communication model, i.e. no ordering on the messages. This means that, e.g., message overtaking is possible. Different communication models are described in section 8.2 below.

The second rule of the execution system executes silent events. The rules of the projection system handle the sequence diagram operators alt, xalt, refuse, assert and loop$\langle n \rangle$. (The operators opt, neg and loop I are derived operators, and therefore do not have their own rules.) Resolving these operators, such as choosing the branch of an alt, are considered silent events. We define the set of silent events to be

$$\mathcal{T} = \{\tau_{alt}, \tau_{xalt}, \tau_{refuse}, \tau_{assert}, \tau_{loop}\} \tag{8.5}$$

with $\mathcal{T} \cap \mathcal{E} = \emptyset$. The reason for introducing all these different silent events is that they give high flexibility in defining execution strategies by making the silent events and their kinds available at the meta-level. The rule is simple:

$$[\beta, d] \xrightarrow{\tau} [\beta, d'] \textbf{ if } \Pi(ll.d, \beta, d) \xrightarrow{\tau} \Pi(ll.d, \beta, d') \wedge \tau \in \mathcal{T} \tag{8.6}$$

The empty diagram skip cannot be rewritten, but we assert that it produces the empty trace, i.e.:

$$[\beta, \mathsf{skip}] \xrightarrow{\langle\rangle} [\beta, \mathsf{skip}] \tag{8.7}$$

This also means that execution terminates when skip is reached.

Throughout this thesis we apply the following shorthand notation:

- If there exists a sequence of transitions

  $$[\beta, d] \xrightarrow{e_1} [\beta_1, d_1] \xrightarrow{e_2} \cdots \xrightarrow{e_n} [\beta_n, d_n]$$

  we write

  $$[\beta, d] \xrightarrow{\langle e_1, e_2, \ldots, e_n \rangle} [\beta_n, d_n]$$

- We say that an event e is *enabled* in execution state $[\beta, d]$ iff there exists a transition
$$[\beta, d] \xrightarrow{e} [\beta', d']$$
for some β', d'.

- We write
$$[\beta, d] \xrightarrow{e}$$
for
$$\exists \beta', d' : [\beta, d] \xrightarrow{e} [\beta', d']$$

- If an event e is not enabled in $[\beta, d]$ we write
$$[\beta, d] \not\xrightarrow{e}$$

- If for all events e we have that e is not enabled in $[\beta, d]$ (i.e. no events are enabled in $[\beta, d]$) we write
$$[\beta, d] \not\rightarrow$$

8.2 Communication model

We assume that lifelines communicate by means of asynchronous message passing. This assumption is however not sufficient in order to fully define the operational semantics. We also need to make assumptions regarding the communication model, i.e. the mechanisms for transporting messages between lifelines. In [44], a classification of four general communication models for Message Sequence Charts are characterized:

- One FIFO[1] buffer for each message. This is equivalent to what may be called a random access buffer.

- One FIFO buffer for each (ordered) pair of lifelines. All messages from one lifeline to another will pass through a specific buffer.

- One FIFO buffer for each lifeline. All messages to a lifeline pass through the same buffer.

- One global FIFO buffer. All messages sent between all lifelines pass through the same buffer.

Further it is shown that the choice of communication model affects the semantics of the diagrams.

We adopt this classification, but do not make any decision on which of these communication models should be regarded as the "correct" one. Instead we define all four and let them be options for instantiation of the underspecified communication medium characterized in section 8.1. The "one FIFO for each message" model must be considered the most general of the models, and is also the model (implicitly) assumed by STAIRS. We therefore use this model, defined in section 8.2.1 below, as the default, and apply this communication model in our soundness and completeness proofs.

[1] First In First Out.

In the following sub-sections, we define the set of states of the communication medium $\mathcal{B}$, and the functions $add \in \mathcal{B} \times \mathcal{M} \to \mathcal{B}$, $rm \in \mathcal{B} \times \mathcal{M} \to \mathcal{B}$ and $ready \in \mathcal{B} \times \mathcal{M} \to \mathbb{B}$. For all four we assume arbitrary size, i.e. that no messages are discarded due to overflow.

Because events at gates are not present in the syntactical representation of a sequence diagram, we must assume that messages sent from gates are available in the execution of the diagram (see also appendix A.1). For the first two of our comminication models we can handle this by defining a function $initC \in \mathcal{D} \to \mathcal{B}$ that returns a initial state of the commnuication medium containing the necessary messages for execution of the diagram. For the other two communication models it does not make sense to define such a function becuase the specific syntactic representation of the diagrams will affect the order of the messages in the initial state of the communication medium, and therefore also the order or the execution of events. In other words, different syntactic representations of semantically equivalent diagrams may give different inital states of the communication medium. In these communication models the initial state of the communication medium should be explicitely specified.

8.2.1 One FIFO for each message

In this communication model, each message is assumed to have its own channel from the transmitter to the receiver, something that allows for message overtaking. Because we assume all messages to be unique and the messages to contain transmitter and receiver information, there is no need for defining the channels or buffers explicitly – at any point in time a set of messages is sufficient to represent the state of the communication medium. Hence, the set of states of the communication medium are defined as a powerset of the set of messages:

$$\mathcal{B} \stackrel{\mathsf{def}}{=} \mathbb{P}(\mathcal{M})$$

The functions add, rm and $ready$ for manipulating the state of the communication medium can then be defined using set operations:

$$\begin{aligned} add(\beta, m) &\stackrel{\mathsf{def}}{=} \beta \cup \{m\} \\ rm(\beta, m) &\stackrel{\mathsf{def}}{=} \beta \setminus \{m\} \\ ready(\beta, m) &\stackrel{\mathsf{def}}{=} m \in \beta \end{aligned} \tag{8.8}$$

The function $initC$ for this communication medium is defined as follows:[2]

$$\begin{aligned} initC(e) &\stackrel{\mathsf{def}}{=} \begin{cases} \{m.e\} & \textbf{if } k.e = ? \wedge tr.e \in \mathcal{G} \\ \emptyset & \textbf{if } k.e = ! \vee tr.e \notin \mathcal{G} \end{cases} \\ initC(\mathsf{skip}) &\stackrel{\mathsf{def}}{=} \emptyset \\ initC(\mathsf{op}\ d) &\stackrel{\mathsf{def}}{=} initC(d),\ \mathsf{op} \in \{\mathsf{refuse}, \mathsf{assert}\} \\ initC(d_1\ \mathsf{op}\ d_2) &\stackrel{\mathsf{def}}{=} initC(d_1) \cup initC(d_2),\ \mathsf{op} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{strict}, \mathsf{alt}, \mathsf{xalt}\} \\ initC(\mathsf{loop}\langle n \rangle\ d) &\stackrel{\mathsf{def}}{=} initC(d) \cup initC(\mathsf{loop}\langle n-1 \rangle\ d) \end{aligned}$$

[2] We assume in this definition that messages from different iterations of a loop is distinguishable. See appendix A.3 for details.

8.2.2 One FIFO for each pair of lifelines

In this communication model, each pair of lifelines in a diagram is assumed to have a directed channel in each direction through which the messages are transported. The set of states of FIFO buffers $\mathcal{F}$ is defined as sequences of messages:

$$\mathcal{F} \stackrel{\text{def}}{=} \mathcal{M}^*$$

Each state of the communication model is represented as a function from pair of lifelines to FIFO buffers, and the set of states of the communication medium $\mathcal{B}$ is then defined as the set of functions from pairs of lifelines to FIFOs:

$$\mathcal{B} \stackrel{\text{def}}{=} \mathcal{L} \times \mathcal{L} \to \mathcal{F}$$

We can think of a state of the commuinication medium β as a mapping from pairs of lifelines to FIFOs:

$$\beta = \{(l_1, l_1) \mapsto b_{11}, (l_1, l_2) \mapsto b_{12}, \ldots, (l_2, l_1) \mapsto b_{21}, \ldots\}$$

Let $\mathit{emptyComm} \in \mathcal{B}$ be the set of empty channels, defined as

$$\mathit{emptyComm} \stackrel{\text{def}}{=} \{(l, l') \mapsto \langle\rangle \mid l, l' \in \mathcal{L}\}$$

Further, let $\beta[(l, l') \mapsto b]$ be the state β updated with the mapping $(l, l') \mapsto b$, such that:

$$(\beta[(l, l') \mapsto b])(l'', l''') = \begin{cases} b & \textbf{if } l = l'' \wedge l' = l''' \\ \beta(l'', l''') & \textbf{otherwise} \end{cases}$$

We then define a function $\mathit{comm} \in \mathcal{M} \to \mathcal{B}$ that returns the state of the commuinication medium that only contains a single message m:

$$\mathit{comm}(m) \stackrel{\text{def}}{=} \mathit{emptyComm}[(tr.m, re.m) \mapsto \langle m \rangle]$$

For manipulation of the state of the communication medium we first define an auxillary "FIFO concatenation" function $\_ \ll \_ \in \mathcal{B} \times \mathcal{B} \to \mathcal{B}$:

$$\beta_1 \ll \beta_2 \stackrel{\text{def}}{=} \{(l, l') \mapsto \beta_1(l, l') \frown \beta_2(l, l') \mid l, l' \in \mathcal{L}\}$$

The functions *add*, *rm* and *ready* for manipulating the state of the communication medium are defined as follows:

$$\begin{array}{rcl} \mathit{add}(\beta, m) & \stackrel{\text{def}}{=} & \beta \ll \mathit{comm}(m) \\ \mathit{rm}(\beta, m) & \stackrel{\text{def}}{=} & \begin{cases} \langle\rangle & \textbf{if } \beta(tr.m, re.m) = \langle\rangle \\ \beta[(tr.m, re.m) \mapsto \mathit{tail}(\beta(tr.m, re.m))] & \textbf{if } \mathit{ready}(\beta, m) \\ \beta & \textbf{if } \neg \mathit{ready}(\beta, m) \end{cases} \\ \mathit{ready}(\beta, m) & \stackrel{\text{def}}{=} & \mathit{head}(\beta(tr.m, re.m)) = m \end{array}$$

The functions $\mathit{head} \in \mathcal{M}^* \to \mathcal{M}$ and $\mathit{tail} \in \mathcal{M}^* \to \mathcal{M}^*$ are functions that return the head and tail of a sequence in the usual fashion:

$$\begin{array}{rcl} \mathit{head}(\langle m \rangle \frown b) & \stackrel{\text{def}}{=} & m \\ \mathit{tail}(\langle m \rangle \frown b) & \stackrel{\text{def}}{=} & b \end{array}$$

We define the initial state of the communication medium in the following way:

$$\begin{aligned}
initC.e &\stackrel{\text{def}}{=} \begin{cases} comm(m.e) & \textbf{if } k.e = ? \wedge tr.e \in \mathcal{G} \\ emptyComm & \textbf{if } k.e = ! \vee tr.e \notin \mathcal{G} \end{cases} \\
initC.\mathsf{skip} &\stackrel{\text{def}}{=} emptyComm \\
initC.(\mathsf{op}\ d) &\stackrel{\text{def}}{=} initC.d,\ \mathsf{op} \in \{\mathsf{refuse}, \mathsf{assert}\} \\
initC.(d_1\ \mathsf{op}\ d_1) &\stackrel{\text{def}}{=} initC.d_1 \ll initC.d_2,\ \mathsf{op} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{strict}, \mathsf{alt}, \mathsf{xalt}\} \\
initC.(\mathsf{loop}\langle n \rangle\ d) &\stackrel{\text{def}}{=} initC.d \ll initC.(\mathsf{loop}\langle n-1 \rangle\ d)
\end{aligned}$$

Because all gate names are unique, this means the initial state of the communication medium will contain a separate FIFO for each message sent from a gate to lifelines in the diagram. The exception is messages from gates inside loops, where the FIFO will be a sequence of messages whose length corresponds to the number of iterations of the loop.

8.2.3 One FIFO for each lifeline

This communication model assumes that each lifeline has a single channel through which all incomming messages are transported, regardless of the transmitters of the messages. Similar to the above communication model, a state of the communication medium is a function from lifelines to FIFOs. The difference is that the function only takes as argument the receiver lifeline. The set of states of the communication medium is defined as

$$\mathcal{B} \stackrel{\text{def}}{=} \mathcal{L} \to \mathcal{F}$$

where $\mathcal{F}$ is defined as above.

We also define similar *emptyComm*, *comm* and $\_ \ll \_$ functions for this communication medium:

$$emptyComm \stackrel{\text{def}}{=} \{l \mapsto \langle\rangle \mid l \in \mathcal{L}\}$$

$$comm(m) \stackrel{\text{def}}{=} emptyComm[re.m \mapsto \langle m \rangle]$$

$$\beta_1 \ll \beta_2 \stackrel{\text{def}}{=} \{l \mapsto \beta_1(l)^\frown\beta_2(l) \mid l \in \mathcal{L}\}$$

The functions *add*, *rm* and *ready* for manipulation of the state of the communication medium are defined as follows:

$$\begin{aligned}
add(\beta, m) &\stackrel{\text{def}}{=} \beta \ll comm(m) \\
rm(\beta, m) &\stackrel{\text{def}}{=} \begin{cases} \langle\rangle & \textbf{if } \beta(re.m) = \langle\rangle \\ \beta[re.m \mapsto tail(\beta(re.m))] & \textbf{if } ready(\beta, m) \\ \beta & \textbf{if } \neg ready(\beta, m) \end{cases} \\
ready(\beta, m) &\stackrel{\text{def}}{=} head(\beta(re.m)) = m
\end{aligned}$$

8.2.4 One global FIFO

In this communication model, all messages are transported through a single channel regardless of sender and receiver. This channel is represented by a FIFO buffer, and the states of the communication medium are then equal the states of the buffer. Further no transmitter and receiver information is needed.

The states of the communication medium is hence defined as the set of sequences of messages:

$$\mathcal{B} \stackrel{\mathsf{def}}{=} \mathcal{F}$$

The functions *add*, *rm* and *ready* for manipulating the state of the communication medium are defined by use of sequnce manipulation operators:

$$\begin{array}{rcl} add(\beta, m) & \stackrel{\mathsf{def}}{=} & \beta^\frown\langle m\rangle \\ rm(\beta, m) & \stackrel{\mathsf{def}}{=} & \begin{cases} \langle\rangle & \textbf{if } \beta = \langle\rangle \\ tail(\beta) & \textbf{if } ready(\beta, m) \\ \beta & \textbf{if } \neg ready(\beta, m) \end{cases} \\ ready(\beta, m) & \stackrel{\mathsf{def}}{=} & head(\beta) = m \end{array} \tag{8.9}$$

8.3 The projection system

In the following sub-sections we present the rules of the projection system for each of the syntactic constructs.

8.3.1 The empty diagram

It is not possible to rewrite $\Pi(L, \beta, \mathsf{skip})$. skip is the identity element of seq, par and strict, and we therefore have that skip seq d, d seq skip, skip par d, d par skip, skip strict d and d strict skip are treated as identical to d. We can also note that skip alt skip = skip and skip xalt skip = skip. We do not allow the constructions refuse skip and assert skip.

loop$\langle\infty\rangle$ skip is more problematic (see section 8.3.9 for the definition of loop$\langle\infty\rangle$). Seen as a program, this construct is similar to the java fragment `while(true) { }`, i.e., a program that produces nothing and never terminates. When related to the denotational semantics, however, the semantics of loop$\langle\infty\rangle$ skip should be the empty trace $\langle\rangle$, since the denotational semantics characterize observation after infinite time. A simple solution would be to syntactically disallow the construct all together. Because we do not want to make too many syntactic constraints, and because we want to stay close to the denotational semantics we choose to let loop$\langle\infty\rangle$ skip reduce to skip, even though this may be seen as counter-intuitive from an operational point of view.

8.3.2 Event

The simplest case is the diagram consisting of only one event e. In this case the system delivers the event if the event is enabled, given the set L of lifelines and the state of the communication medium. This means first that the event must belong to one of the lifelines in the set L, and secondly that the event either must be a transmit event or its message must be available in the communication medium. The need for L will be evident in the definition of rules for seq below.

$$\begin{array}{l} \Pi(L, \beta, e) \xrightarrow{e} \Pi(L, \beta, \mathsf{skip}) \\ \qquad \textbf{if } l.e \in L \wedge (k.e = ! \vee ready(\beta, m.e)) \end{array} \tag{8.10}$$

8.3.3 Weak sequencing

The weak sequencing operator seq defines a partial order on the events in a diagram; the ordering of events on each lifeline and between the transmit and receive of a message is preserved, but all other ordering of events is arbitrary. Because of this, there may be enabled events in both the left and the right argument of a seq if there are lifelines present in the right argument of the operator that are not present in the left argument. This leads to two rules for the seq operator.

If there is an overlap between the given set of lifelines and the lifelines of the left hand side of the seq, it means that the lifelines in this intersection may have enabled events on the left hand side only. Hence, with respect to these lifelines, the system must look for enabled events in the left operand.

$$\begin{aligned}\Pi(L, \beta, d_1 \ \mathsf{seq}\ d_2) &\xrightarrow{e} \Pi(L, \beta, d_1' \ \mathsf{seq}\ d_2) \\ &\textbf{if } ll.d_1 \cap L \neq \emptyset \wedge \Pi(ll.d_1 \cap L, \beta, d_1) \xrightarrow{e} \Pi(ll.d_1 \cap L, \beta, d_1')\end{aligned} \tag{8.11}$$

If the lifelines of the left hand side do not exhaust the given set of lifelines, it means there are lifelines that are only present on the right hand side, and that there may be enabled events on the right hand side of the operator. This means the system may look for enabled events on the right hand side of the seq, but only with respect to the lifelines not represented on the left hand side.

$$\begin{aligned}\Pi(L, \beta, d_1 \ \mathsf{seq}\ d_2) &\xrightarrow{e} \Pi(L, \beta, d_1 \ \mathsf{seq}\ d_2') \\ &\textbf{if } L \setminus ll.d_1 \neq \emptyset \wedge \Pi(L \setminus ll.d_1, \beta, d_2) \xrightarrow{e} \Pi(L \setminus ll.d_1, \beta, d_2')\end{aligned} \tag{8.12}$$

Note that the two conditions $ll.d_1 \cap L \neq \emptyset$ and $ll.d_1 \setminus L \neq \emptyset$ are not mutually exclusive. If both these condition are true at the same time there may be enabled events on both sides of the seq operator. In such a case the rules may be applied in arbitrary order.

The transitions of the system are used as conditions in the recursion of these rules. Therefore the rules will not be applied unless an enabled event is found deeper in the recursion. Because of this, the system will always be able to return an enabled event if enabled events exist.

8.3.4 Interleaving

The parallel operator par specifies interleaving of the events from each of its arguments; in other words parallel merge of the executions of each of the arguments. The rules of par are similar to the rules of seq, but simpler since we do not have to preserve any order between the two operands. One of the operands is chosen arbitrarily. As with the seq rules, the use of transitions as the conditions of the rules ensures that an enabled event is found if enabled events exist.

$$\begin{aligned}\Pi(L, \beta, d_1 \ \mathsf{par}\ d_2) &\xrightarrow{e} \Pi(L, \beta, d_1' \ \mathsf{par}\ d_2) \\ &\textbf{if } \Pi(ll.d_1 \cap L, \beta, d_1) \xrightarrow{e} \Pi(ll.d_1 \cap L, \beta, d_1')\end{aligned} \tag{8.13}$$

$$\begin{aligned}\Pi(L, \beta, d_1 \ \mathsf{par}\ d_2) &\xrightarrow{e} \Pi(L, \beta, d_1 \ \mathsf{par}\ d_2') \\ &\textbf{if } \Pi(ll.d_2 \cap L, \beta, d_2) \xrightarrow{e} \Pi(ll.d_2 \cap L, \beta, d_2')\end{aligned} \tag{8.14}$$

8.3.5 Strict sequencing

The strict operator defines a strict ordering of events, so that all events on the left side of the operator must be executed before the events on the right side. Obviously then, only one rule is needed:

$$\begin{aligned}\Pi(L, \beta, d_1 \text{ strict } d_2) &\xrightarrow{e} \Pi(L, \beta, d_1' \text{ strict } d_2) \\ \textbf{if } ll.d_1 \cap L \neq \emptyset \ \wedge \ &\Pi(ll.d_1 \cap L, \beta, d_1) \xrightarrow{e} \Pi(ll.d_1 \cap L, \beta, d_1')\end{aligned} \tag{8.15}$$

As we see, this rule is essentially equal to the left side rule for seq.

8.3.6 Choice

The rules for the two choice operators end the recursion; the choice is resolved and a silent event is produced. By resolving the choice instead of looking for events deeper down, we ensure that the same choice is made for all the lifelines covered by a choice operator.

$$\Pi(L, \beta, d_1 \text{ alt } d_2) \xrightarrow{\tau_{alt}} \Pi(L, \beta, d_k) \textbf{ if } L \cap ll.(d_1 \text{ alt } d_2) \neq \emptyset, \quad \textbf{for} \quad k \in \{1, 2\} \tag{8.16}$$

$$\Pi(L, \beta, d_1 \text{ xalt } d_2) \xrightarrow{\tau_{xalt}} \Pi(L, \beta, d_k) \textbf{ if } L \cap ll.(d_1 \text{ xalt } d_2) \neq \emptyset, \quad \textbf{for} \quad k \in \{1, 2\} \tag{8.17}$$

The rules for alt and xalt are identical except for the kind of silent event they produce. This reflects the fact that the operators are indistinguishable at the execution level. Since they produce different events, the kind of the choice is available at the meta-level and this is used in the definition of meta-strategies. As with rule (8.10), there is a condition that makes sure the rules are only applied if the event produced is relevant to the set L of lifelines.

The operator opt is treated as an alt with one empty argument (see section 5.3.6).

8.3.7 Negative behavior

The rule for refuse produces a silent event when a refuse operator is resolved. This flags that the execution becomes negative and makes this information available at the meta-level.

$$\Pi(L, \beta, \text{refuse } d) \xrightarrow{\tau_{refuse}} \Pi(L, \beta, d) \textbf{ if } ll.(\text{refuse } d) \cap L \neq \emptyset \tag{8.18}$$

Similar to the choice rules, we have the condition that $ll.(\text{refuse } d) \cap L = ll.d \cap L \neq \emptyset$ to ensure that the produced event is relevant to the set of lifelines L.

The operator neg is treated as a choice between a negative branch and an empty branch by making it a derived operator from alt and refuse (see section 5.3.7).

8.3.8 Universal behavior

The rule assert simply resolves the operator and produces a silent event. The silent event makes the information that the execution has entered an "asserted" region available to the meta-level.

$$\Pi(L, \beta, \text{assert } d) \xrightarrow{\tau_{assert}} \Pi(L, \beta, d) \textbf{ if } ll.(\text{assert } d) \cap L \neq \emptyset \tag{8.19}$$

Again we have the same condition on the set of lifelines.

8.3.9 Iteration

Informally, in $\mathsf{loop}\ I\ d$ there is a non-deterministic choice between the numbers of I, which is formalized by making $\mathsf{loop}\ I$ a derived operator of generalized alt and $\mathsf{loop}\langle n\rangle$ (see section 5.3.9). If $n \in I$ is picked, d should be iterated n times.

$\mathsf{loop}\langle n\rangle\ d$ is a loop with a counter. In the rule the counter is decreased by one for each iteration. We also produce a silent event to represent the iteration of a loop. Even though iteration of a loop in itself is not the most relevant information at the meta-level, it may be useful for defining execution strategies, for example if we want to give iteration of the loop low priority.

$$\Pi(L, \beta, \mathsf{loop}\langle n\rangle\ d) \xrightarrow{\tau_{loop}} \Pi(L, \beta, d\ \mathsf{seq}\ \mathsf{loop}\langle n-1\rangle\ d)\ \mathbf{if}\ ll.(\mathsf{loop}\langle n\rangle\ d) \cap L \neq \emptyset \quad (8.20)$$

Also here we have the condition that $ll.(\mathsf{loop}\langle n\rangle\ d) \cap L = ll.d \cap L \neq \emptyset$. Since we have that $\infty - 1 = \infty$, $\mathsf{loop}\langle\infty\rangle\ d$ specifies an infinite loop. Further we assert that $\mathsf{loop}\langle 0\rangle\ d$ is equal to skip, i.e., $\mathsf{loop}\langle 0\rangle\ d \stackrel{\mathsf{def}}{=} \mathsf{skip}$, so we do not need a special rule for this situation.

8.4 Fairness

With respect to diagrams that contain infinite loop, we must assume weak fairness between diagram parts for the operational semantics to be complete. This means that an arbitrary diagram part may not be enabled infinitely many consecutive execution steps without being executed. With this assumption we avoid situations where some part of a diagram is both enabled and starved for a inifinitely long time. Below we formalize this notion of weak fairness.

8.4.1 Diagram projection part

We define diagram projection parts to be the parts (or fragments) of a diagram that may be reached by the recursion of the projection system. This means that any argument of seq or par and the left argument of strict are projection parts, while arguments of the high-level operators are not. In addition skip is a projection part of any diagram. For example in the diagram

$$d = d_1\ \mathsf{seq}\ ((d_2\ \mathsf{alt}\ d_3)\ \mathsf{par}\ d_4)$$

we have that skip, d_1, $d_2\ \mathsf{alt}\ d_3$ and d_4 are projection parts, while d_2 and d_3 are not.

The reason why the arguments of the high-level operators are not projection parts is that events inside an argument of a high-level operator in a given state will not be reached by the projection system and therefore will not be executed. Hence, for characterizing what is executable in a state, the events inside the arguments of high-level operators are irrelevant since it is the enclosing operator itself that will be executed. Because skip is the identity element of seq, par and strict, skip will by definition be a projection part of any diagram. We nevertheless formalize this explicitly in the definition.

We define a relation $\_ \lhd \_ \in \mathcal{D} \times \mathcal{D} \to \mathbb{B}$ where $d \lhd d'$ should be read as "d is a

projection part of d'." We let $\_ \lhd_1 \_ \in \mathcal{D} \times \mathcal{D} \to \mathbb{B}$ be defined as

$$\begin{aligned} d \lhd_1 d' \stackrel{\mathsf{def}}{=} d = \mathsf{skip} \vee \\ \exists d'' \in \mathcal{D} : d' = d \ \mathsf{seq}\ d'' \vee \\ d' = d'' \ \mathsf{seq}\ d \vee \\ d' = d \ \mathsf{par}\ d'' \vee \\ d' = d'' \ \mathsf{par}\ d \vee \\ d' = d \ \mathsf{strict}\ d'' \end{aligned} \tag{8.21}$$

and define $\lhd$ to be the reflexive and transitive closure of $\lhd_1$.

8.4.2 Enabled and executed projection parts

A diagram projection part d is executed in a execution step $[\beta, d'] \xrightarrow{x} [\beta', d'']$, $x \in \mathcal{E} \cup \mathcal{T}$, if d is changed in the execution of the step (i.e. what is executed is d itself or a part of d). We formalize this by means of a relation $\mathit{executed} \in \mathcal{D} \times \mathcal{EX} \times (\mathcal{E} \cup \mathcal{T}) \times \mathcal{EX} \to \mathbb{B}$:

$$\begin{aligned} &\mathit{executed}(d, [\beta, d'], x, [\beta', d'']) \stackrel{\mathsf{def}}{=} \\ &\quad d \lhd d' \wedge \\ &\quad [\beta, d'] \xrightarrow{x} [\beta', d''] \wedge \\ &\quad \exists d''' \in \mathcal{D} : (\Pi(ll.d, \beta, d) \xrightarrow{x} \Pi(ll.d, \beta, d''') \wedge d''' \lhd d'') \end{aligned} \tag{8.22}$$

Obviously a diagram projection part d is enabled in $[\beta, d']$ if there exists an execution step starting in $[\beta, d']$ in which d may be executed. This is formalized by the relation $\mathit{enabled} \in \mathcal{D} \times \mathcal{EX} \to \mathbb{B}$:

$$\mathit{enabled}(d, [\beta, d']) \stackrel{\mathsf{def}}{=} \exists x, \beta', d'' : \mathit{executed}(d, [\beta, d'], x, [\beta', d'']) \tag{8.23}$$

8.4.3 Weak fairness

We are now able to formalize the notion of weak fairness described above. An *execution* is a sequence of execution steps:

$$[\beta_1, d_1] \xrightarrow{x_1} [\beta_2, d_2] \xrightarrow{x_2} [\beta_3, d_3] \xrightarrow{x_3} \cdots$$

We define

$$\begin{aligned} \Xi \stackrel{\mathsf{def}}{=} \{[\beta_1, d_1] \xrightarrow{x_1} [\beta_2, d_2] \xrightarrow{x_2} [\beta_3, d_3] \xrightarrow{x_3} \cdots \in (\mathcal{EX} \times (\mathcal{E} \cup \mathcal{T}))^\infty \mid \\ \forall i \in \mathbb{N} : \mathit{executed}(d_i, [\beta_i, d_i], x_i, [\beta_{i+1}, d_{i+1}])\} \end{aligned}$$

to be the set of all infinite executions. We say that an execution is weakly fair if no diagram projection part that eventually becomes enabled, stays enabled forever, without being executed infinitely often.[3] This is formalized in the relation $\mathit{wfe} \in \Xi \to \mathbb{B}$:

$$\begin{aligned} &\mathit{wfe}([\beta_1, d_1] \xrightarrow{x_1} [\beta_2, d_2] \xrightarrow{x_2} [\beta_3, d_3] \xrightarrow{x_3} \cdots) \stackrel{\mathsf{def}}{=} \\ &\quad \forall d \in \mathcal{D}, i \in \mathbb{N} : (\forall j \in \mathbb{N} \cup \{0\} : \mathit{enabled}(d, [\beta_{i+j}, d_{i+j}]) \Rightarrow \\ &\qquad \exists k \in \mathbb{N} \cup \{0\} : \mathit{executed}(d, [\beta_{i+k}, d_{i+k}], x_{i+k}, [\beta_{i+k+1}, d_{i+k+1}])) \end{aligned} \tag{8.24}$$

[3] An intuitive discussion of weak fairness can be found in [100].

This definition is based on the assumption that all diagram fragments are unique, something that follows from the assumption that every event in a diagram is unique.

An equivalent expression for *wfe*, that perhaps is closer to the above formulation of a weakly fair execution, is the following:

$$\begin{aligned} &wfe([\beta_1, d_1] \xrightarrow{x_1} [\beta_2, d_2] \xrightarrow{x_2} [\beta_3, d_3] \xrightarrow{x_3} \cdots) = \\ &\quad \neg\exists d \in \mathcal{D}, i \in \mathbb{N} : (\forall j \in \mathbb{N} \cup \{0\} : enabled(d, [\beta_{i+j}, d_{i+j}]) \wedge \\ &\qquad \neg\exists k \in \mathbb{N} \cup \{0\} : executed(d, [\beta_{i+k}, d_{i+k}], x_{i+k}, [\beta_{i+k+1}, d_{i+k+1}])) \end{aligned} \tag{8.25}$$

To ensure weak fairness of the operational semantics we place the following condition on executions:

$$\forall \sigma \in \Xi : wfe(\sigma) \tag{8.26}$$

By using the definition of weakly fair executions we may express that a trace is weakly fair. Let $tr \in \Xi \to (\mathcal{E} \cup \mathcal{T})^\infty$ be a function that picks out all the events of an execution and returns a trace:

$$tr([\beta_1, d_1] \xrightarrow{x_1} [\beta_2, d_2] \xrightarrow{x_2} [\beta_3, d_3] \xrightarrow{x_3} \cdots) \stackrel{\mathsf{def}}{=} \langle x_1, x_2, x_3, \ldots \rangle$$

The relation $wft \in (\mathcal{E} \cup \mathcal{T}) \times \mathcal{D} \to \mathbb{B}$ formalizes that a trace is weakly fair with respect to a diagram:

$$wft(t, d) \stackrel{\mathsf{def}}{=} \exists \sigma \in \Xi : \pi_2(head(\sigma)) = d \wedge tr(\sigma) = t \wedge wfe(\sigma) \tag{8.27}$$

The function *head* returns the first element of σ, which is an execution state, and π_2 returns the second element of an execution state, which is a diagram. The composition $\pi_2(head(\sigma))$ then gives the first diagram in execution σ. Hence, $wft(t, d)$ expresses that the trace t represents a weakly fair execution starting with the diagram d.

8.5 Soundness and completeness

The operational semantics is sound and complete with respect to the denotational semantics presented in section 5.3. Informally, the *soundness* property means that if the operational semantics produces a trace from a given diagram, this trace should be included in the denotational semantics of that diagram. By *completeness* we mean that all traces in the denotational semantics of a diagram should be producible applying the operational semantics on that diagram. In addition it can be shown that execution of a diagram without initite loop always will terminate.

Let $\mathcal{O}$ be the set of all interaction obligations. $[\![d]\!] \in \mathbb{P}(\mathcal{O})$ is the denotation of d. We write $t \in [\![d]\!]$ for $t \in \bigcup_{(p,n) \in [\![d]\!]} (p \cup n)$. $E \circledS t$ denotes the trace t with all events not in E filtered away. $env^!_{\mathcal{M}}.d$ is the set of messages m such that the receive event but not the transmit event of m is present in d.

Theorem (Termination) *Given a diagram $d \in \mathcal{D}$ without infinite loops. Then execution of $[env^!_{\mathcal{M}}.d, d]$ will terminate.*

The proof is found as proof of theorem 3 on page 297 in appendix A.3.

Theorem (Soundness) *Given a diagram $d \in \mathcal{D}$. For all $t \in (\mathcal{E} \cup \mathcal{T})^*$, if there exists $\beta \in \mathcal{B}$ such that $[env^!_{\mathcal{M}}.d, d] \xrightarrow{t} [\beta, \mathsf{skip}]$ then $\mathcal{E} \circledS t \in [\![d]\!]$.*

The soundness theorem is a combination of theorem 1 on page 279 in appendix A.2, that proves soundness of diagram with simple operators, theorem 4 on page 279 in appendix A.3, that proves soundness of diagrams with high-level operators, and theorem 6 on page 311 in appendix A.4, which proves soundess of diagrams with inifinite loops.

Theorem (Completeness) *Given a diagram $d \in \mathcal{D}$. For all $t \in \mathcal{E}^*$, if $t \in [\![d]\!]$ then there exist trace $t' \in (\mathcal{E} \cup \mathcal{T})^*$ and $\beta \in \mathcal{B}$ such that $[env^!_{\mathcal{M}}.d, d] \xrightarrow{t'} [\beta, \mathsf{skip}]$* and $\mathcal{E} \circledS t' = t$.

This theorem a combination of theorem 2 on page 293 in appendix A.2, which proves completeness of diagrams with simple operators, theorem 5 on page 306 in appendix A.3, that proves completeness of diagrams with high-level operators and theorem 7 on page 317 in appendix A.4, where completeness of diagrams with infinite loops is proved.

Chapter 9

Meta-strategies

There are several strategies we may choose when executing a sequence diagram and generating the histories of its possible executions. Examples of this may be generating one or a specific number of random traces, all traces, all prefixes of a certain length, etc. We wish to have the possibility of varying the execution strategy without changing the operational semantics of the sequence diagrams. The way to do this is to define different meta-strategies for executing the diagrams with the operational semantics.

In this chapter we formalize meta-levels that enclose the execution system. At these meta-levels we can define meta-strategies that guide the execution and can be used for handling the challenges related to negative behavior, and potential and mandatory choice. Specifically, two such meta-levels, or meta-systems are defined. In section 9.1 we introduce a meta-system with *mode*, i.e. a way of assigning properties to executions, that are later used in the test generation algorithm of chapter 13. In section 9.2 we present a meta-system for generating all traces of a diagram and give refinement verification as an application of this system.

9.1 The one trace with mode strategy

In this section we show how we can produce a trace while capturing the high-level properties of the trace, e.g. the trace representing negative behavior.

In order to capture these properties we define a meta-system over

$$\langle \_, \_, \_, \_ \rangle \in \mathcal{H} \times \mathcal{EX} \times \mathbb{P}(\mathcal{L}) \times \mathcal{MO}$$

where $\mathcal{MO}$ is a set of modes. The place of this tuple is a "container" for a trace, the second place holds a state of the execution system, the third place holds a set of lifelines that can be used for hiding internal communication in case we want a black box view of the execution and the forth place represents the mode of the execution.

For this system we define the following execution rules:

$$\begin{aligned} &\langle t, [\beta, d], L, mo \rangle \longrightarrow \langle t^\frown\langle e \rangle, [\beta', d'], L, mo \rangle \\ &\quad \textbf{if } [\beta, d] \xrightarrow{e} [\beta', d'] \wedge e \in \mathcal{E} \wedge (l.e \in L \vee l^{-1}.e \in L) \end{aligned} \tag{9.1}$$

$$\begin{aligned} &\langle t, [\beta, d], L, mo \rangle \longrightarrow \langle t, [\beta', d'], L, mo \rangle \\ &\quad \textbf{if } [\beta, d] \xrightarrow{e} [\beta', d'] \wedge e \in \mathcal{E} \wedge l.e \notin L \wedge l^{-1}.e \notin L \end{aligned} \tag{9.2}$$

The idea of these rules is that we collect the trace from the execution, but only events with messages related with the lifelines and gates in L. By applying $gates.d$ as the set of

lifelines we get a trace where all events represent communication with the environment of the diagram, i.e. a black box view. If, instead, $ll.d$ is used as the set of lifelines, we get a glass box view.

The silent events are not visible in the trace collected, but may change the mode of the execution by means of a function $mode \in \mathcal{T} \times \mathcal{MO} \to \mathcal{MO}$:

$$\langle t, [\beta, d], L, mo \rangle \longrightarrow \langle t, [\beta', d'], L, mode(\tau, mo) \rangle \textbf{ if } [\beta, d] \xrightarrow{\tau} [\beta', d'] \wedge \tau \in \mathcal{T} \quad (9.3)$$

So far the set $\mathcal{MO}$ of modes and the *mode* function has been undefined, and obviously there are several way they may be defined. Below we give the most obvious definition as an example. When this meta-system is applied in the test generation algotithm in chapter 13, a second definition of the mode set and the mode function is used (see section 13.2.1).

Let the set of modes be defined as:

$$\mathcal{MO} \stackrel{\text{def}}{=} \{\mathsf{postive}, \mathsf{negative}\}$$

Further, let the function *mode* be defined as:

$$mode(\tau, mo) \stackrel{\text{def}}{=} \begin{cases} \mathsf{postitive} & \textbf{if } mo = \mathsf{positive} \wedge \tau \neq \tau_{refuse} \\ \mathsf{negative} & \textbf{if } mo = \mathsf{negative} \vee \tau = \tau_{refuse} \end{cases}$$

If we use these definitions, and we have postive as the initial mode an execution, the mode will represent the property of the produced trace of being positive or negative behavior.

9.2 The all traces strategy and a method for refinement verification

With this strategy we want to generate all possible traces of a diagram d and place them in the correct semantic structure of STAIRS,[1] i.e. as a set of interaction obligations $\{(p_1, n_1), (p_2, n_2), \ldots, (p_m, n_m)\}$ where each interaction obligation (p_i, n_i), p_i is a set of positive traces and n_i is a set of negative traces.

We make use of a meta-system over

$$\{\!|\_, \_, \_|\!\} \in \mathcal{H} \times \mathcal{EX} \times \mathbb{P}(\mathcal{L})$$

As with the meta-system of section 9.1, the places of the triple is for collecting a trace, holding a state of the execution system and specifying a set of lifelines used for filtering the produced trace. Each element $\{\!|t, [\beta, d], L|\!\} \in \mathcal{H} \times \mathcal{EX} \times \mathbb{P}(\mathcal{L})$ we refer to as a meta-system state.

Instead of sets of traces we use "interaction obligations" of sets of positive and negative executions, i.e. meta-system states. Initially we have a set consisting of a single interaction obligation with the initial state of d as the only positive element and no negative elements

$$\{(\{\!|\langle\rangle, [initC.d, d], L|\!\}\}, \emptyset)\} \quad (9.4)$$

[1] We omit assert because this would require a way of producing all unspecified traces. Further, we only concider finite loop and generalized loop with finite index set, so that the resulting semantics is a finite structure.

where $L \subseteq \mathcal{L}$ is the specified set of lifelines. From this we generate the full semantic structure by defining rules that for each execution state deduce the next steps to be made, and in executing these steps rewrite the whole structure.

A feature of the strategy is that the executions are split every time an alt is executed and that the interaction obligations are split every time an xalt is executed. Therefore, the order in which the high level operators is resolved is significant; in some cases, resolving an alt before an xalt will lead to interaction obligations not characterized by the denotational semantics because this will result in first branching the execution and then splitting the interaction obligation for each of these brances.[2]

In this strategy we therefore apply a normal form of sequence diagrams where loops are eliminated and the high-level operators xalt, alt and refuse are pushed to the top of the diagram (remember we only have finite loops, and no assert operators). In the normal form all xalts are at the top of the diagram (i.e. as the most significant operators), followed by the altss which again are followed by the refuse operators. Details of the normal form is found in appendix B. For the remainder of this section we assume diagrams to be in this normal form.

Because of this normal form we can also assume the rules of the strategy to be applied in a specific order; first the rule for xalts, then the rule for alts, then the rules for refuses, and finally the rules for execution of normal events.

The rule for resolving xalts is the first to be applied. The rule involves splitting an interaction obligation:

$$O \cup \{(\{\{\!|t, [\beta, d], L|\!\}\}, \emptyset)\} \longrightarrow O \cup \{(\{\{\!|t, [\beta', d'], L|\!\}\}, \emptyset) \mid [\beta, d] \xrightarrow{\tau_{xalt}} [\beta', d']\} \tag{9.5}$$

After resolving all xalts, there will be a set of interaction obligations with one execution in each interaction obligation.

If we want all traces, we need to make a branch in the execution for every alt. Hence the rule for resolving alts splits executions inside the interaction obligations:

$$\begin{aligned} &O \cup \{(P \cup \{\{\!|t, [\beta, d], L|\!\}\}, \emptyset)\} \longrightarrow \\ &\quad O \cup \{(P \cup \{\{\!|t, [\beta', d'], L|\!\} \mid [\beta, d] \xrightarrow{\tau_{alt}} [\beta', d']\}, \emptyset)\} \\ &\quad \textbf{if } [\beta, d] \overset{\tau_{xalt}}{\not\rightarrow} \end{aligned} \tag{9.6}$$

The rules for resolving a refuse are more complicated since they concern an interaction obligation and not only one of the sets of an interaction obligation. Let P and N be sets. The rule for resolving a refuse in a positive execution is then:

$$\begin{aligned} &O \cup \{(P \cup \{\{\!|t, [\beta, d], L|\!\}\}, N)\} \longrightarrow O \cup \{(P, N \cup \{\{\!|t, [\beta', d'], L|\!\}\})\} \\ &\quad \textbf{if } [\beta, d] \xrightarrow{\tau_{refuse}} [\beta', d'] \wedge [\beta, d] \overset{\tau_{xalt}}{\not\rightarrow} \wedge [\beta, d] \overset{\tau_{alt}}{\not\rightarrow} \end{aligned} \tag{9.7}$$

In an already negative execution, resolving a refuse makes no difference:

$$\begin{aligned} &O \cup \{(P, N \cup \{\{\!|t, [\beta, d], L|\!\}\})\} \longrightarrow O \cup \{(P, N \cup \{\{\!|t, [\beta', d'], L|\!\}\})\} \\ &\quad \textbf{if } [\beta, d] \xrightarrow{\tau_{refuse}} [\beta', d'] \wedge [\beta, d] \overset{\tau_{xalt}}{\not\rightarrow} \wedge [\beta, d] \overset{\tau_{alt}}{\not\rightarrow} \end{aligned} \tag{9.8}$$

The rules for executing events are first applied after all high-level operators are resoleved by the above rules. There are two rules; one for positive executions and one for negative executions. Every time there is a possibility of more than one event

[2] An elaboration of this is found in appendix B.2.

occurring first, the execution must branch. Hence, the rules for executing events asserts that for a given state, the generation must branch for every enabled event:

$$\begin{aligned} O \cup \{(P \cup \{\!\{t, [\beta, d], L\}\!\}, N)\} \longrightarrow \\ O \cup \{(P \cup \{\!\{t^\frown\langle e\rangle, [\beta', d'], L\}\!\} \mid \\ [\beta, d] \xrightarrow{e} [\beta', d'] \wedge e \in \mathcal{E} \wedge (l.e \in L \vee l^{-1}.e \in L)\}, N)\} \\ \cup \{(P \cup \{\!\{t, [\beta', d'], L\}\!\} \mid \\ [\beta, d] \xrightarrow{e} [\beta', d'] \wedge e \in \mathcal{E} \wedge l.e \notin L \wedge l^{-1}.e \notin L\}, N)\} \\ \textbf{if } [\beta, d] \overset{\tau_{xalt}}{\not\rightarrow} \wedge [\beta, d] \overset{\tau_{alt}}{\not\rightarrow} \wedge [\beta, d] \overset{\tau_{refuse}}{\not\rightarrow} \end{aligned} \tag{9.9}$$

$$\begin{aligned} O \cup \{(P, N \cup \{\!\{t, [\beta, d], L\}\!\})\} \longrightarrow \\ O \cup \{(P, N \cup \{\!\{t^\frown\langle e\rangle, [\beta', d'], L\}\!\} \mid \\ [\beta, d] \xrightarrow{e} [\beta', d'] \wedge e \in \mathcal{E} \wedge (l.e \in L \vee l^{-1}.e \in L)\})\} \\ \cup \{(P, N \cup \{\!\{t, [\beta', d'], L\}\!\} \mid \\ [\beta, d] \xrightarrow{e} [\beta', d'] \wedge e \in \mathcal{E} \wedge l.e \notin L \wedge l^{-1}.e \notin L\})\} \\ \textbf{if } [\beta, d] \overset{\tau_{xalt}}{\not\rightarrow} \wedge [\beta, d] \overset{\tau_{alt}}{\not\rightarrow} \wedge [\beta, d] \overset{\tau_{refuse}}{\not\rightarrow} \end{aligned} \tag{9.10}$$

Note also that these rules only append the executed events to the trace if the events are related to the set of lifelines L.

In summary, these rules ensure that the structure first is split into interaction obligations by resolving xalts. After that executions are split by resolving alts, and then moved to the negative executions if there are refuses to be resolved. When all high-level operators have been removed, we have a structre consisting of simple diagrams which then are executed until we have all the traces of the original diagram.

Refinement verification The "all traces" meta-strategy defined above can be applied to devise a simple refinement verification method. The strategy produces sets of interaction obligations of meta-system states, but by adding the following rules to the system, we get a strategy that gives us sets of interaction obligations of traces:

$$\begin{aligned} O \cup \{(P \cup \{\!\{t, [\beta, d], L\}\!\}, N)\} \longrightarrow O \cup \{(P \cup \{t\}, N)\} \\ \textbf{if } \forall x \in \mathcal{E} \cup \mathcal{T} : [\beta, d] \overset{x}{\not\rightarrow} \end{aligned} \tag{9.11}$$

$$\begin{aligned} O \cup \{(P, N \cup \{\!\{t, [\beta, d], L\}\!\})\} \longrightarrow O \cup \{(P, N \cup \{t\})\} \\ \textbf{if } \forall x \in \mathcal{E} \cup \mathcal{T} : [\beta, d] \overset{x}{\not\rightarrow} \end{aligned} \tag{9.12}$$

The refinement verification is then conducted as follows. Let d_1 and d_2 be diagrams in the normal form, and let O_1 and O_2 be sets of interaction obligations obtained by:

$$\{(\{\!\{\langle\rangle, [initC.d_1, d_1], L\}\!\}, \emptyset)\} \longrightarrow^! O_1$$

and

$$\{(\{\!\{\langle\rangle, [initC.d_2, d_2], L\}\!\}, \emptyset)\} \longrightarrow^! O_2$$

where $\longrightarrow^!$ means applying the rules that define the relation $\longrightarrow$ until no more applications are possible. L is the set of lifelines specified to be external in this verification.

The different types of refinement, as defined in section 5.4.2, can then be verified by applying the following equivalences:

$$\begin{aligned} d_1 \rightsquigarrow_g d_2 &\Leftrightarrow \forall o_1 \in O_1 : \exists o_2 \in O_2 : o_1 \rightsquigarrow_r o_2 \\ d_1 \rightsquigarrow_{rg} d_2 &\Leftrightarrow \forall o_1 \in O_1 : \exists o_2 \in O_2 : o_1 \rightsquigarrow_{rr} o_2 \\ d_1 \rightsquigarrow_l d_2 &\Leftrightarrow d_1 \rightsquigarrow_g d_2 \wedge \forall o_2 \in O_2 : \exists o_1 \in O_1 : o_1 \rightsquigarrow_r o_2 \\ d_1 \rightsquigarrow_{rl} d_2 &\Leftrightarrow d_1 \rightsquigarrow_{rg} d_2 \wedge \forall o_2 \in O_2 : \exists o_1 \in O_1 : o_1 \rightsquigarrow_{rr} o_2 \end{aligned}$$

The refinement relations $o_1 \rightsquigarrow_r o_2$ and $o_1 \rightsquigarrow_{rr} o_2$ used here are the same as defined in section 5.4.1.

Chapter 10

Implementation of operational semantics

In chapter 8 we presented an operational semantics for UML 2.x sequence diagrams. In this chapter we present an implementation of the operational semantics in the rewrite language Maude. Further we give some remarks on conventions for the use of the implementation.

The main purpose of this chapter is that of documenting the implementation. We do not implement infinite behavior, but except from that, the full operational semantics is implemented. Further the implementation is made with the property of being extensible and configurable. In this sense, what is presented in this chapter is not merely an implementation of the operational semantics, but also an implementation of the underlying idea of a general, modifiable, extendible and configurable semantics that provides a firm foundation for analysis of sequence diagrams and a tool box for implementing sound methods of analysis.

We do not repeat definitions from other chapters, but it should still be possible to read and understand it as a self-contained text. We do not assume familiarity with Maude beyond what is presented in chapter 6. All the same, familiarity with chapter 8 and with Maude should provide a great advantage for the reader.

In the presentation of the Maude implementation we will stay close to the module structure of the implementation. We will, however, sometimes drop the definitions of auxiliary sorts and functions. In section 10.1 the implementation of the syntax of sequence diagrams is presented. The implementation of the operational semantics is presented in section 10.2. In section 10.3 we present conventions for usage of the implementation and provide an example. The full code of the implementation is found in appendix C.

10.1 Syntax

In this section we present the syntax of sequence diagrams, as defined in section 5.2, implemented in Maude.

10.1.1 Lifelines and gates

Lifelines and gates are defined by the functional module `LIFELINE`:

```
fmod LIFELINE is
  sorts LLName Lifeline Gate .
  subsort Gate < Lifeline .

  op l : LLName -> Lifeline [ctor] .
  op g : LLName -> Gate [ctor] .
endfm
```

The sort `LLName` defines the set of names of lifelines. It does, however, not have any constructors. The idea behind this is that when a specific diagram is defined, names of lifelines and gates should be provided. This is further discussed in section 10.3.

The module further defines two sorts `Lifeline` and `Gate`, and `Gate` is defined as subsort of `Lifeline`. This means that the set of gates is included in the set of lifelines and that gates are treated as lifelines. The constructors are operators `l` and `g`, respectively, which both take a lifeline name as argument.

In addition, a functional module `LLSET` defines the sort `LLSet` as sets of lifelines. We not do provide the module here, but refer to appendix C.1.

10.1.2 Messages and events

Messages are defined by the functional module `MESSAGE`:

```
fmod MESSAGE is
  including LIFELINE .

  sorts Signal Msg .

  op (_,_,_) : Signal Lifeline Lifeline -> Msg [ctor] .

  ops tr re : Msg -> Lifeline .

  var S : Signal .
  vars T R : Lifeline .

  eq tr(S, T, R) = T .
  eq re(S, T, R) = R .
endfm
```

The module includes the module `LIFELINE` and defines two sorts `Signal` and `Msg`. As with lifeline names, the sort `Signal` has no constructors; the signals should be provided when defining a specific diagram.

As in section 5.2.1, messages are defined as tuples of signals, transmitter lifelines and receiver lifelines. On messages we define a transmitter function `tr` and receiver function `re` that return the transmitter lifeline and receiver lifeline of a message, respectively.

Events are defined in the functional module `EVENT`:

```
fmod EVENT is
  including MESSAGE .

  sorts Event EventKind SEvent SEventKind .
  subsort SEvent < Event .
```

```
  ops ! ? : -> EventKind [ctor] .
  op (_,_) : EventKind Msg -> Event [ctor] .
  op tau : SEventKind -> SEvent [ctor] .

  op k : Event -> EventKind .
  op m : Event -> Msg .
  ops tr re l l-1 : Event -> Lifeline .

  var M : Msg .
  var K : EventKind .

  eq k(K,M) = K .
  eq m(K,M) = M .
  eq tr(K,M) = tr(M) .
  eq re(K,M) = re(M) .
  ceq l(K,M) = tr(M) if K = ! .
  ceq l(K,M) = re(M) if K = ? .
  ceq l-1(K,M) = re(M) if K = ! .
  ceq l-1(K,M) = tr(M) if K = ? .
endfm
```

The module defines four sorts: `Event`, representing the set of events, `EventKind` representing the set of kinds (transmit and receive) of events, `SEvent` representing the set of silent events, and `SEventKind`, the set of kinds of the silent events. The sort `EventKind` has the two constants `!` and `?` as its constructors and only members. Events are defined as a tuples of kinds and messages. The sort `SEvent` has the constructor `tau` which takes a silent event kind as parameter. The members of `SEventKind` are not defined in this module, but are defined in operator specific modules as part of the configuration mechanism described later.

On the set of events we define a number of functions: `k` returns the kind of an event, `tr` and `re` overload the functions defined in `MESSAGE` and return the transmitter and receiver of an event, `l` returns the lifeline of an event (i.e. the transmitter if the kind is transmit and receiver if the kind is receive), and `l-1` returns the opposite lifeline of an event (i.e. the receiver of the message if the kind is transmit and the transmitter of the message if the kind is receive).

10.1.3 Diagrams

In this section we present the Maude representation of the syntax of sequence diagrams. The definitions are structured such that a module `SD-SYNTAX` gives the definition of simple diagrams, i.e., diagrams built from events and the operators seq, par and strict. The high-level sequence diagram operators, such as alt and loop are defined in separate modules (allowing configurability).

The syntax of simple diagrams is defined in the functional module `SD-SYNTAX`:

```
fmod SD-SYNTAX is
  including EVENT .
  including LLSET .
```

```
  sort SD .
  subsort Event < SD .

  op skip : -> SD [ctor] .
  op _seq_ : SD SD -> SD [ctor assoc prec 20] .
  op _strict_ : SD SD -> SD [ctor assoc prec 25] .
  op _par_ : SD SD -> SD [ctor assoc comm prec 30] .

  op ll : SD -> LLSet .
  op gates : SD -> LLSet .

  vars D D' : SD .
  var E : Event .

  eq D par skip = D .
  eq skip seq D = D .
  eq D seq skip = D .
  eq D strict skip = D .
  eq skip strict D = D .

  eq ll(skip) = emptyLLSet .
  eq ll(E) = l(E) .
  eq ll(D seq D') = ll(D) ; ll(D') .
  eq ll(D par D') = ll(D) ; ll(D') .
  eq ll(D strict D') = ll(D) ; ll(D') .

  eq gates(skip) = emptyLLSet .
  eq gates(E) = if l-1(E) :: Gate then l-1(E) else emptyLLSet fi .
  eq gates(D seq D') = gates(D) ; gates(D') .
  eq gates(D par D') = gates(D) ; gates(D') .
  eq gates(D strict D') = gates(D) ; gates(D') .
endfm
```

A sequence diagram is a defined as a term of the sort `SD`. `Event` is defined as a subsort of `SD`, so that a single event is also a sequence diagram. `SD` has four constructors: The constant `skip` for representing the empty diagram, and the binary, mixfix operators `seq`, `strict` and `par` for building diagrams. The arguments of the constructor definitions define `seq` and `strict` as associative and `par` as associative and commutative. Further the argument `prec` gives the precedence of the operators. In this case this means that `seq` binds stronger than `strict` and `strict` binds stronger than `par`.

Even though `skip` is the identity element of the constructor operators we do not specify this in the constructor definitions (as in the list example in chapter 6), but define this by means of equations reducing, e.g., `D seq skip` to `D` for a diagram `D`. The reason for this is pragmatic; the way in which equations and rewrite rules are defined is sometimes dependent of how identity is defined. In our case this way makes the nicest equations and rules.

The module defines two functions on sequence diagrams, the function `ll` which returns the set of lifelines of a diagram and the function `gates` which returns the gates of a diagram as a set of lifelines.

Every high-level operator is defined in a separate module `EXTENSION-`⟨*operator*

name⟩. The purpose of this is to allow configurability when defining diagrams by inclusion of the appropriate extension. This is further explained in section 10.3.

10.1.3.1 Choice

The module `EXTENSION-ALT` and `EXTENSION-XALT` define the alt operator and xalt operator respectively:

```
fmod EXTENSION-ALT is
  including SD-SYNTAX .

  op _alt_ : SD SD -> SD [ctor assoc comm prec 35] .
  op opt_ : SD -> SD [ctor prec 15] .
  op alt : -> SEventKind [ctor] .

  vars D D' : SD .

  eq skip alt skip = skip .
  eq opt D = D alt skip .

  eq ll(D alt D') = ll(D) ; ll(D') .
  eq gates(D alt D') = gates(D) ; gates(D') .
endfm

fmod EXTENSION-XALT is
  including SD-SYNTAX .

  op _xalt_ : SD SD -> SD [ctor assoc comm prec 40] .
  op xalt : -> SEventKind [ctor] .

  vars D D' : SD .

  eq skip xalt skip = skip .

  eq ll(D xalt D') = ll(D) ; ll(D') .
  eq gates(D xalt D') = gates(D) ; gates(D') .
endfm
```

There are two things to notice in these modules: 1) We define `alt` and `xalt` as constants of the sort `SEventKind`, which means we have `tau(alt)` and `tau(xalt)` as silent events. 2) The modules define the functions `ll` and `gates` also for diagrams containing the operators `alt` and `xalt`.

Also note that the module `EXTENSION-ALT` in addition defines the unary operator `opt` as an operator derived from `alt` by the equation `eq opt D = D alt skip`, and that `skip alt skip` and `skip xalt skip` are reduced to `skip`.

10.1.3.2 Negative behavior

The module defining the operators for negative behavior is similar to the other extensions, and with `neg` defined as an operator derived from `refuse`, `alt` and `skip`:

```
fmod EXTENSION-REFUSE is
  including SD-SYNTAX .
  including EXTENSION-ALT .

  op refuse_ : SD -> SD [ctor prec 1] .
  op refuse : -> SEventKind [ctor] .
  op neg_ : SD -> SD [ctor prec 10] .

  var D : SD .

  eq neg D = refuse D alt skip .

  eq ll(refuse D) = ll(D) .
  eq gates(refuse D) = gates(D) .
endfm
```

10.1.3.3 Universal behavior

The module defining the operator assert is similar to the other extensions:

```
fmod EXTENSION-ASSERT is
  including SD-SYNTAX .

  op assert_ : SD -> SD [ctor prec 5] .
  op assert : -> SEventKind [ctor] .

  var D : SD .

  eq ll(assert D)  = ll(D) .
  eq gates(assert D) = gates(D) .
endfm
```

10.1.3.4 Iteration

In order to define loop in the same way as in section 5.3.9 we need to define a sort for the set of natural numbers. This is provided by the module `NAT-SET` (consult appendix C.1 for details). Note that since we only implement finite behavior, we are only dealing with finite sets of natural numbers. The module provides a function `toSet` that converts an interval represented by the lower and upper bounds of the interval such that, e.g., $toSet(2,5) = \{2,3,4,5\}$.

The loop operator is defined in the module `EXTENSION-LOOP`:

```
fmod EXTENSION-LOOP is
  including SD-SYNTAX .
  including NAT-SET .
  including EXTENSION-ALT .

  op loop{_}_ : NatSet SD -> SD [ctor prec 7] .
  op loop<_>_ : Nat SD -> SD [ctor prec 7] .
  op loop(_,_)_ : Nat Nat SD -> SD [ctor prec 7] .
  op loop : -> SEventKind [ctor] .
```

```
  var D : SD .
  vars N N' : Nat .
  var I : NatSet .

  eq loop< 0 > D = skip .
  eq loop(N,N') D = loop{toSet(N,N')} D .
  eq loop{emptyNatSet} D = skip .
  eq loop{N} D = loop< N > D .
  ceq loop{N , I} D = loop< N > D alt loop{I} D if I =/= emptyNatSet .

  eq ll(loop< N > D) = ll(D) .
  eq gates(loop< N > D) = gates(D) .
endfm
```

In this implementation we only define the finite loop. Note that we define three loops. `loop{I} D` is the "standard STAIRS loop" where the set `I` represents a choice for how many time the loop should be iterated, and `loop< N > D` is a loop to be iterated `N` times. `loop(N,N') D` is the loop as it is defined in the UML standard and is treated as a short hand notation for `loop{I} D` where `I` is the interval of natural numbers from `N` to `N'`. As in section 5.3.9, `loop{I}` is defined as a derived operator of `loop< N >` and `alt`, but without defining the generalized alt explicitly.

10.1.3.5 Full syntax

For use of the full syntax, a module `SD-FULL-SYNTAX` may be applied:

```
fmod SD-FULL-SYNTAX is
  including SD-SYNTAX .
  including EXTENSION-ALT .
  including EXTENSION-XALT .
  including EXTENSION-REFUSE .
  including EXTENSION-ASSERT .
  including EXTENSION-LOOP .
endfm
```

10.2 Operational semantics

In the implementation of the operational semantics, rules for both the execution system and the projection system are implemented as rewrite rules in Maude. The rules of our operational semantics are based on placing events on the arrows of the rules, like labels on transitions of a labeled transition system. The rewrite rules of Maude do not have an equivalent of this. Inspired by [185] we solve this by making the event part of the state after the transition/rewrite. The event is in other words placed after the arrow instead of above the arrow, and a transition of the form $[\beta, d] \xrightarrow{e} [\beta', d']$ is in the Maude implementation represented by a rewrite `[B,D] => {E}[B',D']`.

The signature of the projection system is defined by an operator `PI` of sort `Pi` in module `PI`:

```
fmod PI is
```

```
  including SD-SYNTAX .
  including COMMUNICATION .

  sort Pi .

  op PI : LLSet Comm SD -> Pi [ctor] .
  op {_}_ : Event Pi -> Pi [ctor] .
endfm
```

Similar, the module `EXEC` defines a sort `Exec` for execution states, but for technical reasons this module does not provide the signature itself. However the function `update` is defined and the signature of a function `init` is defined.

```
fmod EXEC is
  including PI .

  sort Exec .

  op {_}_ : Event Exec -> Exec [ctor] .

  op init : SD -> Exec .
  op update : Comm Event -> Comm .

  var B : Comm .
  var E : Event .

  ceq update(B,E) =
    (if k(E) == ! then
      add(B,m(E))
     else
       rm(B,m(E))
     fi)
    if not E :: SEvent .

  ceq update(B,E) = B if E :: SEvent .
endfm
```

Both these modules define a constructor operator `{_}_` that allows for having the event as part of the state as described above.

The module `PI` imports the module `COMMUNICATION` which provides a type `COMM` for the communication medium as well as the signatures of the functions to manipulate the communication medium:

```
fmod COMMUNICATION is
  including MESSAGE .
  including SD-FULL-SYNTAX .

  sort Comm .

  op emptyComm : -> Comm [ctor] .
  ops add rm : Comm Msg -> Comm .
  op ready : Comm Msg -> Bool .
```

```
  op initC : SD -> Comm .
endfm
```

In this module the type and the functions are undefined; the definitions are provided by a separate module depending on the choice of communication model. Implementations of the four communication models defined in section 8.2 are found in appendix C.2.1.

10.2.1 Execution system

The execution system is defined in a module `Execution` where both the signature of the execution states and the execution rules are given.

The most natural way of defining the execution system would be to have two rewrite rules, one implementing the rule for execution of a normal events, rule (8.3), and one implementing the rule for execution of silent events, rule (8.6). The rules would look like

```
crl [execute] :
  [B, D] => {E}[update(B,E), D']
  if PI(ll(D), B, D) => {E}PI(L, B, D')
     /\ not E :: SEvent .

crl [execute-silent] :
  [B, D] => {E}[B, D']
  if PI(ll(D), B, D) => {E}PI(L, B, D')
     /\ E :: SEvent .
```

where `E :: SEvent` is a test for whether `E` is a silent event or not (and in the latter case a normal event). Because of performance issues, we instead implement this as a single rule (the Maude interpreter in some cases works faster with fewer rules). This is easily obtained by letting the function `update` (defined in the module `EXEC` above) also handle silent events (by doing nothing with the state of the communication medium in case of a silent event). The implementation of the execution system is then defined as follows:

```
mod EXECUTION is
  including EXEC .

  op [_,_] : Comm SD -> Exec [ctor] .

  var B : Comm .
  vars D D' : SD .
  var E : Event .
  var L : LLSet .

  eq init(D) = [initC(D), D] .

  crl [execute] :
    [B, D] => {E}[update(B,E), D']
    if PI(ll(D), B, D) => {E}PI(L, B, D') .
endm
```

10.2.2 Projection system

The implementation of the projection system is a straight forward implementation of every rule defined in the system as a rewrite rule in Maude. The module below, `PI-SIMPLE`, implements the rules of the projection system for simple diagrams, i.e. rule (8.10) for events, rules (8.11) and (8.12) for seq, rules (8.13) and (8.14) for par and rule (8.15) for strict. Because `par`, in the module `SD-SYNTAX`, is defined as associative, it is sufficient with one rule in the implementation.

```
mod PI-SIMPLE is
  including PI .

  vars D1 D1' D2 D2' : SD .
  var E : Event .
  vars L L' : LLSet .
  var B : Comm .

  crl [PI-event] :
    PI(L, B, E) => {E}PI(L, B, skip)
    if l(E) in L /\
       k(E) == ! or ready(B,m(E)) .

  crl [PI-seq-left] :
    PI(L, B, D1 seq D2) => {E}PI(L, B,  D1' seq D2)
    if intersect(ll(D1),L) =/= emptyLLSet /\
       PI(intersect(ll(D1),L), B, D1) => {E}PI(L', B, D1') .

  crl [PI-seq-right] :
    PI(L, B, D1 seq D2) => {E}PI(L, B, D1 seq D2')
    if L \ ll(D1) =/= emptyLLSet /\
       PI(L \ ll(D1), B, D2) => {E}PI(L', B, D2') .

  crl [PI-par] :
     PI(L, B, D1 par D2) => {E}PI(L, B, D1' par D2)
     if PI(intersect(ll(D1),L), B, D1) => {E}PI(L', B, D1') .

  crl [PI-strict] :
     PI(L, B, D1 strict D2) => {E}PI(L, B, D1' strict D2)
     if intersect(ll(D1), L) =/= emptyLLSet /\
        PI(intersect(ll(D1), L), B, D1) => {E}PI(L', B, D1') .
endm
```

For each of the modules `EXTENSION-`⟨*operator name*⟩ there is a module `PI-`⟨*operator name*⟩. Below we show the module `PI-ALT` which implements the rule (8.16) for resolving alts. Also here, only one rule is necessary since the `alt` operator is defined as associative.

```
mod PI-ALT is
  including PI .
  including EXTENSION-ALT .

  vars D1 D2 : SD .
```

```
  var L : LLSet .
  var B : Comm .

  crl [PI-alt] :
    PI(L, B, D1 alt D2) => {tau(alt)}PI(L, B, D1)
    if intersect(ll(D1 alt D2), L) =/= emptyLLSet .
endm
```

The other modules PI-⟨*operator name*⟩ follow exactly the same pattern, and may be found in appendix C.2.2. For use of the full projection system, a model PI-FULL that imports all the PI-modules may be applied:

```
mod PI-FULL is
  including PI-SIMPLE .
  including PI-ALT .
  including PI-XALT .
  including PI-REFUSE .
  including PI-ASSERT .
  including PI-LOOP .
endm
```

10.2.3 Meta-strategies

In order to implement the meta-strategies we first define a module TRACE that defines a type SDTrace for finite traces:

```
fmod TRACE is
  including EVENT .

  sort SDTrace .
  subsort Event < SDTrace .

  op emptyTrace : -> SDTrace [ctor] .
  op _._ : SDTrace SDTrace -> SDTrace [ctor assoc id: emptyTrace] .
endfm
```

10.2.3.1 One trace with mode

For the implementation of the execution strategy of section 9.1 we introduce a module META-EXEC-MODE where we define a type Meta and use this to define the signature of the meta-system. Further we define a type Mode to represent the set of modes $\mathcal{MO}$, and provide the signature of the *mode* function:

```
mod META-EXEC-MODE is
  including EXEC .
  including TRACE .

  sort Meta .
  sort Mode .

  op <_,_,_,_> : SDTrace Exec LLSet Mode -> Meta [ctor] .
```

```
  op mode : SEvent Mode -> Mode .
endm
```

The default modes defined in section 9.1 are implemented in a seperat module `DEFAULT-MODE` where the type `Mode` is given the values `positive` and `negative`, and the function `mode` is defined:

```
mod DEFAULT-MODE is
  including META-EXEC-MODE .

  ops positive negative : -> Mode [ctor] .

  var T : SEvent .
  var MO : Mode .

  ceq mode(T,MO) = positive if MO == positive /\ T =/= tau(refuse) .
  ceq mode(T,MO) = negative if MO == negative or T == tau(refuse) .
endm
```

In the module `EXTERNAL-TRACE-MODE`, the rules (9.1), (9.2) and (9.3) are implemented as rewrite rules. These rules use transitions of the execution system as condition, which means we apply the built in mechanism in the Maude interpreter for (semi) non-deterministic application of the rule:

```
mod EXTERNAL-TRACE-MODE is
  including META-EXEC-MODE .
  including SD-FULL-SYNTAX .

  var T : SDTrace .
  var E : Event .
  vars V V' : Exec .
  var L : LLSet .
  vars M M' : Mode .
  var D : SD .
  var B : Comm .
  var TAU : SEvent .

  crl < T, V, L, M > => < T . E , V', L, M >
     if V => {E}V' /\
        not E :: SEvent /\
        (l(E) in L or l-1(E) in L) .

  crl < T, V, L, M > => < T, V', L, M >
     if V => {E}V' /\
        not E :: SEvent /\
        not l(E) in L /\
        not l-1(E) in L .

  crl < T , V, L, M > => < T, V', L, mode(TAU,M)  >
      if V => {TAU}V' .
endm
```

In addition to this implementation of the meta-strategy, we also provide another implementation of the strategy. In this other implementation, the execution of the system is randomized instead of using the build in mechanism in the Maude interpreter for applications of the rules. We do not give the details here, but refer to appendix C.3.1.

10.2.3.2 All traces

When implementing the strategy for producing all traces, more work is required. Similar to the above strategy we make use of the type `SDTrace`, and provide a module where a type `Meta` is defined and the signature of the meta-system given:

```
mod META-EXEC is
  including EXEC .
  including TRACE .

  sort Meta .

  op {|_,_,_|} : SDTrace Exec LLSet -> Meta [ctor] .
endm
```

In addition we define a module `SEMANTICS` that defines the types `TraceSet`, `InteractionObligation` and `IOSet` which we need for characterizing the semantic structure of STAIRS. We also need the transformation rules for obtaining the normal form of diagram (see section B.1). An implementation of these is provided by the module `TRANSFORMATIONS`.

The execution strategy is implemented by a module `ALL-TRACES`. In this module the type `Meta` is made subsort of `SDTrace`. This way the semantic structure defined in `SEMANTICS` can be used to structure states of the meta-system as well as traces. In the implementation we make use of the meta-level capabilities of Maude (see section 6.3) so that for every state of the execution system we are able to obtain every possible transition. In this way we define a function `allTraces` that from an execution state produces the semantics of the diagram of that execution state. The implementation of this execution strategy is rather lengthy so we do not provide it here, but it can be found in appendix C.3.2.

Refinement verification. The refinement verification mechanism is implemented in a module `REFINEMENT`. This module contains straight forward definintions of the refinement relations as Boolean operators:

```
ops _~>r_ _~>rr_ : InteractionObligation InteractionObligation -> Bool .
ops _~>g_ _~>rg_ _~>l_  _~>rl_ : IOSet IOSet -> Bool .
```

In addition, a mechanism for substitution of lifelines in traces is provided by a module `MAPPING`. This mechanism may be applied if the refinement verification is used for comparing diagrams with different lifeline names. See appendix C.3.2.1 for details.

10.3 Conventions for use

In this section we present some conventions for the use of the implementation of the operational semantics we have presented in this chapter. Further, we give examples of

use of the Maude implementation together with the Maude interpreter. As example we use the diagram d of figure 5.1 on page 30. As we saw in section 5.2.2, the syntactical representation of this diagram is:

$$d = (?, (a, g, i)) \text{ seq } ((!, (b, i, j)) \text{ seq } (?, (b, i, j)) \text{ alt } (!, (c, i, j)) \text{ seq } (?, (c, i, j)))$$

We let a diagram be represented by a module, so the diagram d is represented by a module `D`. This module must import the modules defining the syntax of at least the operators that are used in the diagram.

The types `LLName` (lifeline name) and `Signal` are without any values. In the module `D` we therefore define the lifelines and gates of the diagram as constants of type `LLName` and the signals of the diagram as constants of type `Signal`:

```
ops i j g : -> LLName .
ops a b c : -> Signal .
```

We can now build lifelines and gates from lifeline names, messages from lifelines, gates and signal, events from messages, and diagrams from events. For increased readability we also give names to lifelines, gates, messages and events (`$` in these names is just for avoiding name collisions, and `!` and `?` are used for readability). For example do we define the event "reception of message a", i.e. $(?, (a, g, i))$, by:

```
op $i : -> Lifeline .
eq $i = l(i) .
op $g : -> Gate .
eq $g = g(g) .
op $a : -> Msg .
eq $a = (a,$g,$i) .
op ?a : -> Event .
eq ?a = (?,$a) .
```

In the same fashion we give an explicit name to the diagram. The full definition of diagram d may be:

```
mod D is
  including SD-FULL-SYNTAX .

  ops i j g : -> LLName .
  ops $i $j : -> Lifeline .
  op $g : -> Gate .
  eq $i = l(i) .
  eq $j = l(j) .
  eq $g = g(g) .

  ops a b c : -> Signal .
  ops $a $b $c : -> Msg .
  eq $a = (a,$g,$i) .
  eq $b = (b,$i,$j) .
  eq $c = (c,$i,$j) .

  ops ?a !b ?b !c ?c : -> Event .
  eq ?a = (?,$a) .
```

```
  eq !b = (!,$b) .
  eq ?b = (?,$b) .
  eq !c = (!,$c) .
  eq ?c = (?,$c) .

  op d : -> SD .
  eq d = ?a seq (!b seq ?b alt !c seq ?c) .
endm
```

Use of the Maude interpreter With the above Maude representation `D` of the diagram d, we can use the commands of the Maude interpreter to execute and, in other ways, manipulate the diagram. In the following we give some examples.

By applying the command

```
reduce in D : d .
```

we get the Maude definition of the diagram:

```
result SD: (?,a,g(g),l(i)) seq ((!,b,l(i),l(j)) seq (?,b,l(i),l(j))
                             alt (!,c,l(i),l(j)) seq (?,c,l(i),l(j)))
```

In order to execute the diagram we need to import modules implementing the operational semantics. If we, e.g., want the full projection system and the communication model with no ordering on messages, we add the following lines to the module:

```
including EXECUTION .
including PI-FULL .
including ONE-FIFO-MESSAGE .
```

If we want to apply the meta-system for generating a single trace with the default definition of modes we also include the lines:

```
including EXTERNAL-TRACE-MODE .
including DEFAULT-MODE .
```

We can now execute the diagram using the `rewrite` command in the Maude interpreter:

```
rewrite in D : < emptyTrace,[$a,d],ll(d),positive > .
```

Because the diagram contains a message going from a gate to the lifeline of the diagram (i.e. a), we start the execution with this message (i.e. `$a`) in the communication medium in the initial state. This way the message is available for execution of `?a` and execution may proceed as it should and not stop because the event is disabled by the absence of the message. In order to get a glass box view of the execution we provide all the lifelines of the diagram to the meta-system, using the function `ll(d)`.

The result is a state of the meta-system where further execution is not possible:

```
result Meta: < (?,a,g(g),l(i)) . (!,b,l(i),l(j)) . ?,b,l(i),l(j),
               [emptyComm,skip],l(i) ; l(j),positive >
```

This way we obtain a (semi) non-deterministically generated trace. If we would like to get all possible traces from this diagram we can apply the `search` command:

```
search in D : < emptyTrace,[$a,d],ll(d),positive > =>!
              < T:SDTrace,X:Exec,L:LLSet,M:Mode > .

Solution 1 (state 37)
states: 43  rewrites: 25065 in 7672014078ms cpu (274ms real)
    (0 rewrites/second)
T:SDTrace --> (?,a,g(g),l(i)) . (!,b,l(i),l(j)) . ?,b,l(i),l(j)
X:Exec --> [emptyComm,skip]
L:LLSet --> l(i) ; l(j)
M:Mode --> positive

Solution 2 (state 38)
states: 43  rewrites: 25069 in 7672014078ms cpu (275ms real)
    (0 rewrites/second)
T:SDTrace --> (?,a,g(g),l(i)) . (!,c,l(i),l(j)) . ?,c,l(i),l(j)
X:Exec --> [emptyComm,skip]
L:LLSet --> l(i) ; l(j)
M:Mode --> positive

No more solutions.
states: 43  rewrites: 25169 in 7672014078ms cpu (277ms real)
    (0 rewrites/second)
```

Note, however, that this only gives us the traces (essentially as a flat set) and not the semantic structure of STAIRS. This we can obtain by applying the "all traces" strategy, even though the result in this case is not particulary interesting:

```
reduce in D : allTraces([$a,d], ll(d)) .

rewrites: 1215 in -4295712345ms cpu (551ms real) (~ rewrites/second)

result InteractionObligation:
  (((?,a,g(g),l(i)) . (!,b,l(i),l(j)) . ?,b,l(i),l(j)) +
   (?,a,g(g),l(i)) . (!,c,l(i),l(j)) . ?,c,l(i),l(j)),
  emptyTSet
```

Chapter 11

Related work on operational semantics

In this chapter we present related work, which means other approaches to defining operational semantics to sequence diagrams. This presentation cannot, however, be seen independently of the history of sequence diagrams. The various approaches of defining semantics to sequence diagrams have emerged at different points in this history, and are clearly influenced by the state of the language(s) at the time of their emergence (see introduction of chapter 4).

In addition to the STAIRS semantics there exist some other approaches to defining denotational semantics for UML 2.x sequence diagrams [30, 98, 173, 174]. As explained section 7.2, denotational and operational semantics are complementary and serve different purposes. For this reason we do not go into detail on these approaches in this discussion, nor on existing approaches to denotational semantics of MSC, like [92].

Several approaches to defining operational semantics to UML 2.x sequence diagrams and MSCs exist. We do, however, find that none of these semantic definitions are satisfactory for our purposes.

The approach of Cengarle and Knapp [31] has some similarities to our operational semantics; for every execution step an event is produced and at the same time the syntactical representation of the diagram is reduced by the removal of the event produced. Contrary to ours, however, their semantics treats sequence diagrams as complete specifications (with no inconclusive behavior), something that does not conform to the UML standard (see sections 4.2 and 7.3.1). The rules are defined so that a given diagram produces a set of positive and negative traces that together exhaust the trace universe. The neg operator is replaced by an operator not. This operator is defined so that the sets of positive and negative traces are swapped. This is unfortunate since specifying some behavior as negative means also specifying the complement of this behavior as positive. We claim that this is not what you intuitively expect when specifying negative behavior.

There are also some problems on the technical level. The semantics is based on a notion of composition of basic interactions where basic interactions are defined as partially ordered multisets (pomsets), but it is not clear how the pomsets are obtained; in other words it is not clear what the atoms of the compositions are. If the atoms are taken to be single events the reduction rules defining the seq operator do not preserve the message invariant (causality between transmission and reception of a message).

Cavarra and Klüster-Filipe [29] present an operational semantics of UML 2.x se-

quence diagrams inspired by Live Sequence Charts (LSC) (see below). The semantics is formalized in pseudo-code that works on diagrams represented as locations in the diagram, but no translation from diagrams to this representation is provided. A more serious problem is the treatment of the choice operator alt. The operands of alts have guards and there is nothing to prevent the guards of more operands in an alt to evaluate to **true**. In this case the uppermost operand will be chosen, which means that the alts essentially are treated as nested **if-then-else** statements and may not be used for underspecification. Further, each lifeline is executed separately which means that synchronization at the entry of alt-fragments is necessary to ensure that all lifelines choose the same operand. They also make the same assumption about negative behavior as in LSCs, that if negative (neg) fragment is executed, then execution aborts (see section 7.3.4).

Harel and Maoz [60] use LSC semantics (see below) to define neg and assert. The operators are defined using already existing constructs of LSCs, and hence no changes or additions to the LSC semantics are needed in their approach. Because of this they also inherit the problems connected to LSCs.

In [58], Grosu and Smolka provide a semantics for UML 2.x sequence diagrams based on translating the diagrams to Büchi automata. The approach is based on composing simple sequence diagrams (no high-level operators) in high-level sequence diagrams (interaction overview diagrams), where a simple diagram may be a positive or a negative fragment of the high-level diagram it belongs to. Positive behavior is interpreted as liveness properties and negative behavior as safety properties. Hence, for a high-level diagram two Büchi automata is derived; a liveness automaton characterizing the positive behavior of the diagram and a safety automaton characterizing the negative behavior. Compared to our operational semantics the approach is based on a large amount of transformation. Further the diagrams are composed by strict sequencing rather than weak sequencing, and hence has implicit synchronization of lifelines when entering or leaving a simple diagram.

In 1995 a formal algebraic semantics for MSC-92 was standardized by ITU [117,118]. MSC-92 has a lifeline-centric syntax and its semantics is based on characterizing each lifeline as a sequence (total order) of events. These sequences are composed in parallel and a set of algebraic rules transforms the parallel composition into a structure of (strict) sequential composition and choice. The message invariant is obtained by a special function that removes from the structure all paths that violate the invariant. The main drawbacks of this semantics are that it is not a proper operational semantics since a diagram first has to be transformed into the event structure before runs can be obtained, and that this transformation replaces parallel composition with choice and hence creates an explosion in the size of the representation of the diagram. Further, the lifeline-centric syntax is not suitable for defining nested high-level constructs. In [137] a similar semantics for UML 1.x sequence diagram is given.

MSC-96 got a standardized process algebra semantics in 1998 [84, 119, 120]. This semantics is event-centric and has semantic operators for all the syntactic operators in MSC-96. Further these operators are "generalized" to preserve the message invariant by coding information about messages into the operators in the translation from syntactical diagrams to semantic expressions. Runs are characterized by inference rules over the semantic operators. Compared to our work, the most notable thing about this semantics is that it has no notion of negative behavior, and therefore also makes no distinction between negative behavior and inconclusive behavior (behavior that is

neither positive nor negative). This is no surprise since MSC does not have the neg operator, but it is still a shortcoming with respect to UML sequence diagrams. The only available meta-level is a flat transition graph, and this does not give sufficient strength to extend the semantics with negative behavior. Neither is it possible to distinguish between potential and mandatory choice. Another shortcoming is the lack of an explicit communication medium; the communication model is "hardcoded" in the semantics by the "generalized operators" and does not allow for variation.

Another process algebra semantics of MSC is presented in [104]. This semantics may in some respects be seen as more general than both the MSC-92 and the MSC-96 semantics. A simple "core semantics" for MSCs is defined and this semantics is then inserted into an environment definition. Varying the definition of the environment allows for some of the same semantic variability and extendibility that we aim for in our semantics, e.g., with respect to the communication model. However, the semantics is heavily based on synchronization of lifelines on the entry of referenced diagrams and combined fragments and diverges in this respect from the intended semantics of MSCs and UML sequence diagrams. Further, the same strategy as for the MSC-92 semantics is applied; interleaving is defined by means of choice, and the message invariant is obtained by removing deadlocks. In our opinion, this results in an unnecessary amount of computation, especially in the cases where we do not want to produce all traces but rather a selection of the traces that a diagram defines.

In [99] a semantics for MSC-92 is presented which is based on translating MSCs to finite automata with global states. This approach has no high-level operators, but conditions may be used for specifying choices between or iterations of diagrams. However, the insistence on translating the diagrams to *finite* automata makes it impossible to represent all reachable states because the combination of weak sequencing and loops may produce infinitely many states.

Realizability of MSCs is the focus of both [5, 6] and [182]. They define synthesis of MSC to concurrent automata and parallel composition of labeled transition systems (LTS), respectively. (Each lifeline is represented as an automaton or LTS, which are then composed in parallel.) Further they define high-level MSCs as graphs where the nodes are basic MSCs. In addition, [182] defines both syntax and semantics for negative behavior. In both approaches the translation of high-level MSCs to concurrent automata/LTSs removes the semi-global nature of choices in a specification, and the high-level MSC graphs are non-hierarchical disallowing nesting of high-level operators. Further, in [182] communication is synchronous.

Various attempts at defining Petri net semantics for MSCs have been made [57, 59, 73, 163]. In [57, 163] only basic MSCs are considered, and hence are of limited interest. In [59], high-level MSCs are defined as graphs where each node is a basic MSC. As with the above mentioned semantics it is then possible to express choices and loops, but the approach does not allow for nesting of high-level operators. In [73] a Petri net translation of the alternative operator alt is sketched, but no loop defined. In [19] a Petri-net semantics for UML 1.x sequence diagrams is presented, but as with the Petri net semantics of basic MSCs it has major limitations and is of little interest.

Kosiuczenko and Wirsing [96] make a formalization of MSC-96 in Maude. Their semantics is problematic for two reasons: Every lifeline in a diagram is translated into an object specified in Maude, and the behavior of these objects are specified by the means of states and transition rules. We find this problematic because in general the states of a lifeline are implicit while this formalism assumes explicit states. Further,

this way of reducing diagrams to sets of communicating objects has the effect that all choices are made locally in the objects and the choice operator alt looses its semi-global nature. Hence, this formalization does not capture the intended understanding of the alt operator.

An interesting approach is that of Jonsson and Padilla [89]. They present a semantics of MSC which is based on syntactical expansion and projection of diagram fragments during execution. Each lifeline is represented by a thread of labels where the labels refer to events or diagram fragments. The threads are executed in parallel and when a label referring to a fragment is reached the fragment is projected and expanded into the threads. Among the approaches referred to in this chapter, this is probably the approach that most resembles our operational semantics since their execution/expansion scheme resembles our execution/projection scheme. Because expansion produces silent events, the semantics can with small changes be extended with neg and xalt. Expansions may happen at arbitrary points since there are no rules in the semantics itself for when to expand. This creates a need for execution strategies, and the approach may be seen as having an informal meta-level where ad hoc strategies are described. However, if completeness is to be ensured, or if the semantics is to be extended with negative and potential/mandatory behavior this meta-level must be formalized. A problem with this semantics is that it requires explicit naming of all diagram fragments and this yields an unnecessary complicated syntax. Another shortcoming is the lack of an explicit communication medium; the communication model is "hardcoded" into the semantics and does not allow for variation.

Live Sequence Charts (LSC) [39, 61, 62] is a variant of MSC that does define a meta-level. Diagrams may be tagged as universal or existential, and parts of diagrams as hot or cold, and this is evaluated at the meta-level. This yields something similar to the potential/mandatory distinction, and allows for specifying negative behavior. There is however a difference; LSCs specify what must or may happen given that some conditions are fulfilled, while our semantics distinguish between choices that must or may be present in an implementation. Further the semantics complies with neither the MSC nor the UML standard. Most importantly it requires synchronization between lifelines at every entry point of diagram fragments, e.g. when resolving a choice.

As we see, none of these approaches to defining operational semantics for UML sequence diagrams and MSCs fulfill the intentions and criteria we have set for our semantics. The shortcomings that dominate are:

- Non-conformance with the intended semantics of UML.
- No notion of explicit negative behavior and no distinction between negative behavior and inconclusive behavior (behavior that is neither positive nor negative).
- No distinction between potential and mandatory choice. (This is no surprise as the alt/xalt distinction is an invention of STAIRS.)
- Lack of a proper meta-level that may be used for assigning meaning to negative and potential/mandatory behavior.
- Lack of possibility and freedom in defining and formalizing a meta-level.
- Lack of modifiability and extensibility, e.g., with respect to the communication model.

- Requiring transformations from the textual syntax into the formalism of the approach.

Since our intentions have been to stay close to the UML standard in both syntax and semantic, and to facilitate ease of extension and modification when adapting the semantics to different interpretations and applications of sequence diagrams, it is clear that none of the existing approaches suffice for our purposes.

Part IV

Testing

Chapter 12

Introduction to testing

Most system development methodologies recommend testing as the method for improving the correctness and quality of the software developed. In the most general case, testing means that an executable program is subjected to experiments where the program is observed in order to find out how it behaves under certain circumstances. Typically the circumstances are input to the program and what is observed is the output of the program, in which case the process is called functional testing.[1] A typical definition of functional testing is found in [16, p. 4]:

> In functional testing the element is treated as a black box. It is subjected to inputs and its outcomes are verified for conformance to its specification.

This definition also brings in a second element, i.e. that the observations that are made should be used for ensuring that the program is operating in accordance with the specification of the program.

In what is usually referred to as model based testing or specification based testing, this is taken a step further. In [183], four main approaches that may be called model based testing are characterized:

1. Generation of test input data from a domain model.
2. Generation of test cases from an environment model.
3. Generation of test cases from a behavioral model.
4. Generation of test scripts from abstracts tests.

In the following we concentrate on the third of these forms of model based testing. Test are derived from formal or semi-formal models (specifications) of the system under development – often state machines, statecharts or as in our case sequence diagrams – and the tests are used for investigating whether the program conforms to the model (specification).

Important system development methodologies, such as the Rational Unified Process (RUP) [97], and other variants of the Unified Process, emphasize use case based and scenario based development. In such development, requirements to the system

[1] There are of course other types of testing as well (see e.g. [91, 178, 188] for different overviews). E.g. if the circumstances are the hardware the program is executed on and what is observed is how fast it performs, we can talk about performance testing. In the following, however, we concentrate on functional testing.

are specified as usage scenarios which then become the basis for system design, implementation and testing. Also agile methods like Agile Model Driven Development (AMDD) [8] and Test Driven Development (TDD) [88] recommend scenario or use case based specification.

Sequence diagrams easily find their place in this kind of development, since they can be applied for specifying usage scenarios. Both their focus on interactions and their partial nature make them well suited for this. This in itself motivates test generation from sequence diagrams, because it combines the approach of model based or specification based testing with the approach of use case or scenario based development. In [183] it is shown how a model based testing approach fits together with the Unified Process and with agile methods, and in [152] an agile methodology based on modeling with UML and model based testing is outlined.

To bring matters even a step further, we argue for testing not only implementations, but also specifications. In an iterative model based development process, like e.g. RUP, both requirement specifications (use cases) and design specifications are likely to become more and more refined as the project loops through the iterations. In the same way as we would be interested in investigating whether an implementation conforms to the specification, we should also be interested in investigating whether a refinement of a specification conforms to the original specification (if not for any other reason then that the implementation is more likely to be based on the refinement). Because we have an operational semantics for sequence diagrams, that essentially executes a sequence diagram, we can treat a sequence diagram specification as a program for the purpose of testing. By this observation we can use testing as a means for investigating whether a refinement of a sequence diagram specification is correct. We refer to this as refinement testing.

Many of the testing methods that have been created are based on heuristics, and must be considered informal (see e.g. [91, 178, 179]). Our view is that the methods we apply for analysis should have a formally sound foundation, and we hold this view also with respect to methods for testing. When we define our testing method for sequence diagrams it therefore makes sense to use formal testing theory as the staring point.

For the remainder of this chapter we have a brief look at formal testing theory. In chapter 13 we present our testing theory for sequence diagrams, and in chapter 14 a tool based on our operational semantics and our testing theory. Chapter 15 provides a case study using the tool and chapter 16 provides related work.

Formal testing theory In the above definition of functional testing it is stated that the outcome of the testing should be verified for conformance to its specification. This raises the question of how we should understand "conformance to specification". In order to answer such a question in a satisfactory manner it is essential to have a theory for testing that precisely states what conformance is.

In this context the most notable (formal) models for computer programs and processes are process algebras/calculi (such as CCS [125] and CSP [74]), labeled transition systems (LTS) [128] and finite state machines (FSM) [102, 105]. The studies into these formal models have lead to the definition of a number of equivalence relations and preorders on processes, such as observation equivalence, trace equivalence, simulation, bisimulation, and several more.

In [129], De Nicola and Hennessy formulated a theory for what they called "testing equivalence" for CCS based on the notion of observer processes, and showed how the

"testing preorders" they defined could be used to characterize a number of already defined equivalence relations. In [128] the same testing relations where defined for labeled transition systems. Others, such as Phillips [142] and Abramsky [3] continued this work by defining even stronger testing relations.

The purpose of formulating these testing relations was to study the semantics of programs and processes, but the theory turned out to be suitable as a theoretical foundation for software testing. In [23], Brinksma made the first attempts at using this theory for test generation. The elaboration of this has led to the definition of a number of conformance relations, and several tools based on these theories have been developed (see e.g. [24, 25, 177, 180, 184]). A similar development originated from the theory of finite state machines (see e.g. [32, 93, 95, 102]).

All formal testing theory is based on what is called the *test hypothesis*, which may be formulates as follows [24]:

> Correctness and validity of [the] specification is assumed [...] [T]here is an implementation [...] which is treated as a black box, exhibiting external behavior. The [implementation] is a physical, real object that is in principle not amendable to formal reasoning. We can only deal with implementations in a formal way, if we make the assumption that any real implementation has a formal model with which we could reason formally. This formal model is only assumed to exist, but need not be known a priori. [This] allows to reason about implementations as if they were formal objects, and, consequently, to express conformance of implementations with respect to specifications by means of a formal relation between such models of implementations and specifications.

An implication of this assumption is that for a given formalism we can formulate a testing theory, as long as we are able to characterize observation in the formalism and we accept it as a model for implementations. A testing theory is general in the sense that it is not based on any specific formalism, but rather on general principles of processes and observations.

Having a formal theory for testing makes it possible to answer the question of exactly what we mean by conformance and what it means for a test to fail or succeed. Further it is possible to say formally what it means for a set of tests to be sound or complete, give exact definitions of test coverage and formally state requirements for test selection and test prioritizing.

Input-output conformance testing We base our work on the theory called *input-output conformance testing*, or "ioco-testing" for short [177–179]. The theory is formal and is based on labeled transition systems (LTSs). This makes it suitable as a starting point for us, as our operational semantics can easily be related to LTSs.

The LTSs of the ioco theory distinguishs between input and output actions. In addition, in these LTSs it is also possible to observe the absence of output, called *quiescence*. Quiescence is treated as an action δ, and hence it can appear in the traces of an LTS. A trace where δ is allowed to appear repeatedly is called a suspension trace.

A conformance relation $\_\,\mathbf{ioco}\,\_$ relates implementations and specifications such that i **ioco** s holds if the implementation i conformers to the specification s, where i and s are LTSs. The relation is defined as

$$i\ \mathbf{ioco}\ s \stackrel{\mathsf{def}}{=} \forall \sigma \in Straces(s) : out(i\ \mathbf{after}\ \sigma) \subseteq out(s\ \mathbf{after}\ \sigma)$$

where $Straces(s)$ is the set of all finite suspension traces of s, _ **after** _ is a function that returns all states of an LTS reachable by a (suspension) trace and *out* returns all outputs, including δ, possible in a set of states. In other words, the relation states that after every finite suspension trace, the possible outputs of the implementation should be a subset of the outputs possible by the specification. It can be shown that this relation is stronger than trace inclusion (i.e. $traces(i) \subseteq traces(s)$) because the quiescence can distinguish paths with the same finite traces but that differs with respect to quiescence in their states.

The ioco theory also provides a test generation algorithm. The algorithm is based on providing stimulus to a specification and making observations of its outputs or absence of outputs, and based on that assigning verdicts **pass** and **fail** to the execution paths. The algorithm has the property that, if U is a complete set of tests (test suite) generated from a specification s, then

$$\forall i : i \textbf{ ioco } s \iff i \textbf{ passes } U$$

where i **passes** U expresses that none of the tests in U, when executed in parallel with i, has an execution path that result in a verdict **fail**.

Chapter 13

Testing of sequence diagram specifications

In this chapter we define an algorithm for generating tests from UML 2.x sequence diagrams based on the operational semantics defined in chapter 8. Further we show how these tests can be applied for testing of refinement. The algorithm is a modified and adapted version of the algorithm presented in [178, 179] that takes into consideration the partial nature of sequence diagrams as well as the notion of negative and universal behavior introduced in UML 2.x with the operators refuse/neg and assert. We do not consider the xalt operator. Distinguishing between potential and mandatory choice requires building semantic structures that are not available in the framework we have chosen for our testing theory. Hence, in this chapter we assume that the semantics of a diagram d is a single interaction obligation, $[\![d]\!] = (p, n)$. We also make the restriction that the loop operator is provided with a finite counter and that any generalized alt operator have a finite index set. This ensures that p and n are finite sets of finite traces. Because the STAIRS semantics considers prefixes of positive behavior as inconclusive (unless assert is applied, in which case they are negative behavior) we seek to generate maximal tests, i.e. tests that represent complete behavior. This is only possible if the specification from which we generate traces has finite behavior.

The algorithm generates tests in the form of sequence diagrams. These are abstract tests in the terminology of [183], i.e. tests defined by means of the abstract operations and values used in the specification the tests are generated from. We see the possibility of transforming our abstract tests into test scripts, i.e. executable test cases designed for testing implementations built on specific technologies [183] (see also the discussion of the UML testing profile in chapter 16), but do not pursue this any further.

In order to test refinements we execute these abstract tests against other specifications. In conformance testing, tests are generated from specifications and executed against implementations. We define test execution where the "implementation" is a sequence diagram. Conceptually this is unproblematic, since formal conformance testing always assumes the existence of a formal model of the implementation even though this model is unknown (the test hypothesis). We, instead, do test execution against a formal model. A difference, however, is that an implementation can always be assumed to be complete while our specifications may be partial.

This partiality of specifications has the result that generating tests from an abstract specification and executing them against a more concrete specification does not capture all fault situations we may want to detect. Specifically, if a trace is explicitly specified

in the abstract specification as positive or negative, but is inconclusive (i.e. unspecified) in the more concrete specification, this is an incorrect refinement we are not able to capture by this way of testing.

We therefore make a distinction between "testing down" and "testing up". "Testing down" refers to the process described above where tests are generated from an abstract specification and executed against a more concrete specification. "Testing up" refers to the process of generating tests from the concrete specification and executing them against the abstract specification. By "testing up" we capture the incorrect refinements overlooked by "testing down", and hence the two "testing directions" complement each other. However, if we are in a position where we can assume that the "implementation" is complete, "testing down" is sufficient.

In sum the contents of this chapter is: 1) A test generation algorithm based on a well-known algorithm for conformance testing and a formal operational semantics for sequence diagrams, which takes as input sequence diagrams that may contain the operators assert and neg/refuse and that produces abstract tests in the form of sequence diagrams. 2) A characterization of how the same operational semantics can be applied to execute the tests against sequence diagrams and how this may be used to test refinement steps in system development. We are not aware of any other approach that does this.

The structure of the chapter is as follows: Section 13.1 presents the format of our tests and section 13.2 the test generation algorithm. In section 13.3 we characterize test execution. In addition, section 13.4 provides some considerations with respect to asynchronous testing.

13.1 Test format

In the following we will have two kinds of sequence diagrams. "Normal" sequence diagrams that specify systems we refer to as *specifications*. The other kind of diagrams we call *tests*. Tests are special sequence diagrams that may be seen as specifications of test programs or test cases.

A test is a sequence diagram consisting of only one lifeline. We use *tester* as a generic name of this lifeline. A test specifies how a test program operates by sending messages to the system under test and observing the messages received. In a test we may also specify observation of absence of messages from the system under test. Intuitively this can be thought of as timeouts in the test programs as they wait for output from the system under test. In a test we also specify verdicts that the test program will return after being executed against the system under test. Based on the observations the test program does, it will give one out of three possible verdicts: **pass**, **fail** or **inconclusive**.

The timeout/absence of reception is represented in a test by a special event $\theta(l)$ where l is the lifeline of the test. Let $\Theta \stackrel{\mathsf{def}}{=} \{\theta(l) \mid l \in \mathcal{L}\}$ and $\Theta \cap (\mathcal{E} \cup \mathcal{T}) = \emptyset$. We let θ range over the elements of Θ and use θ as shorthand notation for $\theta(l)$ when l is implicitly given. We let $l.\_$ also range over Θ so that $l.\theta(l') \stackrel{\mathsf{def}}{=} l'$.

The verdicts are represented by three terminating diagrams (similar to skip). These three terminating diagrams then extend the definition of sequence diagrams:

$$\mathsf{pass}, \mathsf{fail}, \mathsf{inconc} \in \mathcal{D}$$

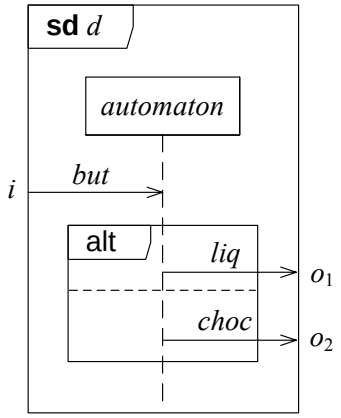


Figure 13.1: Example specification

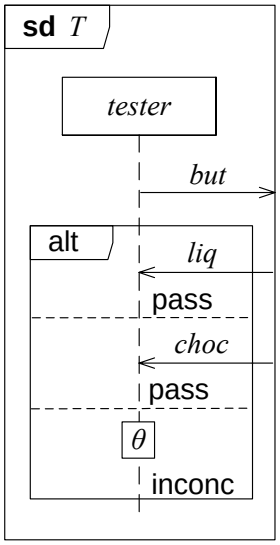


Figure 13.2: Example test

We let $\mathcal{V} \stackrel{\text{def}}{=} \{\mathsf{pass}, \mathsf{fail}, \mathsf{inconc}\}$ denote the set of verdicts.

In figure 13.1 we show a specification of a simple vending machine; if a button (represented by the message *but*) is pushed, the machine will make a non-deterministic choice and provide either liquorice or chocolate (represented by the messages *liq* and *choc*).[1] A test generated from this specification would typically look like:

$$T = !but \ \mathsf{seq} \ (?liq \ \mathsf{seq} \ \mathsf{pass} \ \mathsf{alt} \ ?choc \ \mathsf{seq} \ \mathsf{pass} \ \mathsf{alt} \ \theta \ \mathsf{seq} \ \mathsf{inconc})$$

A graphical representation of this test is shown in figure 13.2. The intuitive understanding of this test should be the following: The test program sends a message *but* to the system under test and then waits for output from the system. If the test observes the system sending a message *liq* or a message *choc* it terminates with the verdict pass because these are positive behaviors in the specification d in figure 13.1 from which it is generated. If the test observes absence of output from the system, this is inconclusive behavior in the specification d and the result is the verdict inconc.

13.2 Test generation

We define an algorithm for test generation. The algorithm takes specifications as input and produces tests of the format shown above. The algorithm is based on the algorithm provided in [178,179], which produce tests in the form of LTSs from LTS specifications. The execution system defined in section 8.1 may be interpreted as an LTS, and because of this, defining test generation in the fashion of [178, 179] is a natural choice for our algorithm.

Both the UML standard and the STAIRS semantics divide the trace universe into positive, negative and inconclusive traces for a given diagram, and hence regard sequence diagrams as partial specifications. This differs from the approach of [178, 179] which only operates with positive and negative traces. Hence it lacks the distinction between negative and inconclusive behavior and the distinction between potential and

[1] This vending machine example is borrowed from [179].

universal behavior we get from the sequence diagram operators refuse/neg and assert. Therefore, one of the modifications we make to the test generation algorithm is an adaption to a semantic model with inconclusive traces.

13.2.1 Preliminaries

As we have shown earlier (see section 7.3 and chapter 9), a meta-level is needed to make the distinction between positive and negative behavior and between potential and universal behavior. Because knowing these properties of executions is necessary in the test generation (in order to have the tests terminate with the appropriate verdicts), we apply the meta-system for generating traces with mode in the test generation algorithm. (See section 9.1 for more background on the mode concept.) We need to distinguish both negative and universal behavior from normal (positive partial) behavior. We use the following set of modes:

$$\mathcal{MO} \stackrel{\mathsf{def}}{=} \{\mathsf{norm}, \mathsf{ref}, \mathsf{ass}, \mathsf{assref}\}$$

The mode norm denotes a normal execution. ref denotes an execution that has entered a refuse operator and ass denotes an execution that has entered an assert operator, while assref denotes that both a refuse and an assert have been entered. The *mode* function is defined as:

$$mode(\tau, mo) \stackrel{\mathsf{def}}{=} \begin{cases} \mathsf{ass} & \textbf{if } \tau = \tau_{assert} \wedge mo = \mathsf{norm} \\ \mathsf{ref} & \textbf{if } \tau = \tau_{refuse} \wedge mo = \mathsf{norm} \\ \mathsf{assref} & \textbf{if } (\tau = \tau_{refuse} \wedge mo = \mathsf{ass}) \vee \\ & \quad (\tau = \tau_{assert} \wedge mo = \mathsf{ref}) \\ mo & \textbf{otherwise} \end{cases} \tag{13.1}$$

Before we present the test generation algorithm, we also introduce some auxiliary definitions. First of all we distinguish between internal and external events. The external events of a specification d are those events that transmit messages to gates or receive messages from gates. We define E^d_O and E^d_I to be the external transmit events and external receive events of d, respectively:

$$\begin{aligned} E^d_O &\stackrel{\mathsf{def}}{=} \{e \in ev.d \mid k.e = ! \wedge l^{-1}.e \in \mathcal{G}\} \\ E^d_I &\stackrel{\mathsf{def}}{=} \{e \in ev.d \mid k.e = ? \wedge l^{-1}.e \in \mathcal{G}\} \end{aligned}$$

The events in $ev.d \setminus (E^d_O \cup E^d_I)$ are then the internal events. A function $enabledEvents \in \mathcal{EX} \to \mathbb{P}(\mathcal{E})$ returns the set of enabled events of an execution state:

$$enabledEvents([\beta, d]) \stackrel{\mathsf{def}}{=} \{e \in \mathcal{E} \mid \exists t \in \mathcal{T}^*, \beta' \in \mathcal{B}, d' \in \mathcal{D} : [\beta, d] \xrightarrow{t^\frown\langle e\rangle} [\beta', d']\}$$

Note that this function returns the proper events that are possible next events from an execution state, after the execution of any number of silent events.

We define quiescence in the manner of [179]. An execution state is *quiescent* if no external transmit events or internal events are enabled. This is the same as saying that only external receive events are enabled. We define a quiescence function $\delta \in \mathcal{EX} \to \mathbb{B}$ that returns **true** iff an execution state is quiescent:

$$\delta([\beta, d]) \stackrel{\mathsf{def}}{=} enabledEvents([\beta, d]) \subseteq E^d_I$$

Quiescence may be treated as an observable event. We let δ be a special event that we use as a placeholder for observation of quiescence in a specification; is not part of the syntax of the specifications but an abstract event we make use of to denote quiescence while generating tests.

A function $out \in \mathcal{EX} \to \mathbb{P}(\mathcal{E} \cup \{\delta\})$ returns the set of possible observations of an execution state of a specification:

$$out([\beta, d]) \stackrel{\mathsf{def}}{=} \begin{cases} \{\delta\} & \textbf{if } \delta([\beta, d]) \\ E_O^d \cap enabledEvents([\beta, d]) & \textbf{if } \neg\delta([\beta, d]) \end{cases}$$

The test generation algorithm involves maintaining a set of states. We do have the execution states, but also need a way of keeping track of the modes of these execution states. We therefore define a *test generation state* as a pair of an execution state and a mode, and let $\mathcal{ST}$ denote the set of all test generation states:

$$\mathcal{ST} \stackrel{\mathsf{def}}{=} \mathcal{EX} \times \mathcal{MO}$$

In the following we let S range over sets of test generation states, i.e., $S \subseteq \mathcal{ST}$. We overload the function *out* to range over sets of test generation states, and define:

$$out(S) \stackrel{\mathsf{def}}{=} \bigcup_{([\beta,d],mo) \in S} out([\beta, d])$$

The *out* function then returns the set of possible observations from a set of states, independent of the modes of the states.

Similar to *out*, we define an overloaded function $in \in \mathcal{EX} \to \mathbb{P}(\mathcal{E}), \mathbb{P}(\mathcal{ST}) \to \mathbb{P}(\mathcal{E})$:

$$in([\beta, d]) \stackrel{\mathsf{def}}{=} E_I^d \cap enabledEvents([\beta, d])$$

$$in(S) \stackrel{\mathsf{def}}{=} \bigcup_{([\beta,d],mo) \in S} in([\beta, d])$$

Let E_O be the set of external events. We define an overloaded function $\_\textbf{after}\_ \in \mathcal{ST} \times E_O^* \to \mathbb{P}(\mathcal{ST}), \in \mathbb{P}(\mathcal{ST}) \times E_O^* \to \mathbb{P}(\mathcal{ST})$ that returns the new test generation states after execution of a trace of external events:

$$\begin{aligned} ([\beta, d], mo) \textbf{ after } t \stackrel{\mathsf{def}}{=} \{([\beta', d'], mo') \mid \\ \langle\langle\rangle, [\beta, d], gates.d, mo\rangle \longrightarrow^* \langle t, [\beta', d'], gates.d, mo'\rangle\} \end{aligned}$$

$$S \textbf{ after } t \stackrel{\mathsf{def}}{=} \bigcup_{([\beta,d],mo) \in S} ([\beta, d], mo) \textbf{ after } t$$

In the definition, $\longrightarrow^*$ denotes the reflexive, transitive closure of the transition relation $\longrightarrow$ defined by the meta-system. In order to get external traces from the meta-system we use $gates.d$ as the set of lifelines.

13.2.2 Test generation algorithm

We define a test generation algorithm that takes a specification as input and produces a test as output. Both the specification and the test is in the form of sequence diagrams. The structure of the algorithm is as follows:

- A set of test generation states S and a set of events E are maintained. S changes throughout the test generation, and initially it is the set of all states reachable by zero or more internal or silent events, i.e. $([initC.d, d]$ **after** $\langle\rangle)$ if d is the specification from which the test is generated from. E is a set of transmit events such that $E_O^d \subseteq E$ and remains unchanged thoughout the test generation. We allow E to be bigger than E_O^d in order to generate tests that can observe a bigger alphabet than the specification we are generating tests from.
- For each step there is a non-deterministic choice of whether the test should 1) observe the system under test or 2) provide stimulus to the system. Later we formalize the algorithm by means of three rules (13.2), (13.3) and (13.2).
  - If the first option, formalized by rule (13.2), is chosen, the set of all possible observations that can be made in the curret set of states is calculated, possibly including δ.
    - For each event e in E a branch is made where e is converted to a receive event of the test. For the cases where e is among the possbible observation the test generation continues with a new set of test generation states (S **after** $\langle e\rangle$). In the cases e is not in E, we have observed something not specified by the specification. We then get the verdict fail or inconc depending on the set of test generation states S and the test generation terminates in this branch.
    - An additional branch is also made where the test specifies a timeout event θ. In this branch test generation continues (with the set of test generation states (S **after** $\langle\rangle$)) only if δ is among the possible observations. In the other case we have observed something not specified by the specification; we get the verdict fail or inconc depending on the set of test generation states S and the test generation terminates for that branch.

    In order to avoid an inifinite recursion of producing timeout events in the test generation we insist that immediately after a timeout event has been generated, the choice of providing stimulus must be chosen.
  - If the option, formalized by rule (13.3), of letting the test make stimulus to the system is chosen, the set of receive events that the states of S accept is calculated. One of these events is chosen, let us call it e, and transformed to a transmit event of the test. The test generation then continues with the set (S **after** $\langle e\rangle$) of states.
  - If the last event produced in a branch of the test generation is a timeout event and set of possible stimuli is empty, we have nothing more to do and the generation of this branch terminates. We then give the verdict pass or fail depending on the set of test generation states S and the "testing direction". Because we are interested in maxiaml tests (i.e. complete observations), this is the only way we can get the verdict pass. (When we have termination as decribed in the first point we have inconclusive behavior unless assert is used in the specification.) This termination is formalized in rule (13.4).

The algorithm is formalized by means of a function:

$$testGen \in \mathbb{P}(\mathcal{ST}) \times \{0, 1\} \to \mathcal{D}$$

The function is defined by means of three rules shown below. Because the algorithm sometimes makes non-deterministic choices, the application of these rules is non-deterministic. The restrictions in the application of the rules explained above is guided by a bit such that the bit is 1 when a timeout was the last generated event, or else 0.

When a test is built with the algorithm, we need to construct the events of the test from the events of the specification. A transmit event of the diagram corresponds to a receive event of the test and vice versa. Further, the gates of the messages (remember we are only dealing with external events) must be replaced with the lifeline of the test. Let $tester \in \mathcal{L}$ be the lifeline of the test. We define a function $p \in \mathcal{E} \to \mathcal{E}$ that produce an event for the test from an event of the specification:

$$\begin{aligned} p((!,(s,l,g))) &\stackrel{\text{def}}{=} (?,(s,l,tester)) \\ p((?,(s,g,l))) &\stackrel{\text{def}}{=} (!,(s,tester,l)) \end{aligned}$$

where we have that $l \notin \mathcal{G}$ and $g \in \mathcal{G}$.

The rule (13.2) is responsible for generating the parts of a test that do observation of output from the system under test:

$$\begin{aligned} &testGen(S,0) = \\ &\quad [\mathsf{alt}_{e\in E}\ (p(e)\ \mathsf{seq}\ \textbf{if}\ e \in out(S)\ \textbf{then} \\ &\qquad\qquad testGen(S\ \textbf{after}\ \langle e\rangle, 0) \\ &\qquad\qquad \textbf{else}\ verdictObs(S)\ \textbf{fi})] \\ &\mathsf{alt}\ [\theta(tester)\quad \mathsf{seq}\quad \textbf{if}\ \delta \in out(S)\ \textbf{then} \\ &\qquad\qquad testGen(S\ \textbf{after}\ \langle\rangle, 1) \\ &\qquad\qquad \textbf{else}\ verdictObs(S)\ \textbf{fi}] \end{aligned} \tag{13.2}$$

Two main branches (enclosed in brackets) of the test are made, each of which becomes the argument of an alt operator. The first main branch is divided into one branch for each of the events in the set E of transmit events. For each event e in E, a branch is produced; in case e is an event the specification may provide the test generation continues; in the opposite case, the specification cannot provide e, so a verdict is given by the function *verdictObs* and the test generation terminates. The second main branch represents observation of timeout; if quiescence is a possible observation in one of the states in S, a timeout in the test is considered correct and the test generation continues, or else a verdict based on S is provided by the function *verdictObs*. The rule can only be applied if the argument bit is 0 in order to avoid infinite recursion of generating timeout events.

The verdict function $verdictObs \in \mathbb{P}(\mathcal{S}) \to \mathcal{V}$ is defined as follows:

$$verdictObs(S) \stackrel{\text{def}}{=} \begin{cases} \mathsf{fail} & \textbf{if}\ \forall([\beta,d],mo) \in S : mo \in \{\mathsf{ass},\mathsf{assref}\} \\ \mathsf{inconc} & \textbf{if}\ \exists([\beta,d],mo) \in S : mo \in \{\mathsf{norm},\mathsf{ref}\} \end{cases}$$

If all states have mode either ass or assref, we only have universal behavior and observing something not specified by the in specification would be bad. In all resulting cases we just observe something unspecified, and hence the behavior is inconclusive.

The rule (13.3) generates the parts of the test that provide stimuli:

$$testGen(S,b) = p(e)\ \mathsf{seq}\ testGen(S\ \textbf{after}\ \langle\rangle, 0)\ \textbf{for some}\ e \in in(S) \tag{13.3}$$

This stimulus must be something that one of the test generation states of S is ready to accept, i.e. a specified reception, and is chosen non-deterministically from the set $in(S)$. The value of the argument bit b is irrelevant for this rule. After the event, the test generation continues with all test generation states that are reachable after execution of this event.

The rule (13.4) is the one that terminates the test generation:

$$testGen(S, 1) = verdictTerm_{dir}(S) \textbf{ if } in(S) = \emptyset \tag{13.4}$$

As we see, this rule only applies if the argument bit is 1, i.e. if the last generated event was a timeout, and if the set of possible stimuli $in(S)$ is empty. Because this ensures maximal tests, the verdicts given by this rule is different from the verdicts given by rule 13.2 above. The verdicts are determined by a function $verdictTerm_{dir} \in \mathbb{P}(\mathcal{ST}) \rightarrow \mathcal{V}$:

In the assignment of these verdicts it is necessary to consider whether the test is generated for "testing down" or "testing up". The subscript *dir* is instantiated by *down* or *up* depending on the selected testing direction. Because this branch of the test is maximal, we can use the verdicts pass and fail instead of inconc, but we have to use them differently depending on the direction. If the test is generated for "testing down", positive behavior is considered more general than negative behavior. The reason for this is that in a refinement $d_1 \rightsquigarrow_r d_2$, a positive trace of d_1 can be both positive or negative in d_2 and we still have a correct refinement. Hence, the verdict pass is given if there is at least one state with mode norm or ass in this branch, and the verdicts are as follows:

$$verdictTerm_{down}(S) = \begin{cases} \mathsf{pass} & \textbf{if } \exists([\beta, d], mo) \in S : mo = \mathsf{norm} \vee mo = \mathsf{ass} \\ \mathsf{fail} & \textbf{if } \forall([\beta, d], mo) \in S : mo = \mathsf{ref} \vee mo = \mathsf{assref} \end{cases}$$

On the other hand, if the test is generated for "testing up", negative behavior is considered more general than positive behavior because a negative trace in d_2 can be either positive or negative in d_1. Therefore we give the verdict fail if at least one state at this point in the generation has the verdict ref or assref:

$$verdictTerm_{up}(S) = \begin{cases} \mathsf{pass} & \textbf{if } \forall([\beta, d], mo) \in S : mo = \mathsf{norm} \vee mo = \mathsf{ass} \\ \mathsf{fail} & \textbf{if } \exists([\beta, d], mo) \in S : mo = \mathsf{ref} \vee mo = \mathsf{assref} \end{cases}$$

We start test generation by applying the function *testGen* function on the set of test generation states reachable from the initial state of the specification by execution of only silent and internal events. Hence, if want to generate a test $T \in \mathcal{D}$ from specification d, we apply:

$$T = testGen(([initC.d, d], \mathsf{norm}) \textbf{ after } \langle\rangle, 0)$$

Because the application of the rules defined above are non-deterministic, there may be several possible tests that can be derived from d, and a new application of the *testGen* function may provide a different result. However, because of the restrictions on the specifications from which we generate tests (loops only with finite counter and generalized alts only with finite index set), and because of the restrictions with respect to timeout events in the algorithm, the test generation will always terminate.

A set of tests we refer to as a *test suite*. The maximal test suite that can be derived from a specification d can expressed by:

$$U = \{T \mid T = testGen(([initC.d, d], \mathsf{norm}) \textbf{ after } \langle\rangle)\}$$

13.3 Test execution

In this section we show how we can apply the operational semantics to execute the tests we generate against specifications. In order to do that we need to define a special modified projection system for tests. Further we define a special meta-system for test execution.

13.3.1 Projection system for tests

Test execution is, as test generation, based on our operational semantics. We do, however, make two changes to the operational semantics of tests:

- Tests consists of timeout events in addition to transmit and receive events. We must allow tests to execute these timeout events.
- In a test, an alt always occur in connection with observations of the system under test. We do not want a test to non-deterministically chose what it wants to observe, but to let the actual observation determine the choice. Therefore, the alts of tests are deterministic choices based on observations, as opposed to non-deterministic choices in specifications.

In order to achieve this we define a projection system for tests. We denote this system the Π' system. The Π' system consists of rules identical to the Π system's rules for events and seq, i.e. the rules (8.10), (8.11) and (8.12) on page 61. In addition the system has two rules. The first is a rule for execution of timeout events:

$$\Pi'(L, \beta, \theta) \xrightarrow{\theta} \Pi'(L, \beta, \mathsf{skip}) \textbf{ if } \theta \in \Theta \wedge l.\theta \in L \tag{13.5}$$

The second additional rule is a rule that resolved an alt in connection with execution of an event:

$$\begin{aligned} &\Pi'(L, \beta, d_1 \textsf{ alt } d_2) \xrightarrow{e} \Pi'(L, \beta, d'_k) \\ &\quad \textbf{if } \Pi'(L, \beta, d_k) \xrightarrow{e} \Pi'(L, \beta, d'_k),\ k \in \{1, 2\} \end{aligned} \tag{13.6}$$

This Π' system is only applied for tests, while specification being tested use the original Π system. Because seq and alt are the only operators of tests, rules (8.10), (8.11), (8.12), (13.5) and (13.6) are sufficient for a projection system for tests.

13.3.2 Meta-system for test execution

In the test generation algorithm the observation point is at the lifelines, which means we do not assume any communication medium between the lifelines and the points we do our observations. As a result of this we also have to define test execution with the lifelines as the observation point. (A way to simulate a communication medium is discussed in section 13.4.)

We achieve this by defining a special meta-system for test execution, similar to the system used in the test generation. In this system the specifications execute normally, produce τ events and use the communication medium for internal communication between their lifelines. However, execution of tests is in accordance with the Π' system defined above. Because we have the observation point directly at the lifeline and no communication medium between the test and the specification, messages sent between

the specification and the test are treated as being instantaneous. Further, the meta-system handles timeout events in the appropriate way.

The meta-system is a system

$$\langle \_, \_, \_, \_ \rangle \in \mathcal{H} \times \mathcal{EX} \times \mathcal{MO} \times \mathcal{D}$$

where the last place is for the test. In the following we define the rules of this system. The first two rules are concerned with execution of internal and silent events in the specification:

$$\begin{aligned} &\langle t, [\beta, d], mo, T \rangle \longrightarrow \langle t, [\beta', d'], mo, T \rangle \\ &\quad \textbf{if } [\beta, d] \xrightarrow{e} [\beta', d'] \wedge e \in \mathcal{E} \wedge tr.e \neq tester \wedge re.e \neq tester \end{aligned}$$

$$\begin{aligned} &\langle t, [\beta, d], mo, T \rangle \longrightarrow \langle t, [\beta', d'], mode(\tau, mo), T \rangle \\ &\quad \textbf{if } [\beta, d] \xrightarrow{\tau} [\beta', d'] \wedge \tau \in \mathcal{T} \end{aligned}$$

In the latter of these rules, the function *mode* is identical to the definition (13.1) above.

For communication between the test and the specification there are two rules; one for the case where the test sends a message and the specification receives, and one where the specification sends a message and the test receives:

$$\begin{aligned} &\langle t, [\beta, d], mo, T \rangle \longrightarrow \langle t^\frown \langle (!, m), (?, m) \rangle, [\beta, d'], mo, T' \rangle \\ &\quad \textbf{if } \Pi'(ll.T, \emptyset, T) \xrightarrow{(!,m)} \Pi'(ll.T, \emptyset, T') \wedge \\ &\qquad \Pi(ll.d, \{m\}, d) \xrightarrow{(?,m)} \Pi(ll.d, \{m\}, d') \end{aligned}$$

$$\begin{aligned} &\langle t, [\beta, d], mo, T \rangle \longrightarrow \langle t^\frown \langle (!, m), (?, m) \rangle, [\beta, d'], mo, T' \rangle \\ &\quad \textbf{if } \Pi(ll.d, \emptyset, d) \xrightarrow{(!,m)} \Pi(ll.d, \emptyset, d') \wedge \\ &\qquad \Pi'(ll.T, \{m\}, T) \xrightarrow{(?,m)} \Pi'(ll.T, \{m\}, T') \end{aligned}$$

In both rules we see that when the message m is transmitted and received, the trace $\langle (!, m), (?, m) \rangle$ is appended to the trace collected during the execution. This is due to the fact that communication between the test and the specification is instantaneous and no other events may come in between. However, the order of these events is important to preserve the message invariant in the trace, i.e. that the transmit of a message m must precede the receive of m.

The final rule is the rule that executes a timeout event in the test:

$$\begin{aligned} &\langle t, [\beta, d], mo, T \rangle \longrightarrow \langle t^\frown \langle \theta \rangle, [\beta, d], mo, T' \rangle \\ &\quad \textbf{if } \Pi'(ll.T, \emptyset, T) \xrightarrow{\theta} \Pi(ll.T, \emptyset, T') \wedge \theta \in \Theta \wedge \\ &\qquad \forall x \in \mathcal{E} \cup \mathcal{T} : \Pi(ll.d, \beta, d) \xrightarrow{x} \end{aligned}$$

This rule is applicable only if no external transmit or internal events in the specification is possible. As we can see, this is the same condition that makes a specification quiescent in the test generation algorithm. However, here we have no need for the δ event to denote the situation explicitly.

As in the test generation, we let $\longrightarrow^*$ denote the reflexive, transitive closure of $\longrightarrow$. Further we define the relation $\longrightarrow^!$ as:

$$\begin{aligned} &\langle t, [\beta, d], mo, T \rangle \longrightarrow^! \langle t', [\beta', d'], mo', T' \rangle \stackrel{\mathsf{def}}{=} \\ &\quad \langle t, [\beta, d], mo, T \rangle \longrightarrow^* \langle t', [\beta', d'], mo', T' \rangle \wedge \langle t', [\beta', d'], mo', T' \rangle \not\rightarrow \end{aligned}$$

Similar to the function p defined as part of the test generation, we define a function $pr \in \mathcal{D} \to \mathcal{D}$ that prepares a specification for testing by substituting the gates with the lifeline of the test:

$$\begin{aligned}
pr((s, tr, re)) &\stackrel{\text{def}}{=} \begin{cases} (s, tr, re) & \textbf{if } tr \notin \mathcal{G} \wedge re \notin \mathcal{G} \\ (s, tester, re) & \textbf{if } tr \in \mathcal{G} \\ (s, tr, tester) & \textbf{if } re \in \mathcal{G} \end{cases} \\
pr((k, m)) &\stackrel{\text{def}}{=} (k, pr(m)) \\
pr(\mathsf{skip}) &\stackrel{\text{def}}{=} \mathsf{skip} \\
pr(\mathsf{op}\ d) &\stackrel{\text{def}}{=} \mathsf{op}\ pr(d),\ \mathsf{op} \in \{\mathsf{refuse}, \mathsf{assert}, \mathsf{loop}\} \\
pr(d_1\ \mathsf{op}\ d_2) &\stackrel{\text{def}}{=} pr(d_1)\ \mathsf{op}\ pr(d_2),\ \mathsf{op} \in \{\mathsf{seq}, \mathsf{strict}, \mathsf{par}, \mathsf{alt}\}
\end{aligned}$$

With this system a test run is characterized by:

$$\langle \langle\rangle, [\emptyset, pr(d)], \mathsf{norm}, T \rangle \longrightarrow^{!} \langle t, [\beta, d'], mo, v \rangle$$

Here, $T \in \mathcal{D}$ is the test and $d \in \mathcal{D}$ is the specification that we execute the test against. Further, $t \in \mathcal{H}$ is the trace resulting from the test execution, $mo \in \mathcal{MO}$ is the mode resulting from the execution of the specification and $v \in \mathcal{V}$ is the verdict given by the test. Note that the test may terminate before the specification, but also that we may have $d' = \mathsf{skip}$. Because we use the $\longrightarrow^{!}$ relation, the specification will be executed as much as possible (with internal and silent event) after the test has terminated.

13.3.3 Test runs and verdicts

We let the result of a test execution be a pair of a verdict and a trace $(v, t) \in \mathcal{V} \times \mathcal{H}$. We distinguish between the verdict that the test returns in the test execution and the verdict of the test run itself. The reason for this is that we also must take into consideration the mode of the test execution to determine the final verdict. Further, this also depends on the direction of the testing (whether we are "testing down" or "testing up") and the state of the specification after execution of the test.

The set of all test runs for a given test and specification can be characterized by a function:

$$exec_{dir} \in \mathcal{D} \times \mathcal{D} \to \mathbb{P}(\mathcal{V} \times \mathcal{H})$$

where dir indicates the direction of the testing and may stand for either *down* or *up*. In both cases the function is defined as:

$$\begin{aligned}
exec_{dir}(T, d) \stackrel{\text{def}}{=} \{(ver_{dir}(v, mo, d'), t) \mid \exists \beta, d' : \\
\langle \langle\rangle, [\emptyset, pr(d)], \mathsf{norm}, T \rangle \longrightarrow^{!} \langle t, [\beta, d'], mo, v \rangle\}
\end{aligned}$$

The function

$$ver_{dir} \in \mathcal{V} \times \mathcal{MO} \times \mathcal{D} \to \mathcal{V}$$

is used to determine the final verdict of each test run. It takes as its arguments the verdict of the test, the mode of the execution, and the state of the specification after execution of the test. We need the latter do distinguish complete behavior in the specification from prefixes of complete behavior. In the semantic model of STAIRS, only the complete behavior specified in a specification is considered positive; prefixes are considered inconclusive (or negative if assert is applied) if they are not at the same

time explicitly specified as complete behavior (something which may be the case if for example the operator opt is used in the specification). Further, an extension of positive behavior may be inconclusive behavior for the same reason. The same considerations also hold for negative behavior. If the state of the specification after the execution of the test equals skip, we know that the test run represents a complete behavior in the specification. In the other case, the test run represents behavior that is not complete in the specification, but a prefix of specified behavior.

The definitions of ver_{down} and ver_{up} are based on the notion of refinement of interaction obligations defined in [70], which is to say: Let d_1 and d_2 be diagrams with denotation $[\![d_1]\!] = (p_1, n_1)$ and $[\![d_2]\!] = (p_2, n_2)$. Then d_2 refines d_1, denoted $d_1 \rightsquigarrow_r d_2$, iff $p_1 \subseteq p_2 \cup n_2 \wedge n_1 \subseteq n_2$ (see also section 5.4).

When we define the function ver_{down}, this is based on the following analysis of the possible cases: Assume we are interested in the refinement $d_1 \rightsquigarrow_r d_2$. We generate a test T from specification d_1 and executes it against specification d_2, i.e. we are "testing down".

- If the test terminates with pass it means that the test run represents behavior that is specified as positive in d_1. Further, the pass indicates that the behavior is maximal in d_1. If the specification after the test execution equals skip, the test run represents complete (positive or negative) behavior in d_2. In this case we have correct refinement and keep the verdict pass.

  In the other case we also give the verdict pass for the following reason: The test run has a extension in d_2 (positive or negative) that is inconclusive behavior in d_1. We can know this because the test is maximal with respect to d_1. We can therefore consider the test run as a representative of inconclusive behavior in d_1 that becomes positive or negative in d_2, which is correct refinement.

- If the test terminates with inconc it means the test run represents behavior that is inconclusive in d_1. It is then irrelevant what status this behavior has in d_2 and the final verdict of the test run is pass regardless of the mode of the execution and the state of the specification.

- If the test terminates with fail it means the test run represents behavior that is negative in d_1. If the mode of the test execution is ref or assref, then the test run represents behavior that is negative in d_2, in which case it represents correct refinement and the final verdict is therefore changed to pass. If the mode is norm or ass, we may have two different situations, and we need more information to distinguish these two cases. If the specification has terminated, i.e. what remains of the specification is skip, the test run represents complete positive behavior in d_2. It is not a correct refinement and the verdict fail is given.

  If what remains of the specification is different from skip we have the situation, as we had above, that the rest run is a representative of a behavior that extends the test run in such a way that it is inconclusive in d_1 and positive or negative in d_2. In this case it can be considered a correct refinement and the final verdict is changed to pass.

In summary we see that there is one case that gives the verdict fail, and all other cases

give pass. Hence, we define ver_{down}:

$$ver_{down}(v, mo, d) \stackrel{\text{def}}{=} \begin{cases} \mathsf{fail} & \textbf{if } v = \mathsf{fail} \wedge mo \in \{\mathsf{norm}, \mathsf{ass}\} \wedge d = \mathsf{skip} \\ \mathsf{pass} & \textbf{otherwise} \end{cases}$$

For "testing up" we do a similar analysis. Again we are interested in the refinement $d_1 \rightsquigarrow_r d_2$, but this time we generate a test T from d_2 and execute it against d_1:

- If the test terminates with the verdict fail it means the test run represents behavior that is negative in d_2. Whatever the status of this behavior in d_1 it must be a correct refinement, and we return the final verdict pass.
- If the verdict of the test is inconc, the execution represents behavior that is inconclusive in d_2. Because the tests are maximal, there exist no extensions of the behavior that is not inconclusive in d_2. However, the behavior is specified in d_1 (as positive or negative). We are therefore dealing with an incorrect refinement where positive or negative behavior is made inconclusive, and the final verdict of the test run is given as fail.
- If the test gives the verdict pass, a more thorough analysis is necessary. The test run represents behavior that is positive in d_2, and the behavior must therefore be positive or inconclusive in d_1 for the refinement to be correct.

  If the remainer of the specification is equal to skip, the test run represents complete behavior in d_1. If the mode is norm or ass, this complete behavior is positive in d_1, and we give the verdict pass. On the other hand, if the mode is ref or assref, the behavior is negative in d_1, and we give the verdict fail.

  If the reminder of the specification is different from skip, the test run represents a prefix of specified behavior in d_1. If the mode is norm or ref, no assert operator has been executed during the test execution. The test run represents behavior that is inconclusive in d_1, and we give the verdict pass. If the mode is ass or assref, an assert operator have been executed during the test run. An assert makes all unspecified behavior negative, so the test run represent negative behavior in d_1 and the verdict fail must be given.

For "testing up" we then define these verdicts:

$$ver_{up}(v, mo, d) \stackrel{\text{def}}{=} \begin{cases} \mathsf{pass} & \textbf{if } v = \mathsf{fail} \vee \\ & \quad (v = \mathsf{pass} \wedge (mo = \mathsf{norm} \vee \\ & \qquad (mo = \mathsf{ass} \ \wedge \ d = \mathsf{skip}) \vee \\ & \qquad (mo = \mathsf{ref} \ \wedge \ d \neq \mathsf{skip}))) \\ \mathsf{fail} & \textbf{if } v = \mathsf{inconc} \ \vee \\ & \quad (v = \mathsf{pass} \wedge (mo = \mathsf{assref} \vee \\ & \qquad (mo = \mathsf{ass} \ \wedge \ d \neq \mathsf{skip}) \vee \\ & \qquad (mo = \mathsf{ref} \ \wedge \ d = \mathsf{skip}))) \end{cases}$$

The functions $exec_{dir}$ return sets of pairs of verdicts and traces. Each such pair represent a test run. Given a test T and diagram d we say that d passes the test T if no verdicts are fail, i.e.:

$$d \ \mathbf{passes}_{dir} \ T \iff \forall (v, t) \in exec_{dir}(T, d) : v = \mathsf{pass}$$

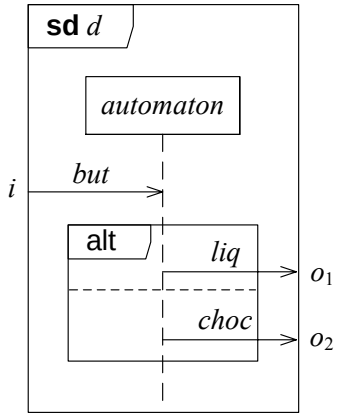


Figure 13.3: Spec. d

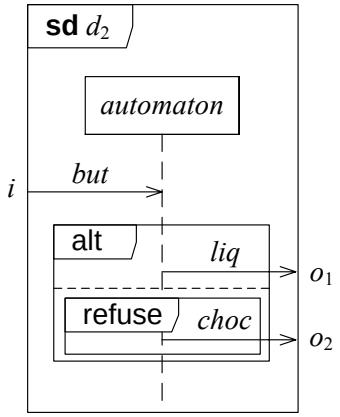


Figure 13.4: Spec. d_2

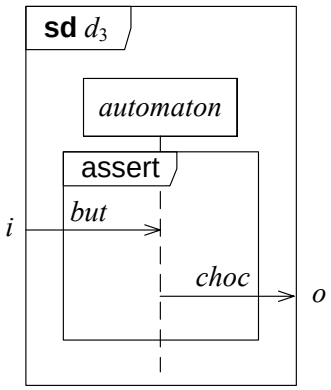


Figure 13.5: Spec. d_3

A diagram passes a test suite iff it passes all the tests of the suite:

$$d \textbf{ passes}_{dir}\ U \iff \forall T \in U : d \textbf{ passes}_{dir}\ T$$

If we want to compare two specifications d and d', and we have test suites U and U' generated from d and d' we can do this by checking whether

$$d' \textbf{ passes}_{down}\ U$$

and

$$d \textbf{ passes}_{up}\ U'$$

Together, these two operations constitute what we call refinement testing.

13.3.4 Examples

In figures 13.3, 13.4 and 13.5 we show three specifications d, d_2 and d_3 that are related in the following manner:

$$d \rightsquigarrow_r d,\ d \rightsquigarrow_r d_2,\ d \rightsquigarrow_r d_3,\ d_2 \rightsquigarrow_r d_2,\ d_3 \rightsquigarrow_r d_3$$

$$d_2 \not\rightsquigarrow_r d,\ d_2 \not\rightsquigarrow_r d_3,\ d_3 \not\rightsquigarrow_r d,\ d_3 \not\rightsquigarrow_r d_2$$

Figures 13.6, 13.7 and 13.8 show three tests T, T_2 and T_3 generated from d, d_2 and d_3, respectively.[2] Using these tests we now investigate the relation between the diagrams. The results of this testing are shown in table 13.1, where we use the following shorthand notation for the traces:

$$\begin{aligned} t_{liq} &= \langle !but, ?but, !liq, ?liq \rangle \\ t_{choc} &= \langle !but, ?but, !choc, ?choc \rangle \end{aligned}$$

[2]Several things should be noted about these tests. 1) We have that $E_O^d = E_O^{d_2} = \{!liq, !choc\}$, while $E_O^{d_3} = \{!choc\}$. In the generation of all three tests we have applied $E = \{!liq, !choc\}$. 2) The constraint that test generation only terminates after a timeout event is relaxed to make the tests more readable. 3) The tests are identical whether the option for "testing down" or for "testing up" is applied and can therefore be used in both testing directions.

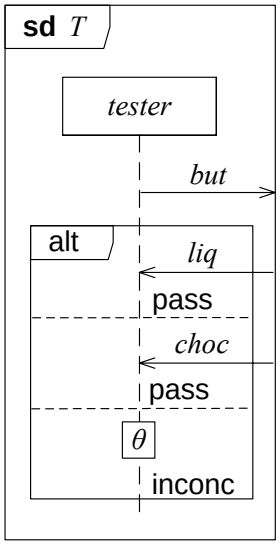


Figure 13.6: Test T

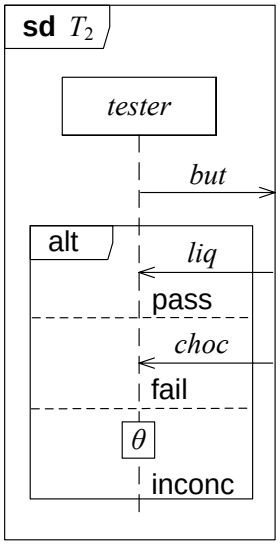


Figure 13.7: Test T_2

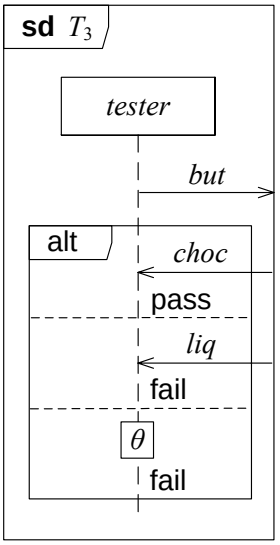


Figure 13.8: Test T_3

From these results we make three observations. First, we see that the specifications pass the tests in exactly the cases of correct refinement, which gives an indication that our testing scheme actually may be applied for checking of refinement. Secondly, we see that in two cases an incorrect refinement was refuted by "testing down" alone, which means "testing down" and "testing up" complement each other. Thirdly, we see that none of the incorrect refinements were refuted by "testing up" alone. This tells us that none of the incorrect refinements were due to introduction of new inconclusive behavior

Refinement	**Test runs**	**Result**
$d \rightsquigarrow_r d$	$exec_{down}(T, d) = \{(\mathsf{pass}, t_{liq}), (\mathsf{pass}, t_{choc})\}$ $exec_{up}(T, d) = \{(\mathsf{pass}, t_{liq}), (\mathsf{pass}, t_{choc})\}$	d **passes**$_{down}$ T d **passes**$_{up}$ T
$d \rightsquigarrow_r d_2$	$exec_{down}(T, d_2) = \{(\mathsf{pass}, t_{liq}), (\mathsf{pass}, t_{choc})\}$ $exec_{up}(T_2, d) = \{(\mathsf{pass}, t_{liq}), (\mathsf{pass}, t_{choc})\}$	d_2 **passes**$_{down}$ T d **passes**$_{up}$ T_2
$d \rightsquigarrow_r d_3$	$exec_{down}(T, d_3) = \{(\mathsf{pass}, t_{choc})\}$ $exec_{up}(T_3, d) = \{(\mathsf{pass}, t_{liq}), (\mathsf{pass}, t_{choc})\}$	d_3 **passes**$_{down}$ T d **passes**$_{up}$ T_3
$d_2 \not\rightsquigarrow_r d$	$exec_{down}(T_2, d) = \{(\mathsf{pass}, t_{liq}), (\mathsf{fail}, t_{choc})\}$ $exec_{up}(T, d_2) = \{(\mathsf{pass}, t_{liq}), (\mathsf{fail}, t_{choc})\}$	$\neg(d$ **passes**$_{down}$ $T_2)$ $\neg(d_2$ **passes**$_{up}$ $T)$
$d_2 \rightsquigarrow_r d_2$	$exec_{down}(T_2, d_2) = \{(\mathsf{pass}, t_{liq}), (\mathsf{pass}, t_{choc})\}$ $exec_{up}(T_2, d_2) = \{(\mathsf{pass}, t_{liq}), (\mathsf{pass}, t_{choc})\}$	d_2 **passes**$_{down}$ T_2 d_2 **passes**$_{up}$ T_2
$d_2 \not\rightsquigarrow_r d_3$	$exec_{down}(T_2, d_3) = \{(\mathsf{fail}, t_{choc})\}$ $exec_{up}(T_3, d_2) = \{(\mathsf{pass}, t_{liq}), (\mathsf{fail}, t_{choc})\}$	$\neg(d_3$ **passes**$_{down}$ $T_2)$ $\neg(d_2$ **passes**$_{up}$ $T_3)$
$d_3 \not\rightsquigarrow_r d$	$exec_{down}(T_3, d) = \{(\mathsf{fail}, t_{liq}), (\mathsf{pass}, t_{choc})\}$ $exec_{up}(T, d_3) = \{(\mathsf{pass}, t_{choc})\}$	$\neg(d$ **passes**$_{down}$ $T_3)$ d_3 **passes**$_{up}$ T
$d_3 \not\rightsquigarrow_r d_2$	$exec_{down}(T_3, d_2) = \{(\mathsf{fail}, t_{liq}), (\mathsf{pass}, t_{choc})\}$ $exec_{up}(T_2, d_3) = \{(\mathsf{pass}, t_{choc})\}$	$\neg(d_2$ **passes**$_{down}$ $T_3)$ d_3 **passes**$_{up}$ T_2
$d_3 \rightsquigarrow_r d_3$	$exec_{down}(T_3, d_3) = \{(\mathsf{pass}, t_{choc})\}$ $exec_{up}(T_3, d_3) = \{(\mathsf{pass}, t_{choc})\}$	d_3 **passes**$_{down}$ T_3 d_3 **passes**$_{up}$ T_3

Table 13.1: Test results

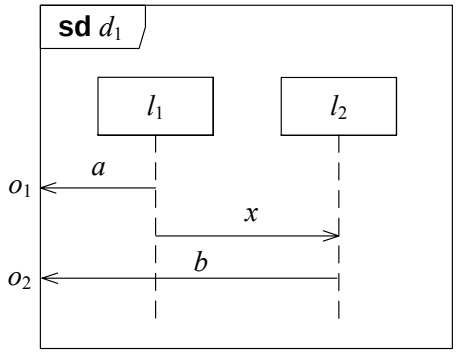


Figure 13.9: Specification d_1

(which would be the case with e.g. $d \rightsquigarrow_r d_3$ if d_3 did not have the assert operator), and the examples presented in this section therefore do not exhaust all interesting cases.

13.4 Asynchronous testing

In the test generation and test execution defined in the preceeding sections, the observation point is at the lifelines of the specifications. As a result, the test execution is defined as a form of synchronous communication between the test and the specification being tested (because messages between test and specification are assumed to be instantanuous).

Assume the specification d_1 of figure 13.9. Because the test generation algorithm operates directly on the interface of the specification, it will consider observation of $\langle !b, !a \rangle$ as inconclusive because $!a$ must happen before $!b$.

However, if we were to assume that there is one communication link from l_1 to the point of observation and one from l_2 to the point of observation, it would mean that the message b may overtake message a so that we observe $!b$ before $!a$. Because we mainly deal with asynchronous communication, this must be considered an assumption we would want to make in certain situations.

A solution to this is proposed by [181, 186]. The proposed solution is to still define the test generation and execution with the observation point direcly at the interface, but to model the communication link explicitely as a process composed in parallel with the processes we generate tests from and execute tests against.

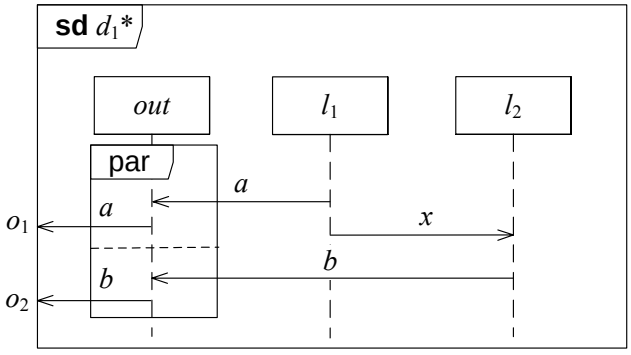


Figure 13.10: Specification d_1^*

We can obain this by including a new lifeline *out* in d_1 which receives and retransmits all messages bound for the frame. The resulting specification, called d_1^* is shown in figure 13.10. This modified specification is then used in test generation or test execution.

When generating tests from d_1^* both $\langle !a, !b \rangle$ and $\langle !b, !a \rangle$ will be treated as observations of positive behavior, which is the desired situation. We may, of course, do the same thing for receive events.

Further, we can state this as a general procedure for every given communication model. Given a specification d and the communication model with a single communication link for each message, we obtain a specification d^* in the following way:

1. If lifeline names *in* or *out* are used in d, i.e. $in \in ll.d$ or $out \in ll.d$, rename these lifelines with names not in $ll.d$.

2. Add the lifelines *in* and *out* to the specification.

3. Let *in* have a par operator with one operand for each event in E_I^d.

4. Let *out* have a par operator with one operand for each event in E_O^d.

5. For each $e \in E_I^d$ let the message $m.e$ from the frame to $l.e$ instead go from the frame to *in* and then further from *in* to $l.e$ is such a way that this happens within its designated par operand, and so that all ordering, choices, loops, etc. on $l.e$ are preserved.

6. For each $e \in E_O^d$ let the message $m.e$ from $l.e$ to the frame instead go from $l.e$ to *out* and then further from *out* to the frame in such a way that this happens within its designated par operand, and so that all ordering, choices, loops, etc. on $l.e$ are preserved.

In the test generation and execution we defined in sections 13.2 and 13.3 we implicitly assumed that the specification that a test is generated from, and the specification the test is executed against, have the same names on their external lifelines. As a nice byproduct of the procedure for producing "starred" specifications, we do not have to worry about lifeline names when generating and executing tests as long as we maintain a convention for naming these inserted lifelines in the "starred" specifications.

Chapter 14

The 'Escalator' tool

Escalator is a prototype tool developed to demonstrate the use of our operational semantics and the analysis mechanisms we have defined over the semantics, in particular the test generation and test execution defined in chapter 13. The tool makes direct use of the Maude implementation of the operational semantics and the testing theory, but also provides a user interface with a graphical sequence diagram editor to ease the use sequence diagram specifications.

One of the strengths of sequence diagrams as a specification language is their graphical and diagram based appearance, which makes them easy to comprehend and work with. The Maude implementation of the operational semantics does not have this advantage because it presupposes a textual representation of the diagrams. However, using the Maude interpreter for execution of diagrams via the operational semantics has the advantage that Maude is a formal language and the that Maude interpreter is developed to preserve the semantics of Maude. With an implementation of the operational semantics and the test generation algorithm in, e.g., a programming language like Java or C, we would face the risk of unintentionally changing the semantics. By using Maude, this risk of compromising the formality of our approach is reduced.

With Escalator we combine these two considerations by having a graphical editor as frontend and the Maude interpreter as backend. This way we get both the benefits of a graphical diagram based specification language and the benefits of a formal foundation.

In section 14.1 we present the main use cases of Escalator, while in section 14.2 we describe its architecture. Section 14.3 presents the user interface and its intended use. In chapter 15 we conduct a case study with the help of Escalator.

14.1 Use cases of Escalator

The main functionality of Escalator is shown as use cases in the UML use case diagram of figure 14.1.[1] Below we give a brief, informal characterization of each of the use cases:

Draw sequence diagram. The user of Escalator can make a sequence diagram in the graphical sequence diagram editor.

Configure semantics. The operational semantics allows various instantiations, e.g., with respect to the communication model. Making choices with respect to these

[1] The use cases are represented as ovals. An arrow labed «include» going from use case X to use case Y indicates that conducting X involves conducting Y.

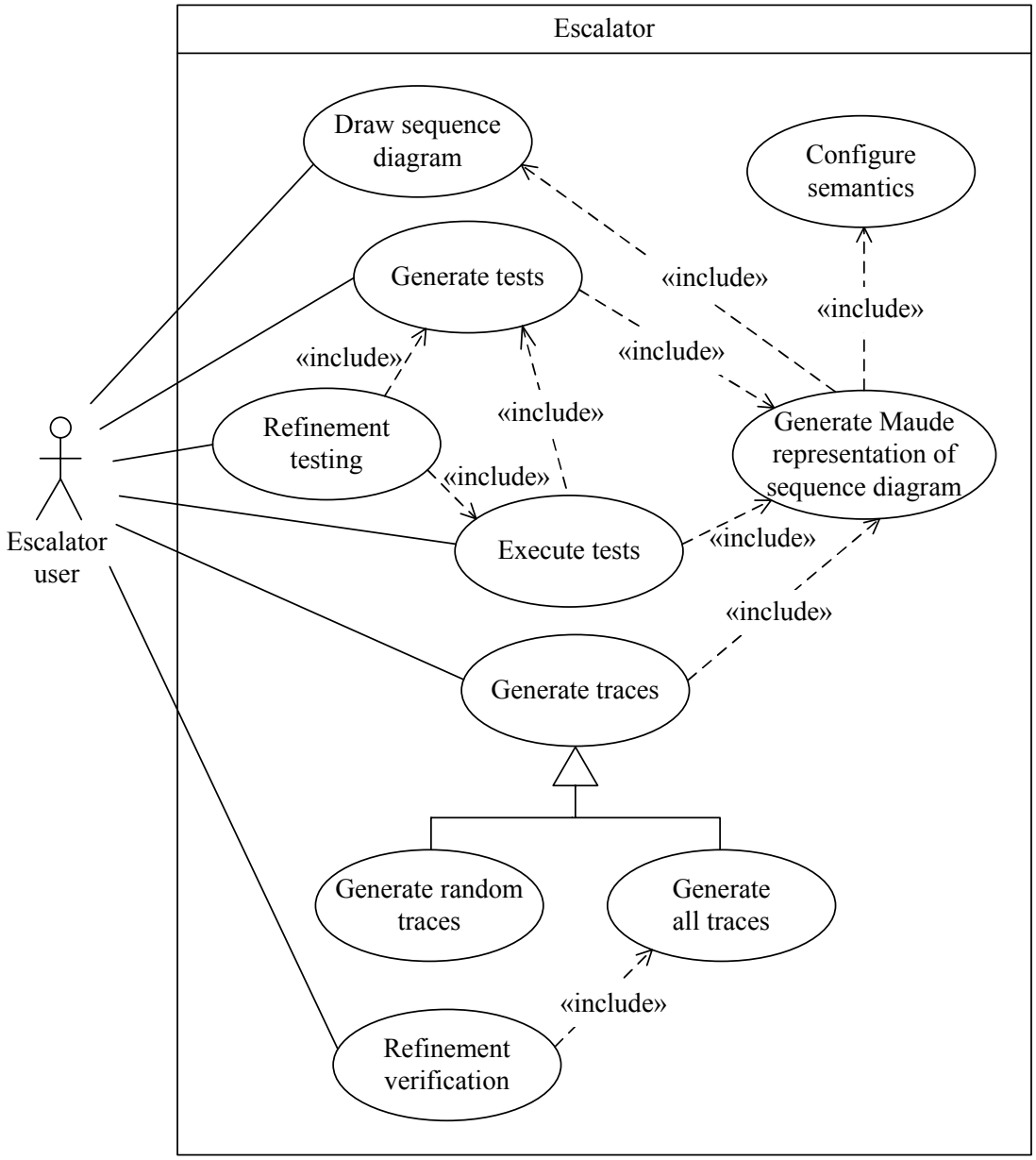


Figure 14.1: Use cases for Escalator

possible variations is what we call configuration of the semantics.

Generate Maude representation of sequence diagram. For sequence diagrams to be executed by the Maude interpreter we need to have Maude representations of the sequence diagrams that have been made in the graphical editor. Obviously this requires that diagrams actually have been made.

Generate tests. Escalator can generate tests from sequence diagrams applying the test generation algorithm defined in section 13.2. This presupposes a Maude representation of the diagram from which tests are generated.

Execute tests. Tests can be executed against sequence diagrams using the method described in section 13.3. This requires that tests have been generated and that

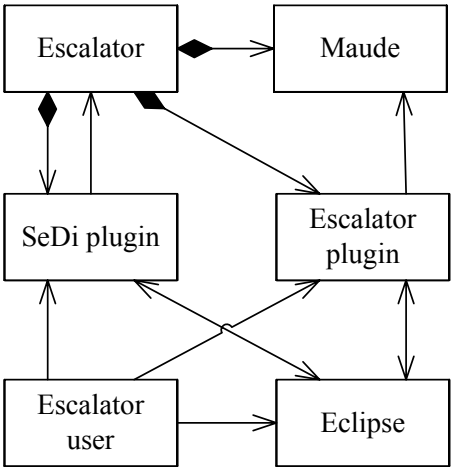


Figure 14.2: Class diagram

a Maude representation of the diagram against which the tests are executed is generated.

Refinement testing. As explained in chapter 13, test execution and test generation can togehter be used as a method for analysing the correctness of a refinement. This we refer to as refinement testing.

Generate traces. Escalator can generate traces by executing a sequence diagram with the Maude interpreter. This requires that a Maude generation of the diagram is made.

Generate random traces. A specialization of test generation is the possibility of generating a random sample of all the traces that a diagram characterizes using the method described in sections 9.1 and 10.2.3.1.

Generate all traces. A specialization of test generation is the possibility of generating all traces characterized by a sequence diagram using the method described in sections 9.2 and 10.2.3.2.

Refinement verification. By comparison of the traces generated from two sequence diagrams, based on the definitions of refinement given by STAIRS, Escalator can analyse whether a diagram is a correct refinement of another diagram. This requires that all traces of the two diagrams have been generated, and is based on the method for refinement verification described in sections 9.2 and 10.2.3.2.

14.2 Architecture of Escalator

Escalator is developed as an Eclipse plugin [43]. We do not ourselves implement an editor, but integrate a simple, open source sequence diagram editor called SeDi [107], itself an Eclipse plugin, into our tool. Maude is integrated into the tool by implementation of a kind of wrapper that can start the Maude interpreter as a separate process. This wrapper provides a Java interface to facilitate communication with the command-line interface of the Maude interpreter.

One can then perceive Escalator as consisting of three main component:

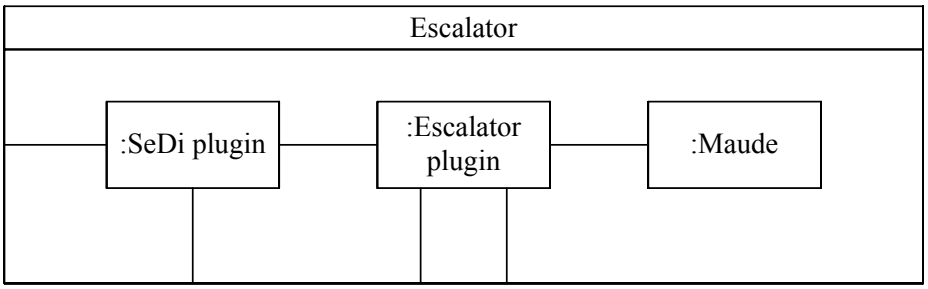


Figure 14.3: Parts diagram

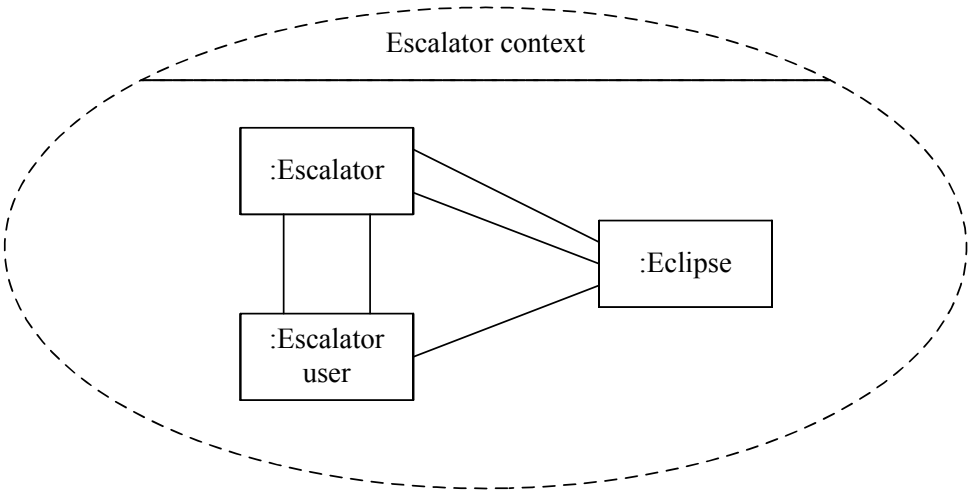


Figure 14.4: Collaboration diagram

- The *SeDi plugin* providing the sequence diagram editor.
- The*Maude wrapper* enabling execution of and communication with the Maude interpreter.
- The *Escalator plugin* providing the necessary functionality for analyzing sequence diagrams with the operational semantics and the testing mechanisms.

The Escalator plugin serves two main purposes:

1. Generating Maude representations of sequence diagrams specified with the editor.
2. Providing user interfaces for specifying analyses of the diagrams, including configuration of the operational semantics, and for presentation of analysis results.

While specification of sequence diagrams is done by means of the SeDi editor and the analysis is carried out by using the Maude interpreter, the Escalator plugin can be seen as lying between these to other components.

A class diagram giving a high-level view of Escalator is shown in figure 14.2.[2] Escalator is represented as a class that contains the tree classes 'Escalator plugin',

[2]Classes are represented as boxes. The arrows between the boxes represents associations or relations between the classes, and the direction of the arrows represent navigability or visibility. A black diamond indicates that the class connected to the diamond is composed of the class the arrow points to.

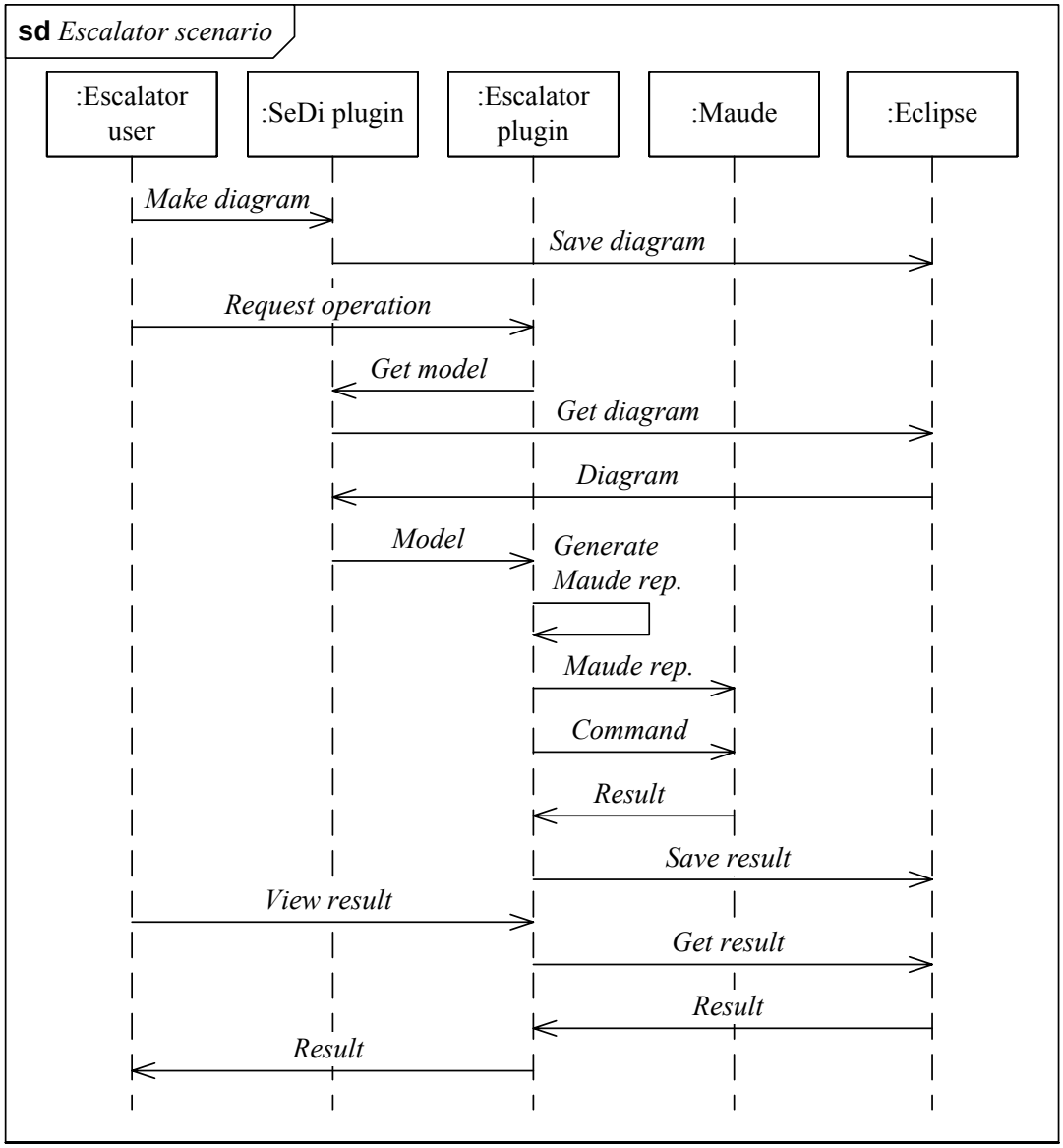


Figure 14.5: Generic user scenario

'SeDi plugin' and 'Maude'. We see that both the SeDi plugin and Maude is visible to the Escalator plugin, but not the other way. This reflects the fact that SeDi and Maude in a way stand alone, while the Escalator plugin integrates functionality from these two components.

Escalator, through the SeDi plugin and the Escalator plugin, also gets functionality from Eclipse itself, such as file and window management. For this reason Eclipse is related to the SeDi plugin and the Escalator plugin, while Maude stands by itself and is not related to Eclipse in the same way. The user communicates with Escalator through the functionality provided by the Escalator plugin, the SeDi plugin and Eclipse, and is therefore related to these three classes. The user does not, however, communicate directly with Maude and is related directly to the Maude class.

In figure 14.3 a parts diagram for Escalator is provided.[3] We have communication

[3] A parts diagram shows the structre of a class. The boxes represents parts (objects, roles) contained by the class. The lines between the parts represents connectors (communication links).

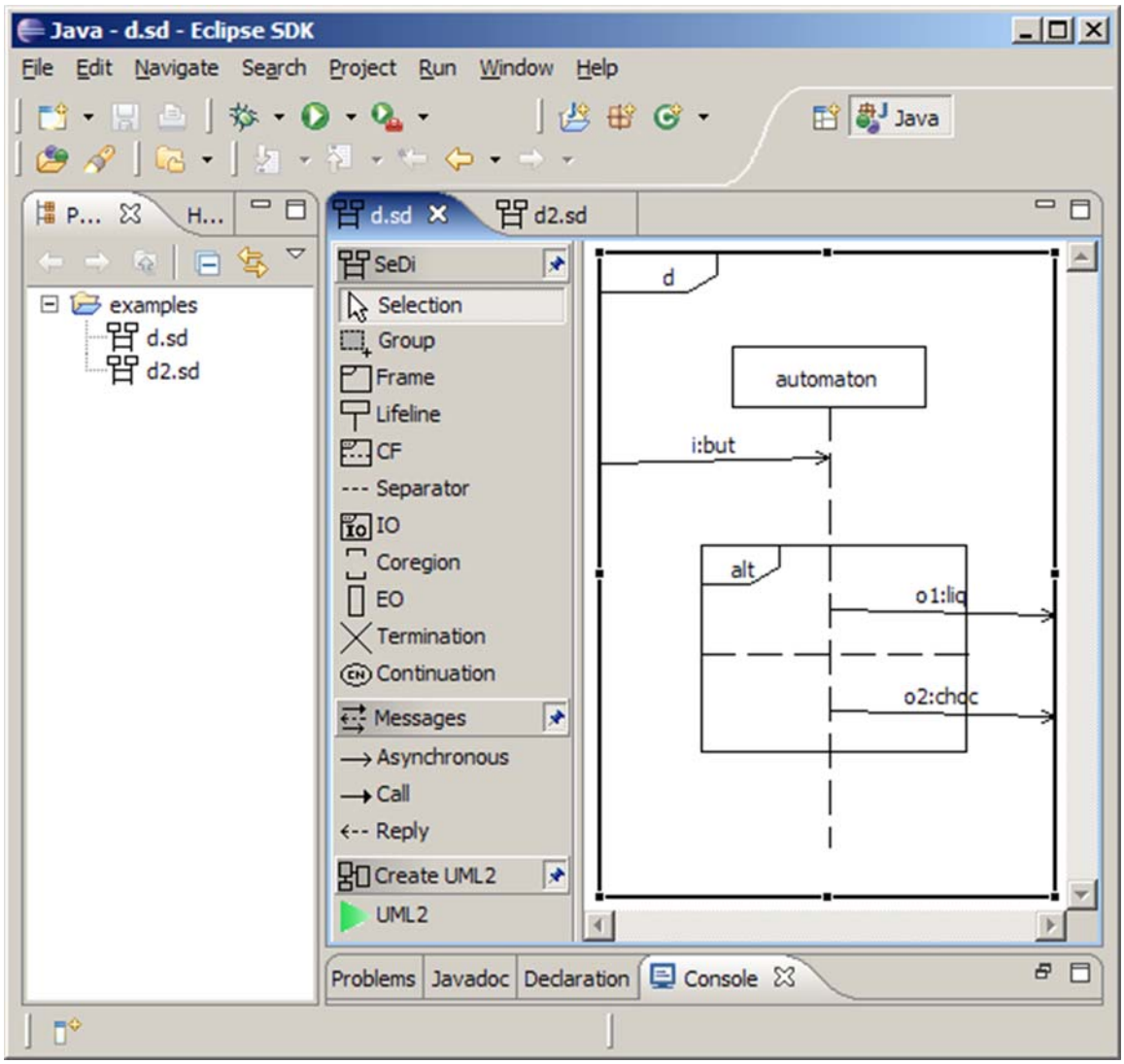


Figure 14.6: The SeDi editor

links between the SeDi plugin and Escalator plugin and between the Escalator plugin and Maude. Through the link to the SeDi plugin, the Escalator plugin obtains diagrams (represented in an internal model), and through the link to Maude it sends commands and receives results from the executions of the Maude interpreter.

Both the SeDi plugin and the Escalator plugin have two external communication links. For each of the components, one is a link for communication with Eclipse and one is a link for communication with the user. Figure 14.4 shows the context of Escalator in a collaboration diagram[4], where we see the two communication links from Escalator to the user and the two links from Escalator to Eclipse. In addition there is an obvious link between the the user and Eclipse since the user is also able to communicate with Eclipse independent of the functionality provided by Escalator.

[4]The oval represents a collaboration, which is a structural relationship between parts (objects, roles). The lines between parts represent connectors (comminication links). The difference from a parts diagram is that the parts of an collaboration exist independently and prior to the creation of an instance of the collaboration, while the parts of a class is created togehter with an instance of the class containing them.

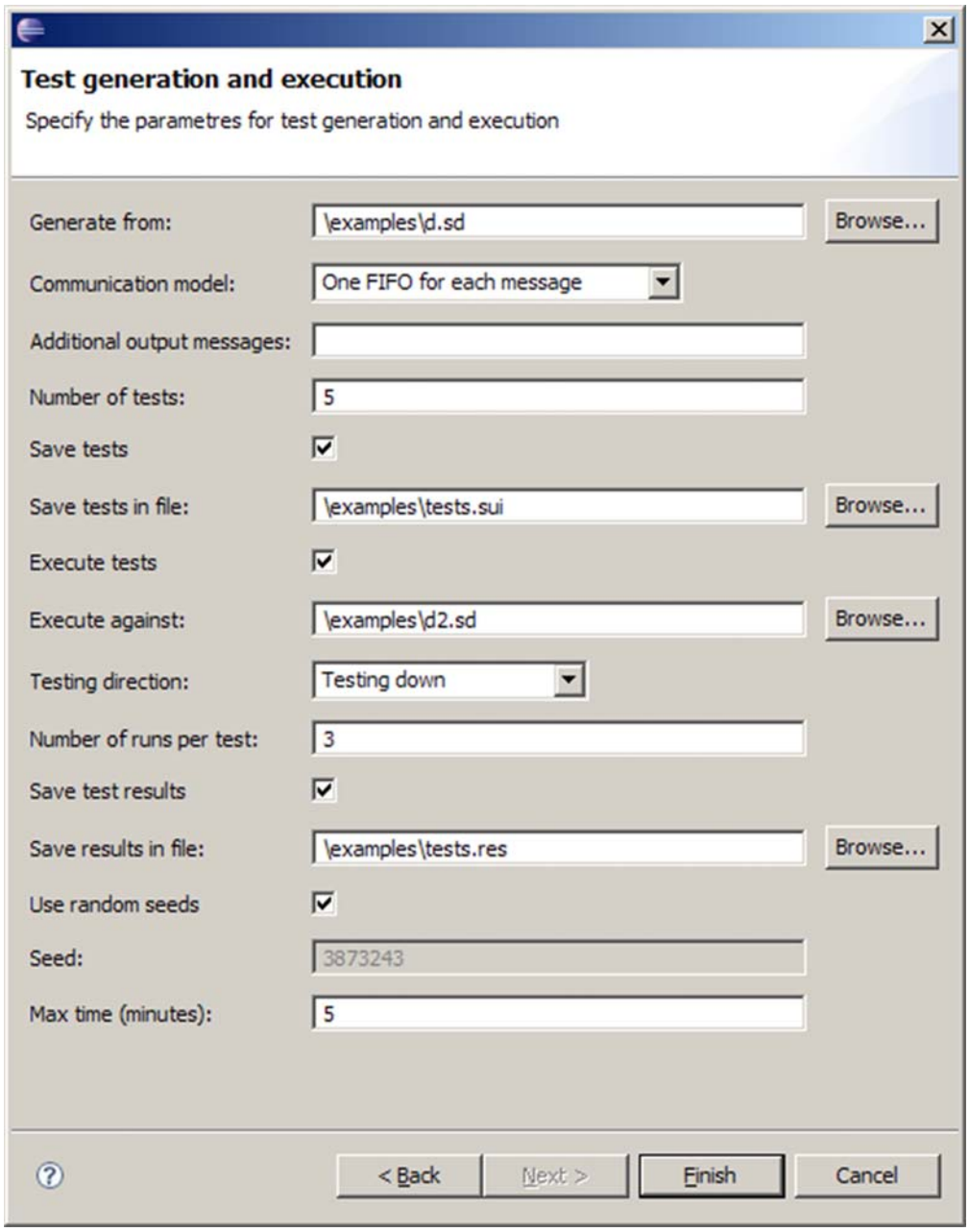


Figure 14.7: Testing wizard

14.3 Use of Escalator

Most of the operations of Escalator follows a specific pattern: The user draws a diagram in SeDi, saves it in the Eclipse file system and then requests an operation on the diagram from Escalator. The Escalator plugin then retrieves the diagram in the form of SeDi's internal model and generates a textual Maude representation of the diagram. This representation together with the appropriate commands are sent to Maude. When the results are received from Maude, they are stored in Eclipse's file system. The user can request to view the results, in which case they are retrieved from Eclipse and presented to the user is a suitable fashion. This generic scenario is presented in the sequence diagram of figure 14.5. With some variations, this scenario applies to most of the central use cases of figure 14.1.

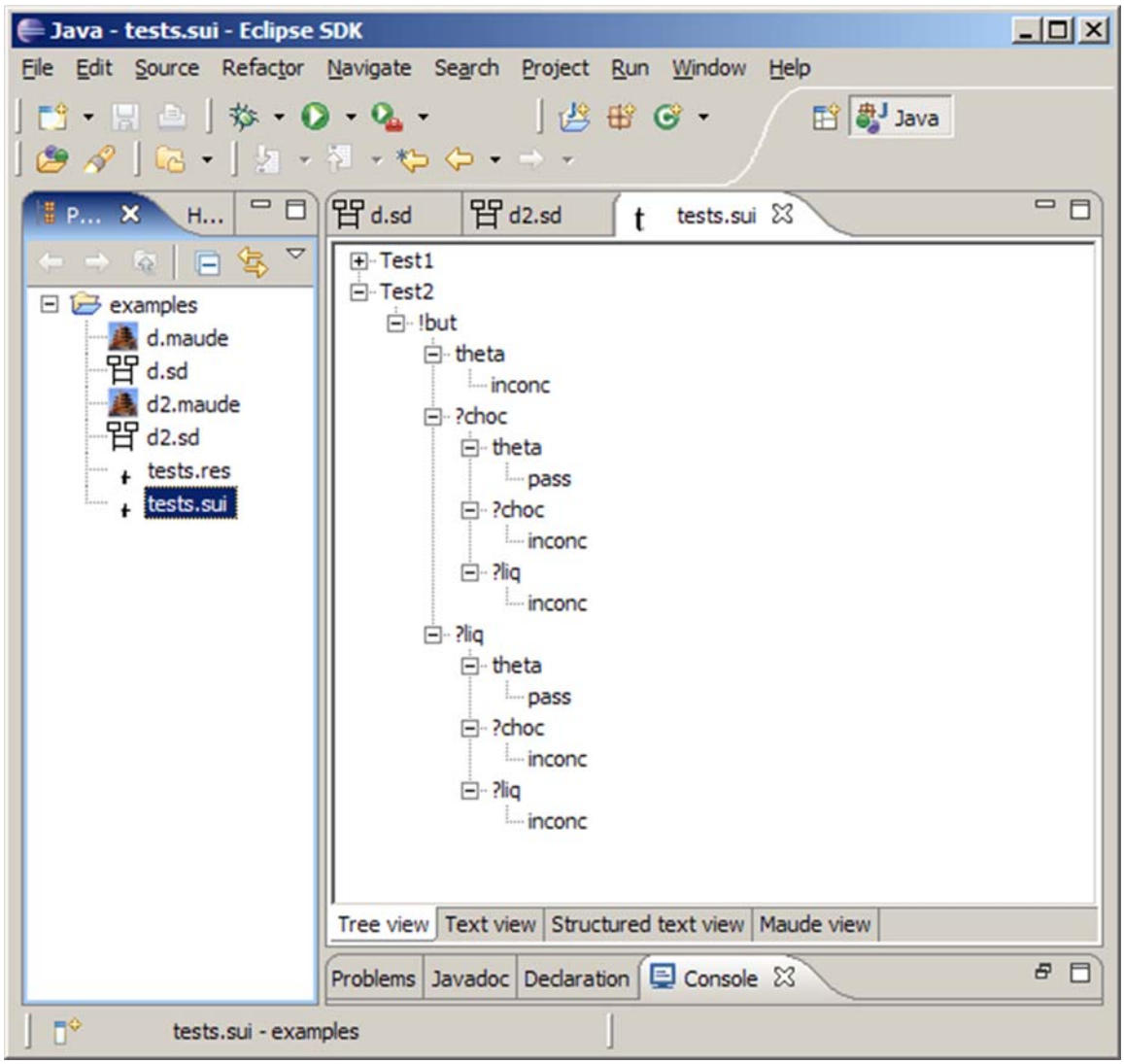


Figure 14.8: Test viewer

In the following we show how such a scenario will manifest itself for the user with respect to combined test generation and test execution. The first thing the user does is to draw one or more diagrams with the SeDi editor. In figure 14.6 we see the editor in which the diagram d from figure 13.3 is drawn. On the left we see the file manager, where the diagram is saved as `d.sd`.

The user tells Escalator to perform operations on the diagrams with the help of wizards. Figure 14.7 shows the wizard for combined test generation and test execution. In this example the user tells Escalator to generate tests from the diagram d, and to execute them against a diagram $d2$. The user has specified that five tests should be generated from d and that each test should be run three times againt $d2$, both applying the "testing down" option. The process should use random seeds, and the process should be terminated if it has not completed within five minutes. Further it is specified that the communication model with one FIFO buffer for each message should be applied, that the resulting tests should be saved in a file `tests.sui` and that the results of the test exectuion should be saved in a file `tests.res`. Clicking the `Finish` button causes the test generation and execution to be carried out as specified. For all

the use cases in figure 14.1 involving the user, similar wizards are provided.

When the test generation and test execution have completed, the tests and test execution results are saved in the files `tests.sui` and `tests.res`, respectively. For each kind of results (tests, test execution results, traces) Escalator provides a viewer for the user to review the results. In figure 14.8 we show the viewer for test suites (sets of tests) presenting the contents of `tests.sui`. Even though the user asked for five tests the suite only contains two, because the same tests where generated several times and Escalator does not keep duplicates of tests in a suite. *Test*2 in the suite can be recognized as the test T of figure 13.6 (with the exception that in the generation of T, for presentation purposes, the test generation algorithm was terminated one step earlier than in generation of *Test*2). In the file manager on the left we can also observe that in the process of generating and executing tests, Maude representations of the diagrams have been generated and stored in the files `d.maude` and `d2.maude`.

Chapter 15

Case study

In this chapter we apply the Escalator tool on a case study. The main hypothesis is that the tool is able to handle a medium sized case, and the main functions of the tool, i.e. the refinement verification and the refinement testing, provide the expected results.

The chosen case is a STAIRS specification of a system called the "BuddySync system", documented in [154]. BuddySync is an imaginary and generic system for management of services, for example ordering of taxis by SMS. Subscribers to BuddySync may request and offer services, and the system will match the requests with the offers and notify the users when matches are found. An elaborated description of the system is found in section 15.1 below.

The reasons for choosing BuddySync for the case study are are follows:

- The case is developed for STAIRS, i.e. the case consists of a number of STAIRS specifications and therefore applies STAIRS features such as mandatory choice.
- The case specifies the system at different levels of abstraction and hence is suitable for applying the refinement verification and the refinement testing functionality of Escalator.
- The original BuddySync specification was made by other people than the author of this thesis. Therefore the specification of the case is not made with the Escalator tool in mind.
- The size is suitable, i.e. the case is bigger then being a mere toy example, but still manageable within a reasonable time frame.

Our case study consists of two parts. In the first part we use the refinement verification functionality of Escalator on the BuddySync specification, and in the second part we apply the refinement testing functionality on the specification. As is explained in detail later, we use two different versions of the specification in the two cases, none of which is identical to the original specification in [154]. In section 15.2 the refinement verification part of the case is presented and in section 15.3 we present the refinement testing part of the case.

15.1 The BuddySync system

The BuddySync case as defined in [154] is a STAIRS specification of the BuddySync system. The system provides six usages specified by six diagrams *RequestService*,

Level	Lifelines
0	*Requester*, *Provider*, *User*, *System*
1	*Requester*, *Provider*, *User*, *System*
2	*Requester*, *Provider*, *User*, *Control*, *MessageBoard*
3	*Requester*, *Provider*, *User*, *Control*, *MessageBoard*
4	*Requester*, *Provider*, *User*, *Control*, *MessageBoard*, *MemberList*

Table 15.1: Lifelines at the different abstraction levels

OfferService, *RemoveRequest*, *RemoveOffer*, *SubscribeService* and *UnsubscribeService*. The full specification is a diagram *Overview* that combines these six diagrams using xalt:

$$\begin{aligned} \mathit{Overview} \;=\; & \mathit{RequestService}\ \mathsf{xalt} \\ & \mathit{OfferService}\ \mathsf{xalt} \\ & \mathit{RemoveRequest}\ \mathsf{xalt} \\ & \mathit{RemoveOffer}\ \mathsf{xalt} \\ & \mathit{SubscribeService}\ \mathsf{xalt} \\ & \mathit{UnsubscribeService} \end{aligned}$$

Through refinement steps, variants of these diagrams exist at several levels of abstraction. We have identified five such levels, which we refer to as *level 0* through *level 4*. In order to distinguish between the variants of the diagrams at the different levels we give them names $\mathit{RequestService}_0$, $\mathit{RequestService}_1$, etc. in such a way that $\mathit{RequestService}_0$ is the diagram *RequestService* at level 0, $\mathit{RequestService}_1$ the diagram *RequestService* at level 1, and so forth.

At the most abstract levels of the specification of [154], the diagrams have two lifelines *:System* and *:User*.[1] In lower levels *:System* is decomposed into *:Control*, *:MessageBoard* and *:MemberList*. It should also be noted that the specification operates with three instances of *:User*, namely *requester:User*, *provider:User* and *user:User*. For simplicity we treat these instances as three different lifelines *Requester*, *Provider* and *User*. We also drop the colon in the lifeline names, as this no longer has any function when we treat lifeline instances as separate lifelines. The lifelines present at the different levels of abstraction is presented in table 15.1.

Only the diagrams *RequestService* and *OfferService* are updated at every level. Table 15.2 shows which diagrams we have at the different levels. The placement of the updated versions of the four remaining diagrams may seem somewhat arbitrary,

[1]The colon in front of the names denote that these lifelines are classes from which instances can be made.

Level	Diagrams
0	$\mathit{RequestService}_0$, $\mathit{OfferService}_0$, $\mathit{RemoveRequest}_0$, $\mathit{RemoveOffer}_0$, $\mathit{SubscribeService}_0$, $\mathit{UnsubscribeService}_0$
1	$\mathit{RequestService}_1$, $\mathit{OfferService}_1$
2	$\mathit{RequestService}_2$, $\mathit{OfferService}_2$, $\mathit{RemoveRequest}_2$, $\mathit{RemoveOffer}_2$
3	$\mathit{RequestService}_3$, $\mathit{OfferService}_3$
4	$\mathit{RequestService}_4$, $\mathit{OfferService}_4$, $\mathit{SubscribeService}_4$, $\mathit{UnsubscribeService}_4$

Table 15.2: Diagrams at the different abstraction levels

but is based on the lifelines they contain. *RemoveRequest*$_2$ and *RemoveOffer*$_2$ are placed at level 2 because they contain the lifelines *Control* and *MessageBoard*, while *SubscribeService*$_4$ and *UnsubscribeService*$_4$ are placed at level 4 because they contain the lifeline *MemberList*, which is introduced at that level.

Because we use different versions of the BuddySync specification in the refinement verification and the refinement testing parts of the case, we do not give the detailed presentation of the diagrams of the specification here. The version used in the refinement verification part is presented in section 15.2.2, and the version used in refinement testing part is presented in section 15.3.2. There we also account for the differences between the two versions.

15.2 Refinement verification

In this part of the study we applied the refinement verification functionality of the Escalator tool on the BuddySync specification, i.e. we used the tool to investigate whether the more concrete levels of the specification are correct refinements of the more abstract levels. In section 15.2.1 we provide assumptions and success criteria for this part of the case. Section 15.2.2 presents the BuddySync specification as we used it in the refinement verification, and in section 15.2.3 we present the setup of the case. In section 15.2.4 we provide the results of the refinement verification and in section 15.2.5 we discuss the results.

15.2.1 Assumptions and success criteria

The success criterion for the refinement verification part of the case study is formulated as follows:

> *The refinement verification functionality of the tool provides the expected results when applied to the BuddySync specification.*

The BuddySync case is originally a case study in refinement, and this should indicate that the expected results are that level 1 of the specification is a correct refinement of level 0, that level 2 is a correct refinement of level 1 and so forth. However, because we make some changes to the original specification (these are explained below), this might not necessarily be the case. Therefore, it is part of the success criterion that any unexpected results are explained by identifying incorrect refinements in a separate analysis of the specifications.

15.2.2 The BuddySync specification for the refinement verification

In this section we present the specification of the BuddySync system as we used it in the refinement verification. Some simplifications have been made with respect to the diagrams provided in [154], due to the use of STAIRS features not supported by the refinement verification functionality of Escalator. These simplifications are:

- We have removed the assert operator where it is used in the original specification. This also means that we have alts some places where assert was used to refine alts.

- Parameters of diagrams and messages have been removed.
- *User*, *Requester* and *Provider* are treated as independent lifelines, as explained in section 15.1.
- Actions, constraints and guards are removed. Some places we use a combination of alt and refuse to simulate constraints.

We do not, however, find the application of these features fundamental for the case.

The specification follows in the sub-sections below, together with some explanatory text. As the specification is "borrowed", we do not give much rationale for the way the diagrams look or are made. The purpose of the presentation is merely that of documenting the case. The presentation is structured by level.

15.2.2.1 Level 0

Level 0 gives the most abstract specification of the functionality of the BuddySync system. The specification consists of the six diagrams of figures 15.1–15.6, which show the interaction between users and the system. $RequestService_0$ shows how a user of the system in the role of requester can request a service form the system and get the response either that somebody has agreed to perform the service or that the request is registered. $OfferService_0$ is symmetric for the provider role; the user offers a service to the system and gets one of two possible responses: an instruction to perform the service or a message that the offer has been registered.

$RemoveRequest_0$ and $RemoveOffer_0$ allow requesters and providers to remove the requests or offers they have communicated to the system. In response they receive a confirmation that the request or offer has been removed.

Finally, $SubscribeService_0$ and $UnsubscribeService_0$ allow users to subscribe and unsubscribe to services. A user that subscribes to a service will get the answer that he/she has been added as provider or requester, and a user that unsubscribes will get the answer that he/she has been unsubscribed from the service.

It should be noted that in [154], messages are parameterized with variables to identify the service that is invocated, and other information to the system. Both at level 0 and at the remaining levels, we have suppressed this and left it unspecified.

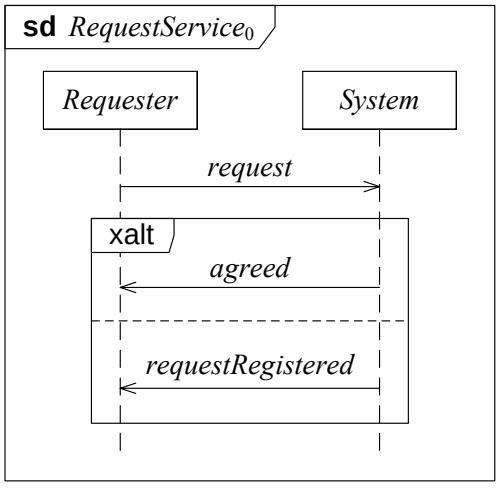


Figure 15.1: $RequestService_0$

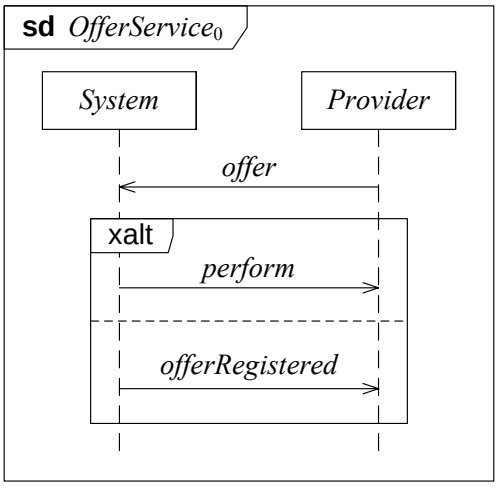


Figure 15.2: $OfferService_0$

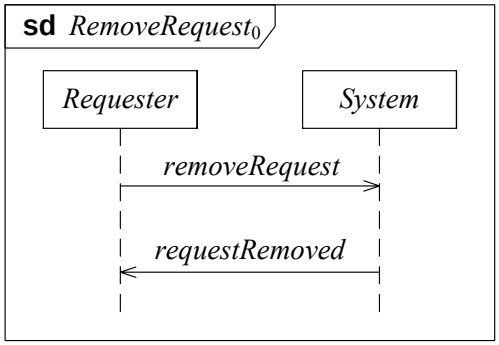


Figure 15.3: *RemoveRequest*$_0$

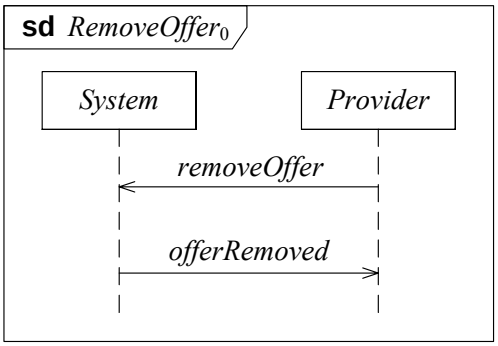


Figure 15.4: *RemoveOffer*$_0$

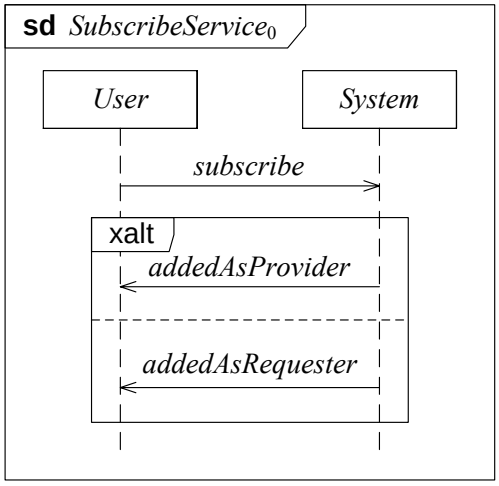


Figure 15.5: *SubscribeService*$_0$

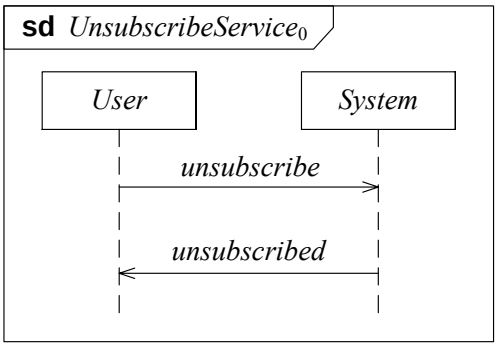


Figure 15.6: *UnsubscribeService*$_0$

15.2.2.2 Level 1

At level 1, the diagrams *RequestService*$_0$ and *OfferService*$_0$ are refined into *RequestService*$_1$ and *OfferService*$_1$. These refined diagrams are shown in figures 15.7 and 15.8. In these diagrams, the sequence diagram feature *interaction occurrence* is applied, represented in the diagrams as boxes with the label ref attached to them. *RequestService*$_1$ refers to *Match*$_1$ (figure 15.9) and *SaveRequest*$_1$ (figure 15.10), and *OfferService*$_1$ to *Match*$_1$ and *SaveOffer*$_1$ (figure 15.11).

We treat these references simply as syntactic substitutions of the references by the diagrams to which they refer. This means that in the textual syntax we do for example have:

$$\begin{aligned}
Match_1 &= !perform\ \mathsf{seq}\ ?perform\ \mathsf{seq}\ !agreed\ \mathsf{seq}\ ?agreed \\
SaveOffer_1 &= !offerRegistered\ \mathsf{seq}\ ?offerRegistered \\
OfferService_1 &= !offer\ \mathsf{seq}\ ?offer\ \mathsf{seq} \\
&\quad (\mathsf{ref}(Match_1)\ \mathsf{xalt}\ \mathsf{ref}(SaveOffer_1)) \\
&= !offer\ \mathsf{seq}\ ?offer\ \mathsf{seq} \\
&\quad (!perform\ \mathsf{seq}\ ?perform\ \mathsf{seq}\ !agreed\ \mathsf{seq}\ ?agreed\ \mathsf{xalt} \\
&\quad\ !offerRegistered\ \mathsf{seq}\ ?offerRegistered)
\end{aligned}$$

Since this is purely syntactic, it has no effect on the semantics of the diagrams.

The main difference compared to level 0 is that both *Requester* and *Provider* are now present in the diagrams. Both messages *perform* and *agreed* are sent in both

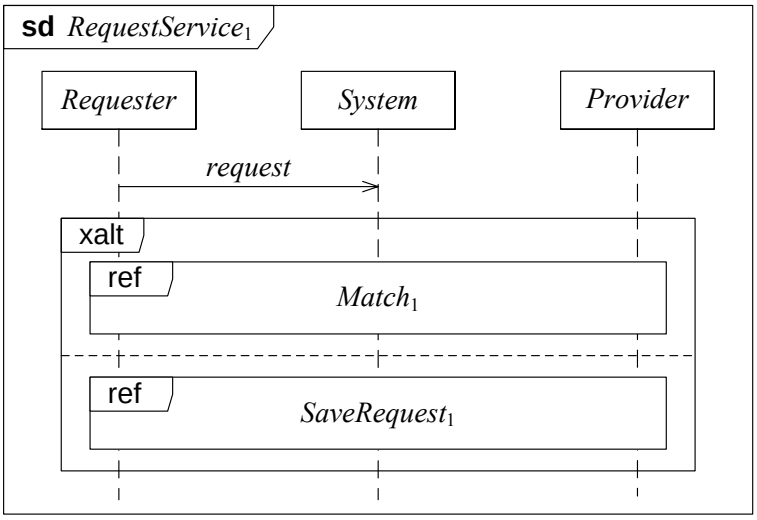


Figure 15.7: $RequestService_1$

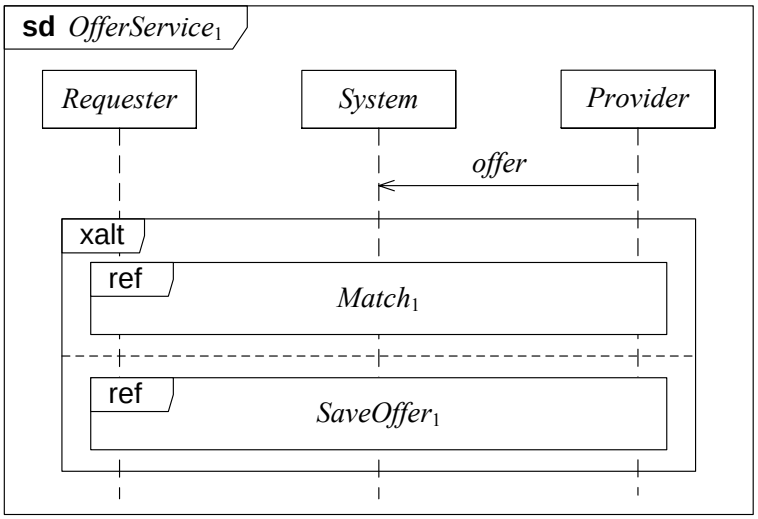


Figure 15.8: $OfferService_1$

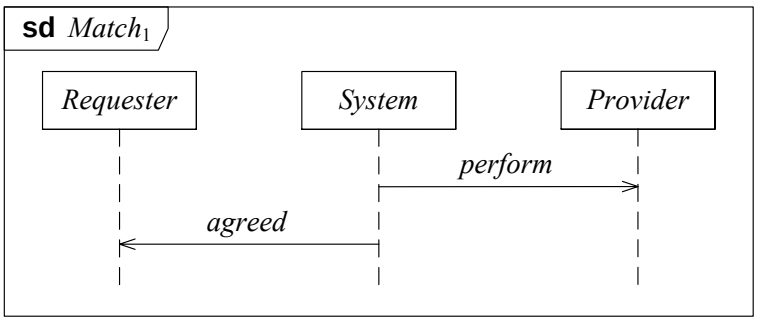


Figure 15.9: $Match_1$

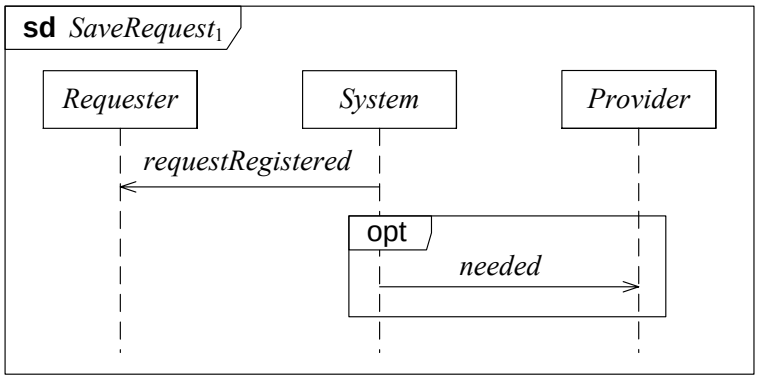


Figure 15.10: *SaveRequest*$_1$

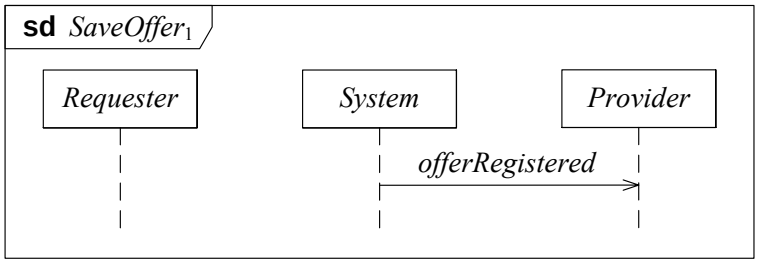


Figure 15.11: *SaveOffer*$_1$

RequestService$_1$ and *OfferService*$_1$ (through the reference to *Match*$_1$) to *Provider* and *Requester*, respectively.

In *SaveOffer*$_1$, merely a notification to the provider that the offer is registered is sent, so this represent no change with respect to the second xalt operand of *OfferService*$_0$. However, in *SaveRequest*$_1$ an optional message *needed* to *Provider* is specified, and hence represents a refinement with respect to the second xalt operand of *RequestService*$_0$.

15.2.2.3 Level 2

At level 2, the lifeline *System* is decomposed into the lifelines *Control* and *MessageBoard*. In accordance with this change, this level has the diagrams *RequestService*$_2$, *OfferService*$_2$, *RemoveRequest*$_2$ and *RemoveOffer*$_2$ as refinements of *RequestService*$_1$, *OfferService*$_1$, *RemoveRequest*$_0$ and *RemoveOffer*$_0$.

After the decomposition of *System* into *Control* and *MessageBoard*, *Control* is the interface towards the users of the system, while *MessageBoard* is an internal lifeline that keeps track of pending offers and requests. In *RequestService*$_2$ (figure 15.12) and *OfferService*$_2$ (figure 15.13) this has the effect that *Control*, upon reception of a *request* or *offer* message, sends *findOffer* or *findRequest* to *MessageBoard* and receives a message *matchFound* or *noMatch* as reply.

Match$_1$ in *RequestService*$_1$ and *OfferService*$_1$ is replaced by *MatchRequest*$_2$ (figure 15.14) and *MatchOffer*$_2$ (figure 15.15), respectively, in *RequestService*$_2$ and *OfferService*$_2$. Both sends the *matchFound* reply before referring to *Match*$_2$ (figure 15.16). *Match*$_2$ is identical to *Match*$_1$ except that the lifeline *System* is replaced by *Control*.

The diagrams *SaveRequest*$_2$ (figure 15.17) and *SaveOffer*$_2$ (figure 15.18), referred to by *RequestService*$_2$ and *OfferService*$_2$, are identical to *SaveRequest*$_1$ and *SaveOffer*$_1$

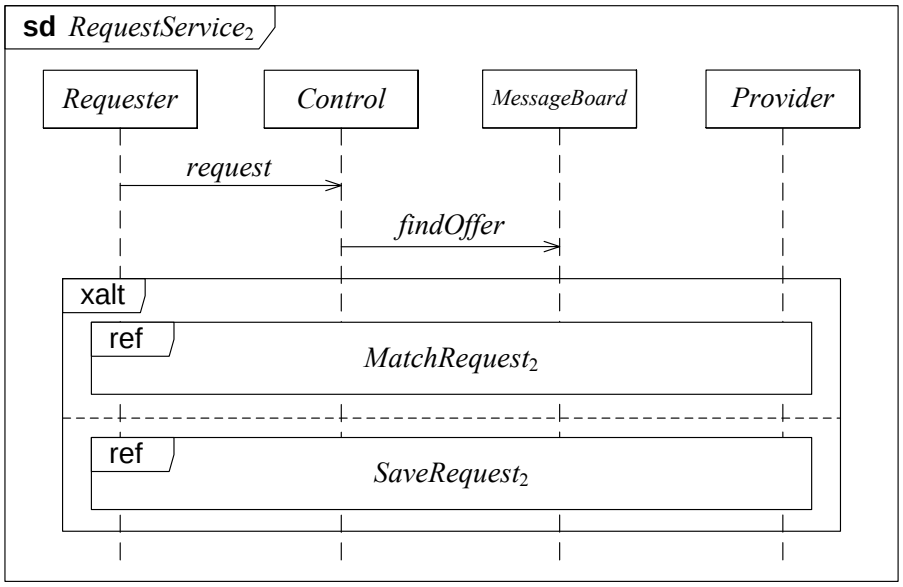


Figure 15.12: *RequestService*$_2$

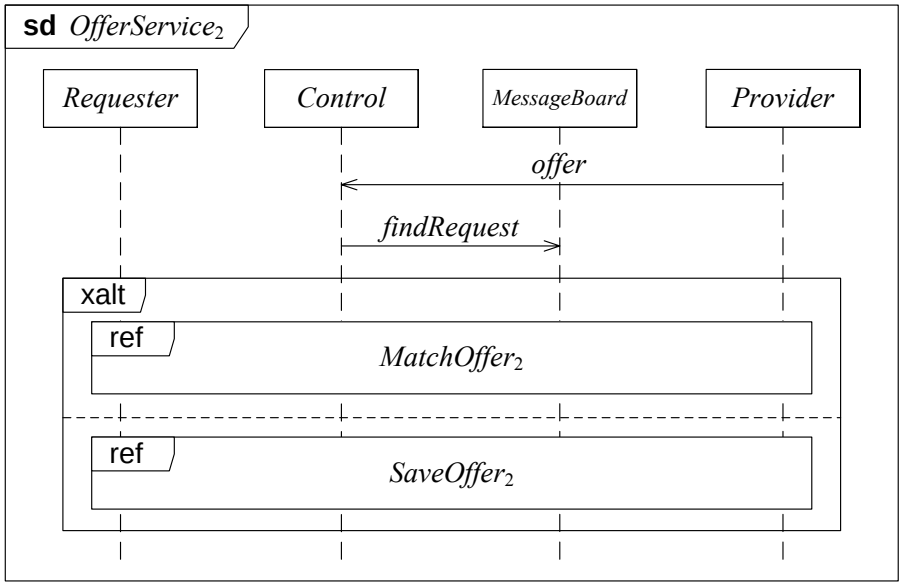


Figure 15.13: *OfferService*$_2$

with the exceptions that the *noMatch* message is sent from *MessageBoard* to *Control* and that *Control*, instead of *System*, sends messages to the users.

The alt constructs in these diagrams where the messages *matchFound* and *noMatch* are specified as both positive and negative might seem strange. The rationale for this is to express that both messages may be both positive and negative; *matchFound* is positive if a match *is* found and negative if a match is *not* found. Similarly, *noMatch* is positive if a match is *not* found, but negative if a match *is* found. In [154] this is made explicit by use of constraints, but since we de not apply constraints in this case study we have chosen this way of representing it.

The updated diagrams *RemoveRequest*$_2$ (figure 15.19) and *RemoveOffer*$_2$ (figure 15.20) are similar to *RemoveRequest*$_0$ and *RemoveOffer*$_0$. The difference is that *Control*, instead of *System*, communicates with the user of the system, and that *Control*

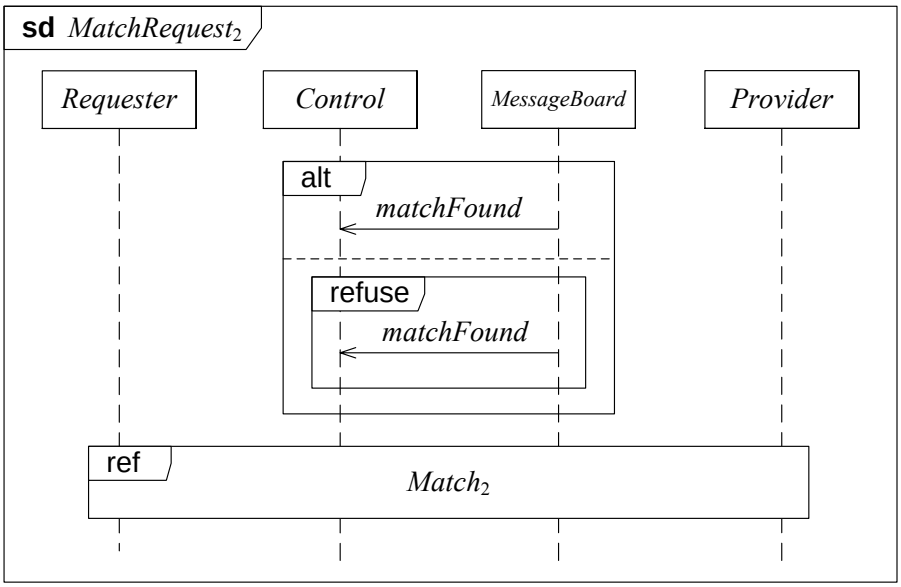


Figure 15.14: $MatchRequest_2$

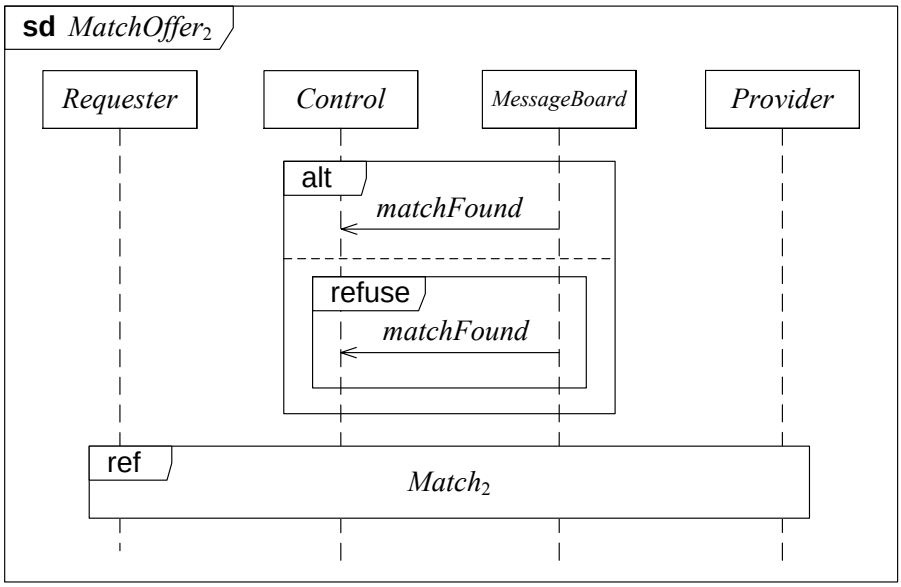


Figure 15.15: $MatchOffer_2$

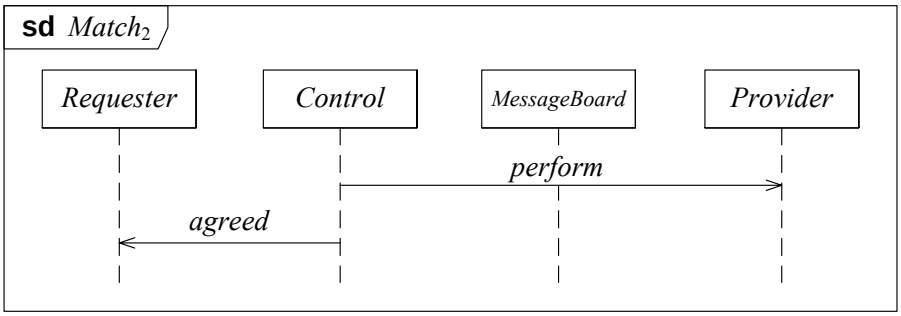


Figure 15.16: $Match_2$

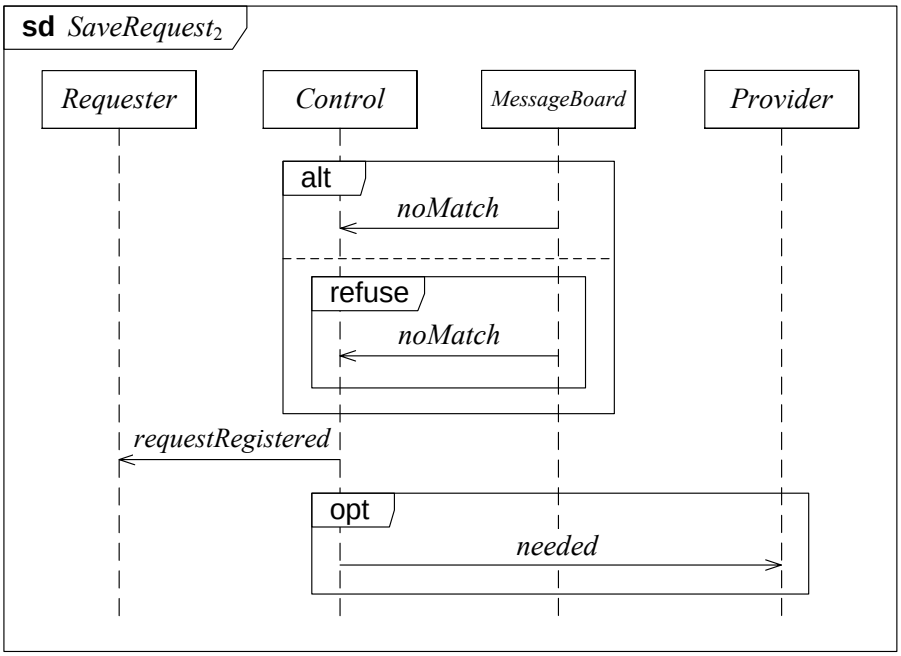


Figure 15.17: $SaveRequest_2$

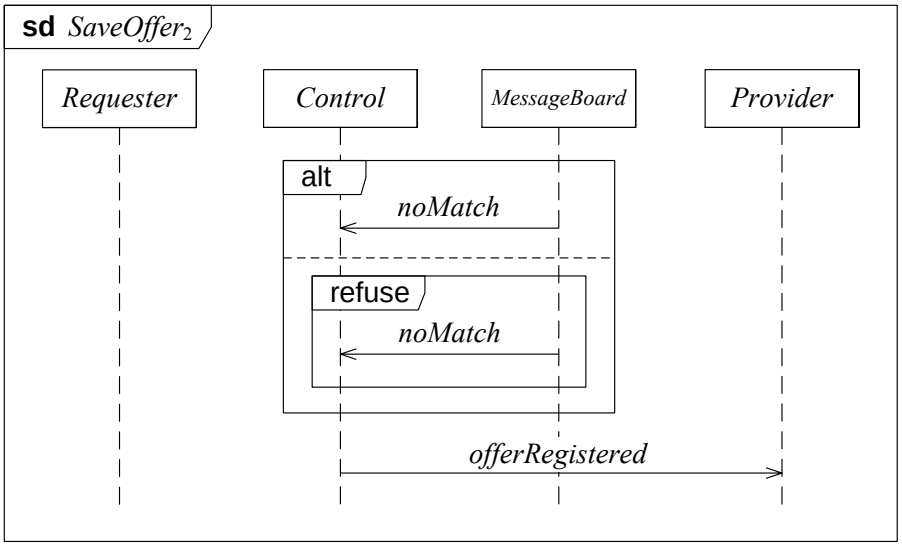


Figure 15.18: $SaveOffer_2$

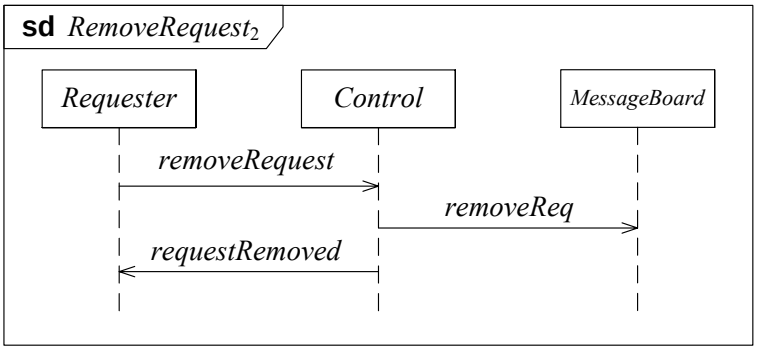


Figure 15.19: $RemoveRequest_2$

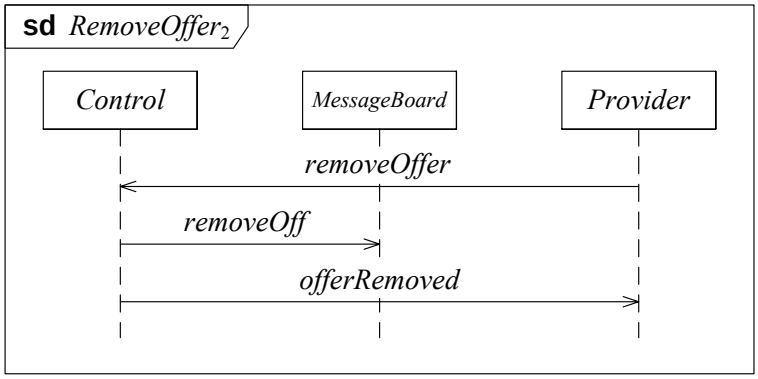


Figure 15.20: $RemoveOffer_2$

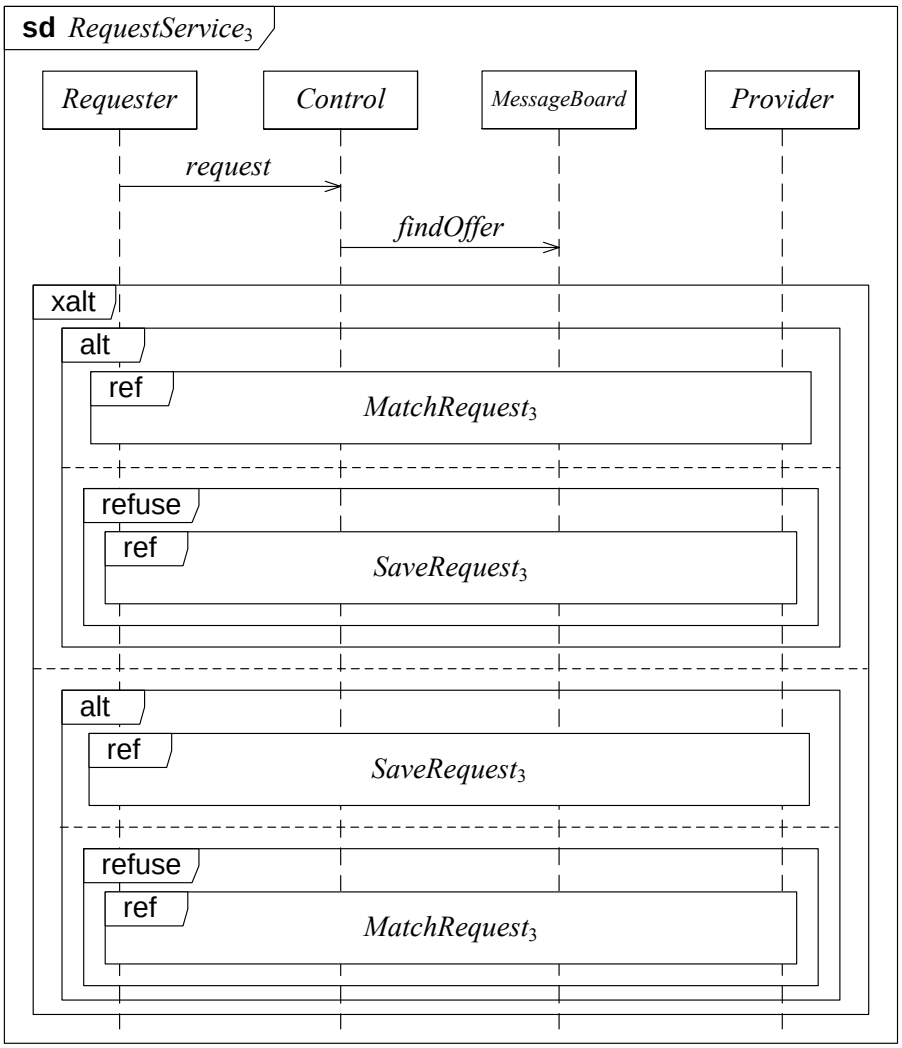


Figure 15.21: $RequestService_3$

sends messages *removeReq* and *removeOff* to *MessageBoard* when users remove requests or offers.

15.2.2.4 Level 3

At level 3, $RequestService_3$ and $OfferService_3$ refine $RequestService_2$ and $OfferService_2$. The transition from $RequestService_2$ to $RequestService_3$ (figure 15.21) introduces two changes. The main difference is that in each of the xalt operands of the diagram, the

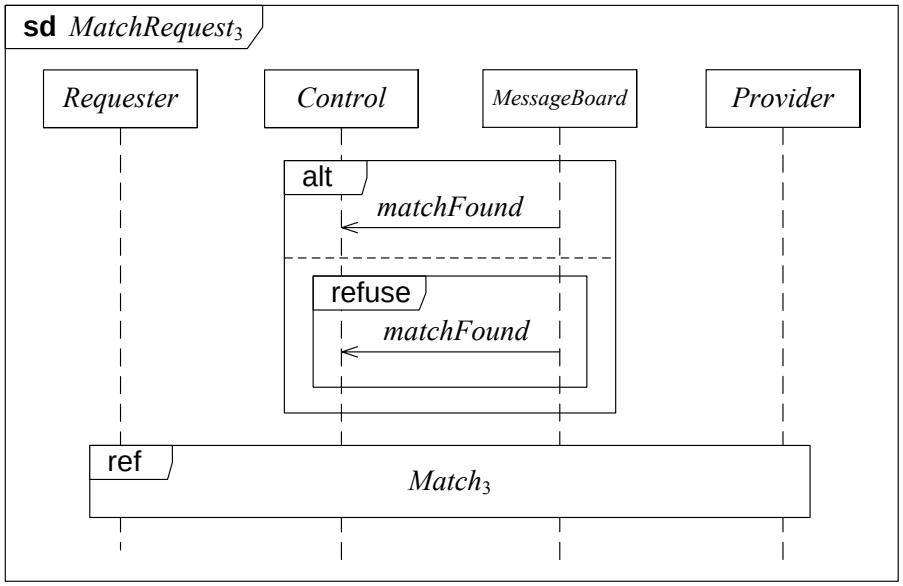


Figure 15.22: $MatchRequest_3$

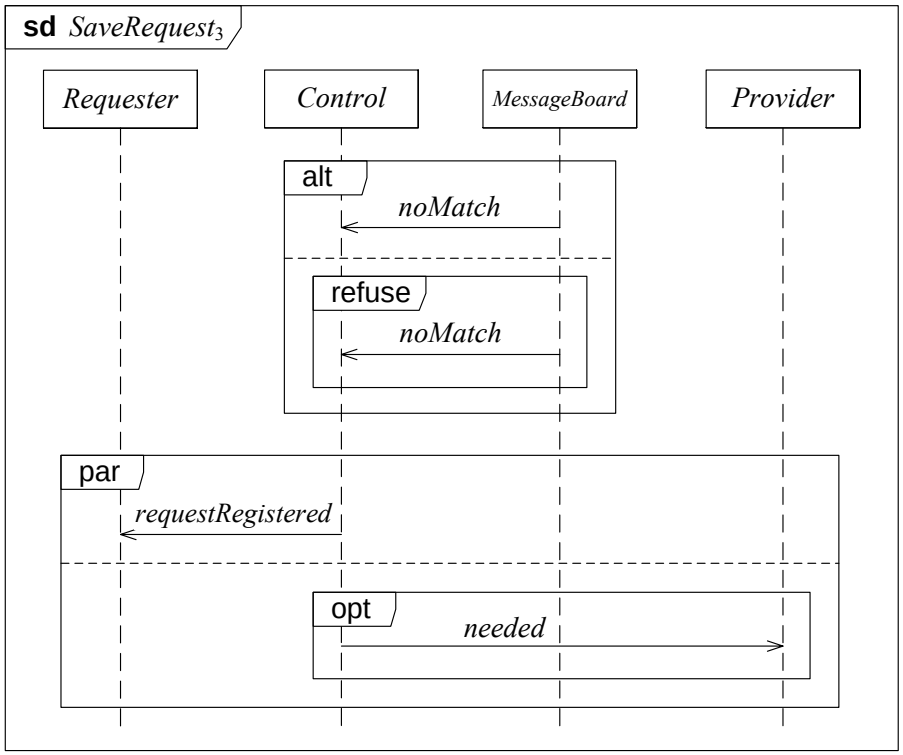


Figure 15.23: $SaveRequest_3$

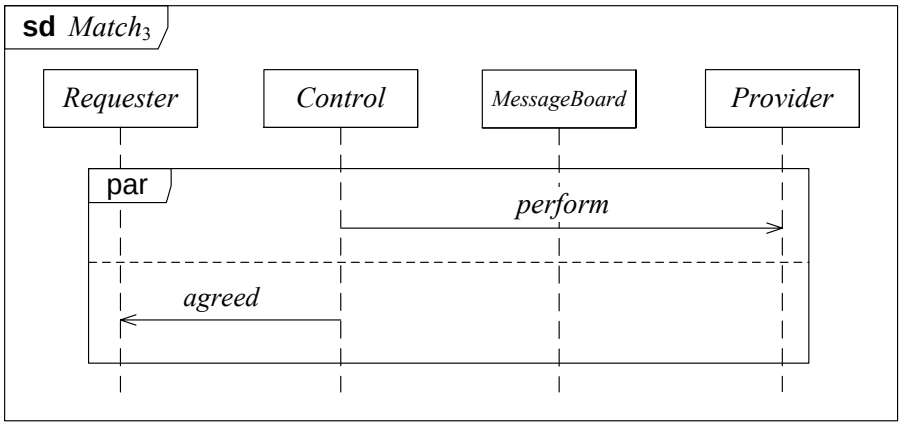


Figure 15.24: $Match_3$

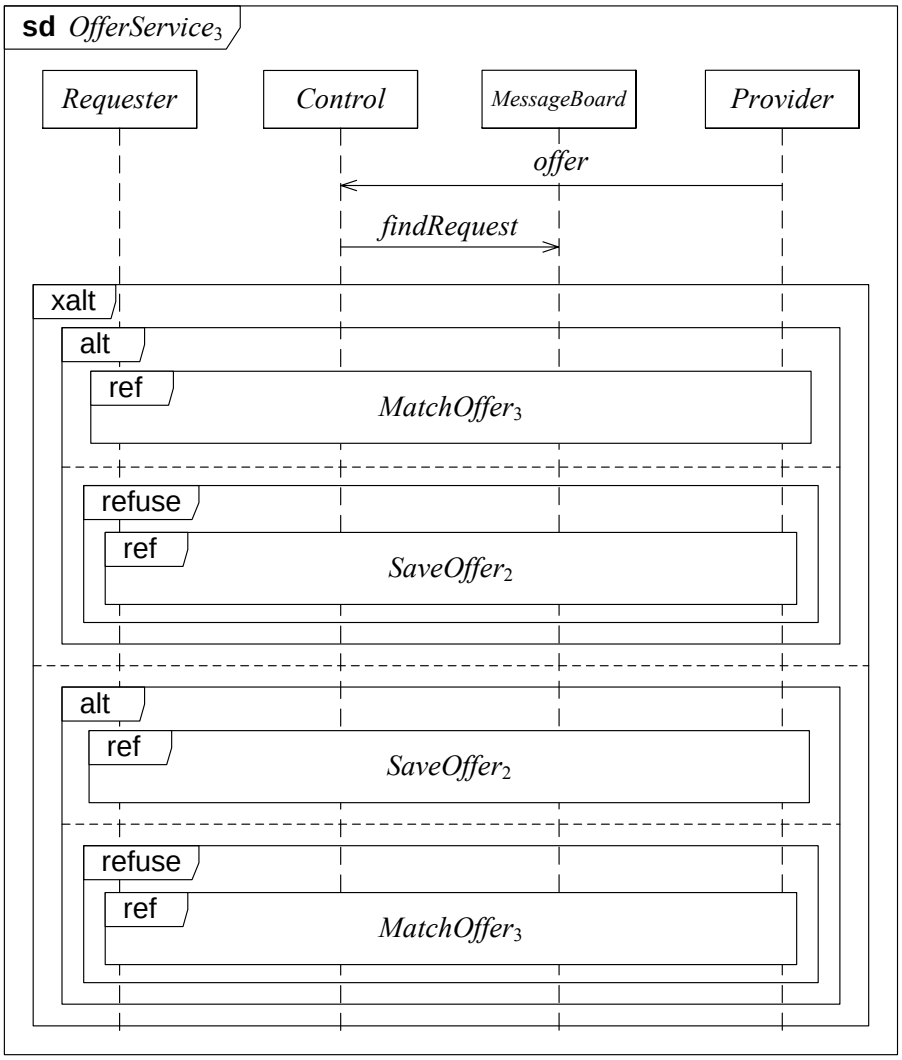


Figure 15.25: $OfferService_3$

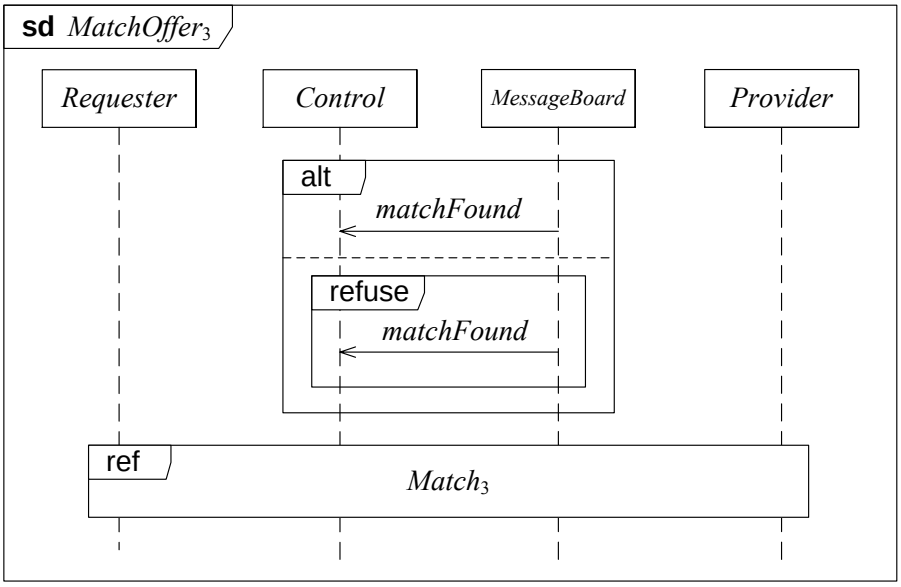


Figure 15.26: $MatchOffer_3$

contents of the other operand is made negative with a combination of alt and refuse.

The other difference is that $MatchRequest_2$ and $SaveRequest_2$ in $RequestService_2$ are replaced by $MatchRequest_2$ (figure 15.22) and $SaveRequest_3$ (figure 15.23). $MatchRequest_3$ is identical to $MatchRequest_2$ with the sole exception that if refers to $Match_3$ (figure 15.24) instead of $Match_2$. In $Match_3$, a par operator is placed around the messages *perform* and *agreed*, making them interleaved rather than sequenced along the *Control* lifeline. The transition from $SaveRequest_2$ to $SaveRequest_3$ is similar; a par operator is placed around the messages *requestRegistered* and *needed*, making them interleaved on *Control*.

The changes from $OfferService_2$ to $OfferService_3$ (figure 15.25) are almost identical to the changes from $RequestService_2$ to $RequestService_3$; the contents of each xalt operand is made negative in the other operand, and $MatchOffer_2$ is replaced by $MatchOffer_3$ (figure 15.26). Note, however, that $OfferService_3$ still refers to $SaveOffer_2$. As with $MatchRequest_3$, the only difference from $MatchOffer_2$ to $MatchOffer_3$ is that $MatchOffer_3$ refers to $Match_3$ instead of $Match_2$.

15.2.2.5 Level 4

Level 4 deals with refining the subscription functionality of the BuddySync system. A lifeline *MemberList* is added to the specification. The diagrams $SubscribeService_4$ (figure 15.27) and $UnsubscribeService_4$ (figure 15.28) refine $SubscribeService_0$ and $UnsubscribeService_0$ by decomposing the lifeline *System* into *Control* and *MemberList*. Upon reception of a *subscribe* or *unsubscribe* message, *Control* sends a message *add* or *remove*, respectively, to *MemberList* and receives a message *ok* from *MemberList*.

$RequestService_4$ (figure 15.29) and $OfferService_4$ (figure 15.30) are updates of $RequestService_3$ and $OfferService_3$ where the *MemberList* lifeline is added. *Control* sends a message *checkSubscription* to *MemberList* after reception of *request* or *offer* messages, and receives as reply either a message *subscriptionFound* or a message *subscriptionNotFound*. A new xalt is introduced in each of the diagrams to represent this choice. In the first operand the message *subscriptionNotFound* is followed by a message *notSubscribed*

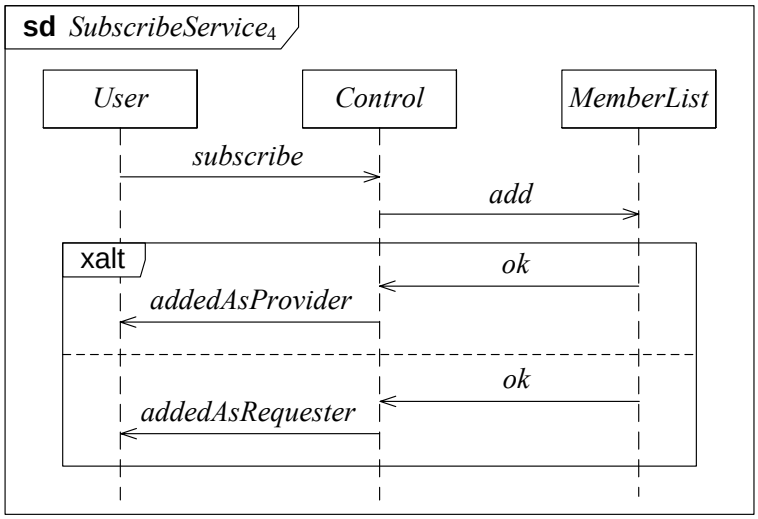


Figure 15.27: *SubscribeService*$_4$

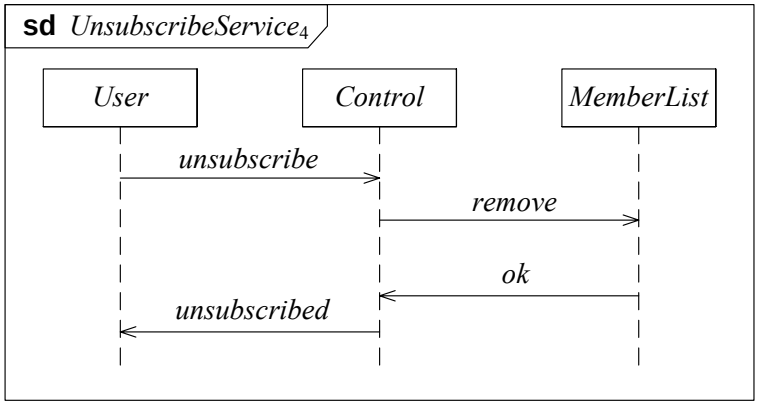


Figure 15.28: *UnsubscribeService*$_4$

from *Control* to the user (*Requester* in the case of *RequestService*$_4$ and *Provider* in the case of *OfferService*$_4$). In the second operand of the new xalt, the message *subscriptionFound* is sent from *MemberList* to *Control*, and the reminder is identical to the versions of the diagrams at level 3.

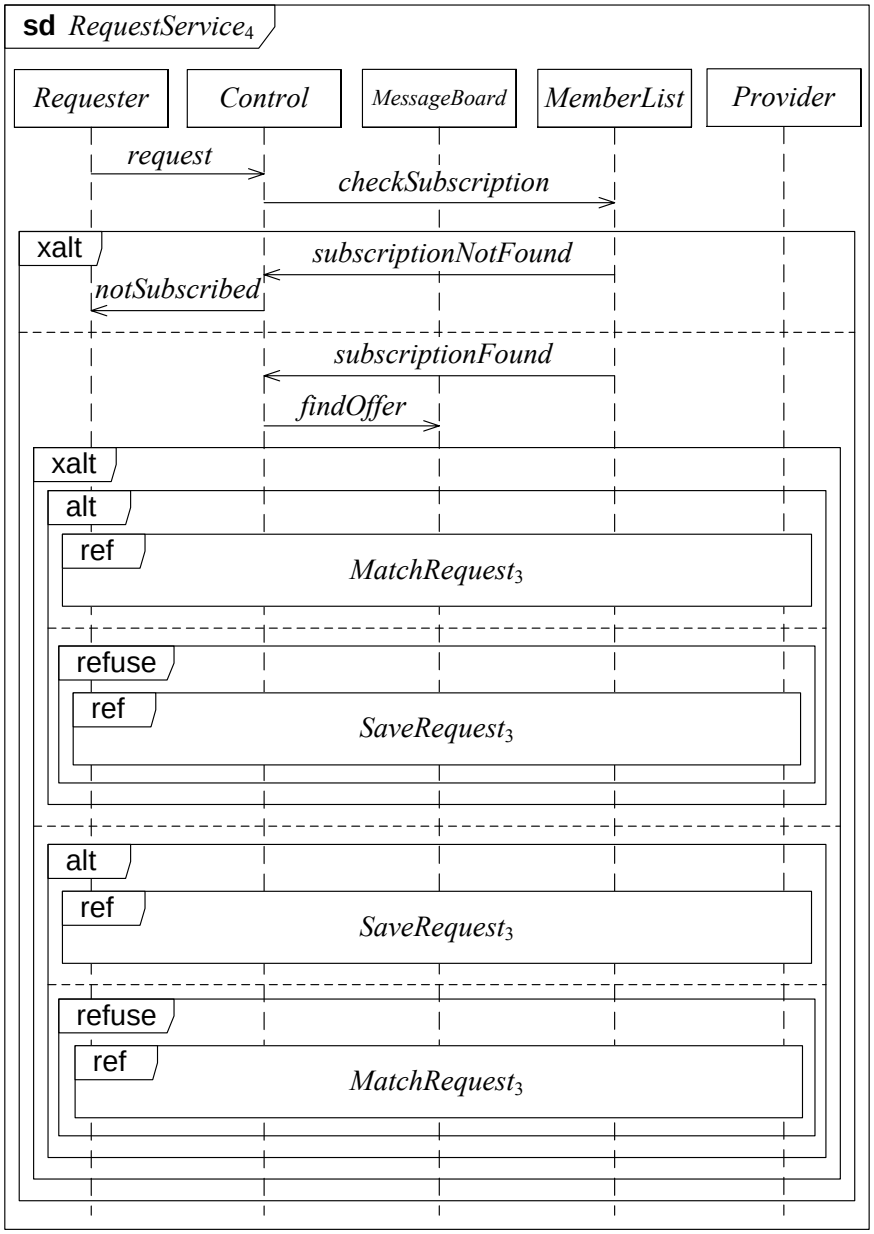


Figure 15.29: *RequestService*$_4$

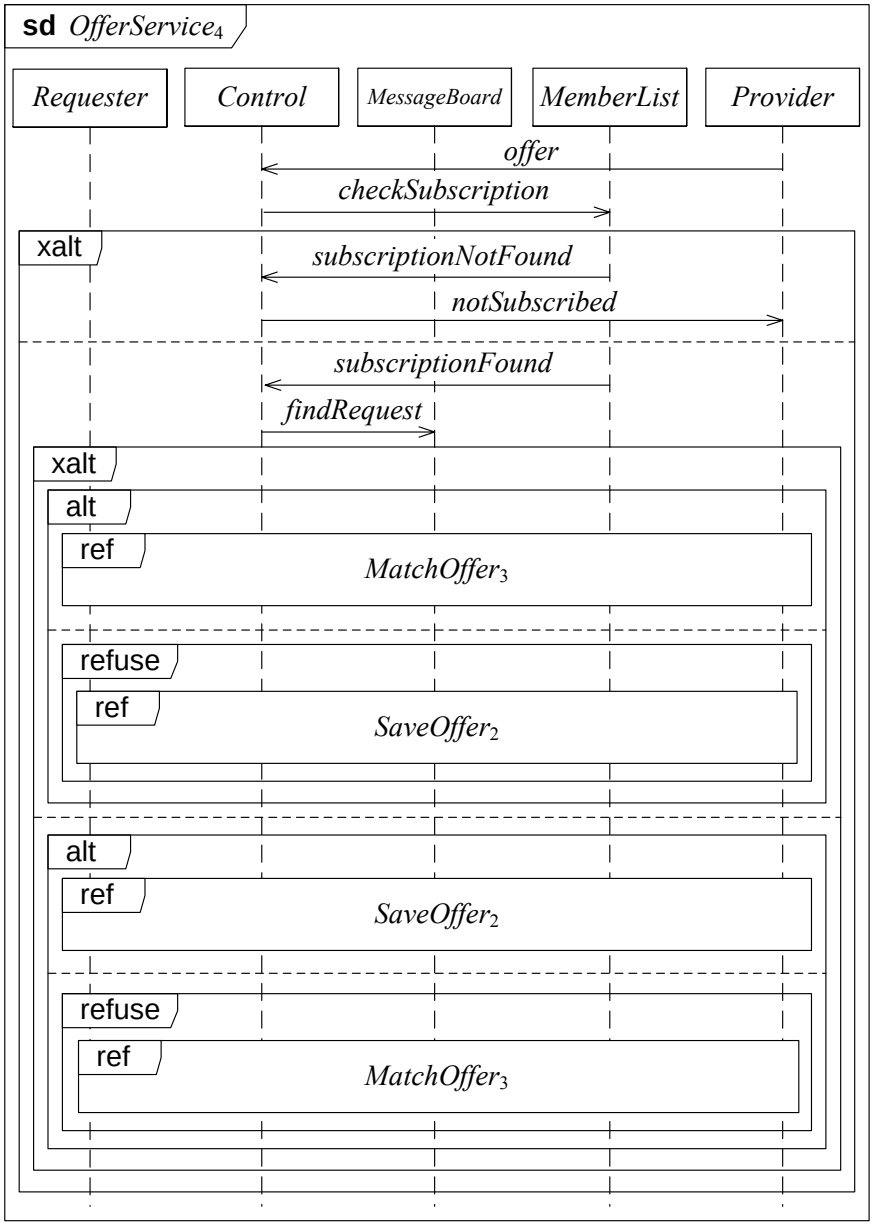


Figure 15.30: *OfferService*$_4$

15.2.3 Setup

The refinement verification case was carried out with the prototype version of the Escalator tool available at the time of the conclusion of the work of this thesis. The refinement verification mechanism of the tool compares two diagrams by generating their traces and comparing their sets of interaction obligations.

In [70, 159] four refinement relations are defined: general refinement, restricted general refinement, limited refinement and restricted limited refinement. These refinement relations are presented in section 5.4. In refinement verification, the Escalator tool checks all of these four refinement relations.

The refinement verification was applied to each diagram of the BuddySync specification for every update. This means, for example, that $RequestService_1$ was compared to $RequestService_0$, $RequestService_2$ was compared to $RequestService_1$, $RemoveRequest_2$ was compared to $RemoveRequest_0$, and $SubscribeService_4$ was compared to $SubscribeService_0$.

In addition to the verification of these refinements, we also did refinement verifications of the *Overview* diagram for some of the levels. The diagrams $Overview_0$, $Overview_1$ and $Overview_4$ were defined as follows:

$$
\begin{aligned}
Overview_0 \;=\; & RequestService_0 \ \mathsf{xalt} \\
& OfferService_0 \ \mathsf{xalt} \\
& RemoveRequest_0 \ \mathsf{xalt} \\
& RemoveOffer_0 \ \mathsf{xalt} \\
& SubscribeService_0 \ \mathsf{xalt} \\
& UnsubscribeService_0
\end{aligned}
$$

$$
\begin{aligned}
Overview_1 \;=\; & RequestService_1 \ \mathsf{xalt} \\
& OfferService_1 \ \mathsf{xalt} \\
& RemoveRequest_0 \ \mathsf{xalt} \\
& RemoveOffer_0 \ \mathsf{xalt} \\
& SubscribeService_0 \ \mathsf{xalt} \\
& UnsubscribeService_0
\end{aligned}
$$

$$
\begin{aligned}
Overview_4 \;=\; & RequestService_4 \ \mathsf{xalt} \\
& OfferService_4 \ \mathsf{xalt} \\
& RemoveRequest_2 \ \mathsf{xalt} \\
& RemoveOffer_2 \ \mathsf{xalt} \\
& SubscribeService_4 \ \mathsf{xalt} \\
& UnsubscribeService_4
\end{aligned}
$$

Using these diagrams we did refinement verification between $Overview_0$ and $Overview_4$, and between $Overview_1$ and $Overview_4$.

Because some of the refinements involve decomposition of lifelines, special care was taken when the refinements were specified. Escalator has a mechanism for hiding lifelines in the trace generation, by specifying which lifelines should be considered external. When traces are generated, only events that have a lifeline in the set of external lifelines as transmitter or receiver will be present in the traces, and all other events hidden. In Escalator it is also possible to specify lifeline mappings to be applied in the refinement verification. When applied, lifelines in the events of the traces are substituted according to the mappings. In this way, diagrams may be compared even

if the names of their lifelines differ. Both these mechanisms were carefully applied to handle the decompositions of lifelines appropriately when investigating the refinements of the case.

15.2.4 Results

Table 15.3 provides the results of the refinement verification of the diagrams of the BuddySync specification. The leftmost column shows which refinements that were checked. For each refinement the table shows which lifelines that were considered external in the refinement and the lifeline mappings applied, if any. The results are shown for each of the four refinement relations, abbreviated *g* for general, *rg* for restricted general, *l* for limited and *rl* for restricted limited refinement, where "Y" means it is a correct refinement and "N" that it is not a correct refinement.

In addition to the refinements reported in table 15.3, we also did refinement verification of the $Overview_0$, $Overview_1$ and $Overview_4$ diagrams defined in the above

Refinement	**Lifelines**	**Mapping**	***g***	***rg***	***l***	***rl***
$RequestService_0 \overset{?}{\rightsquigarrow} RequestService_1$	*Requester*		Y	Y	Y	Y
$OfferService_0 \overset{?}{\rightsquigarrow} OfferService_1$	*Provider*		Y	Y	Y	Y
$RequestService_1 \overset{?}{\rightsquigarrow} RequestService_2$	*Requester*, *Provider*	$Control \mapsto System$	Y	Y	Y	Y
$OfferService_1 \overset{?}{\rightsquigarrow} OfferService_2$	*Requester*, *Provider*	$Control \mapsto System$	Y	Y	Y	Y
$RemoveRequest_0 \overset{?}{\rightsquigarrow} RemoveRequest_2$	*Requester*	$Control \mapsto System$	Y	Y	Y	Y
$RemoveOffer_0 \overset{?}{\rightsquigarrow} RemoveOffer_2$	*Provider*	$Control \mapsto System$	Y	Y	Y	Y
$RequestService_2 \overset{?}{\rightsquigarrow} RequestService_3$	*All*		Y	N	Y	N
$OfferService_2 \overset{?}{\rightsquigarrow} OfferService_3$	*All*		Y	N	Y	N
$RequestService_3 \overset{?}{\rightsquigarrow} RequestService_4$	*Requester*, *Provider*, *MessageBoard*		Y	Y	N	N
$OfferService_3 \overset{?}{\rightsquigarrow} OfferService_4$	*Requester*, *Provider*, *MessageBoard*		Y	Y	N	N
$SubscribeService_0 \overset{?}{\rightsquigarrow} SubscribeService_4$	*User*	$Control \mapsto System$	Y	Y	Y	Y
$UnsubscribeService_0 \overset{?}{\rightsquigarrow} UnsubscribeService_4$	*User*	$Control \mapsto System$	Y	Y	Y	Y

Table 15.3: Refinement verification results

Refinement	Lifelines	Mappings	*g*	*rg*	*l*	*rl*
$Overview_0 \overset{?}{\rightsquigarrow} Overview_4$	*Requester*, *Provider*, *User*	*Control* $\mapsto$ *System*	N	N	N	N
$Overview_1 \overset{?}{\rightsquigarrow} Overview_4$	*Requester*, *Provider*, *User*	*Control* $\mapsto$ *System*	Y	N	N	N

Table 15.4: Refinement verification results of *Overview* diagrams

section. The result of these refinement verifications are presented in table 15.4.

15.2.5 Discussion

From table 15.3, we see that most of the steps in the development of the specification are correct refinements with respect to all of the four refinement relations. The exceptions are $RequestService_2 \rightsquigarrow RequestService_3$ and $OfferService_2 \rightsquigarrow OfferService_3$ that are not restricted (general or limited) refinements, and $RequestService_3 \rightsquigarrow RequestService_4$ and $OfferService_3 \rightsquigarrow OfferService_4$ that are not limited (restricted or unrestricted) refinements. An analysis of the diagrams reveals that the reason for the former is that new positive behaviors are introduced in $RequestService_3$ and $OfferService_3$ by the par operators introduced in the referred diagrams $SaveRequest_3$ and $Match_3$. Adding positive traces to a specification is not allowed in restricted refinement.

The latter is a result of the new xalt operator in $RequestService_4$ and $OfferService_4$. These xalts introduce new interaction obligations in the diagrams that are not refinements of pre-existing interaction obligations. This is not allowed in limited refinement.

When we look at table 15.4 we note that $Overview_4$ is not a refinement of $Overview_0$. The reason for this is that $RequestService_0$ and $OfferService_0$ do not have communication involving, respectively, *Provider* and *Requester*, while this is introduced in later levels. When in table 15.3, $RequestService_1$ and $OfferService_1$ are refinements of $RequestService_0$ and $OfferService_0$, it is because only *Requester* was external lifeline in the case of *RequestService* and only *Provider* in the case of *OfferService*. When we used both *Requester* and *Provider* as external lifelines in the refinement verification of *Overview*, these refinements were no longer valid.

Table 15.4 shows that $Overview_4$ is indeed a general refinement of $Overview_1$, where $RequestService_0$ and $OfferService_0$ are replaced by $RequestService_1$ and $OfferService_1$. $Overview_4$ is not a restricted nor a limited refinement of $Overview_1$. This is expected since 1) $RequestService_3$ and $OfferService_3$ are not restricted refinements of $RequestService_2$ and $OfferService_2$, 2) $RequestService_4$ and $OfferService_4$ are not limited refinements of $RequestService_3$ and $OfferService_3$, and 3) we know that the refinement relations are transitive and monotonic.

To summarize, we see that all unexpected results (in the sense that the tool reported incorrect refinements) can be explained as actual incorrect refinements. An analysis of the diagrams for the remaining cases reveals that these are in fact correct refinements. We can therefore conclude that the results of the refinement verification of the BuddySync specification are as expected, and that the success criterion is fulfilled.

15.3 Refinement testing

In this second part of the case study we apply the refinement testing functionality of the Escalator tool to the BuddySync specification. In section 15.3.1 we provide assumptions and success criteria for the refinement testing part of the study. Section 15.3.2 presents the version of the BuddySync specification applied, and section 15.3.3 presents the setup of the refinement testing part of the case study. In section 15.3.4 the results of the refinement testing are provided and in section 15.3.5 we discuss the results.

15.3.1 Assumptions and success criteria

In this part of the case study we still apply a simplified version of the BuddySync specification, but due to specifics of the refinement testing functionality there are some differences from the version applied in the refinement verification. These difference are explained in the following section. We do refinement testing of the same refinements that we applied the refinement verification to in the first part of the case study.

For this part of the case study we formulate two success criteria:

1. *The refinement testing functionality of the tool provides the expected results when applied to the BuddySync specification.* As with the first part of the case, the expected result is that all refinements are correct, but also here we must consider the possibility that the changes we have done in the specification result in incorrect refinements.

2. *The refinement testing applied to the BuddySync specification gives a reasonable coverage of the specification.* As opposed to the refinement verification, the refinement testing is randomized. We therefore have no guarantee for full coverage of the specification in the refinement testing. A desired property is that we have reasonable coverage. By this we mean that even though not all details of the specification are covered by the refinement testing, at least the major parts or branches (e.g. alt operands) of the specification are covered.

15.3.2 The BuddySync specification for the refinement testing

We make four changes to the BuddySync specification with respect to the specification presented in section 15.2.2:

- The most visible change is that we remove the lifelines *Requester*, *Provider* and *User* from all the diagrams so that all messages going to and from these lifelines instead go to and from the frames of the diagrams. The reason for this is that in the test generation and execution, the frame is used to define the interface of a diagram.

- The testing mechanism does not handle the difference between alt and xalt; for this reason we replace all xalts with alts. This applies to the diagrams *RequestService*$_0$, *OfferService*$_0$, *SubscribeService*$_0$, *RequestService*$_1$, *OfferService*$_1$, *RequestService*$_2$, *OfferService*$_2$, *RequestService*$_3$, *OfferService*$_3$, *RequestService*$_4$, *OfferService*$_4$ and *SubscribeService*$_4$.

- In contrast to the refinement verification mechanism, the testing functionality of Escalator does handle the assert operator. We therefore insert asserts in the diagrams where there are asserts in the original specification of [154]. This applies to the diagrams $RemoveRequest_0$, $RemoveOffer_0$, $UnsubscribeService_0$, $RemoveRequest_2$, $RemoveOffer_2$, $RequestService_4$, $OfferService_4$ and $UnsubscribeService_4$.

- In levels 2 through 4 we rename the lifeline *Control* to *System*. The reason for this is that we need the interface/observation point of the diagrams to be the same throughout the levels, and choose to use the lifeline name *System* to be this interface/observation point.

The simplification we did in the refinement verification when we removed actions, constraints and guards from the original specification of [154], still apply.

With these changes, the specification of the BuddySync system is presented below:

- Level 0 of the specification of the BuddySync system is found in figures 15.31–15.36.

- Level 1 is found in figures 15.37–15.41.

- Level 2 is found in figures 15.42–15.50.

- Level 3 is found in figures 15.51–15.56.

- Finally, level 4 of the specification is found in figures 15.57–15.60.

In this presentation we do not provide any explanatory text, but refer to the explanations of the specification given in section 15.2.2.

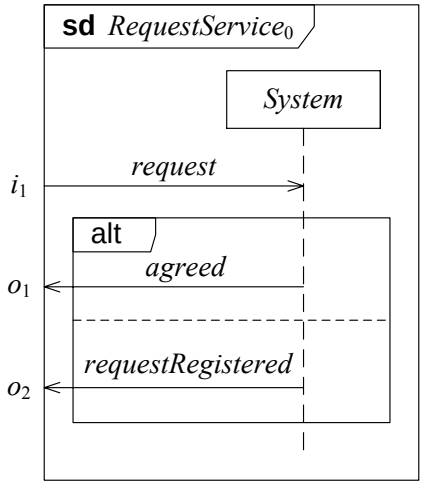


Figure 15.31: $RequestService_0$

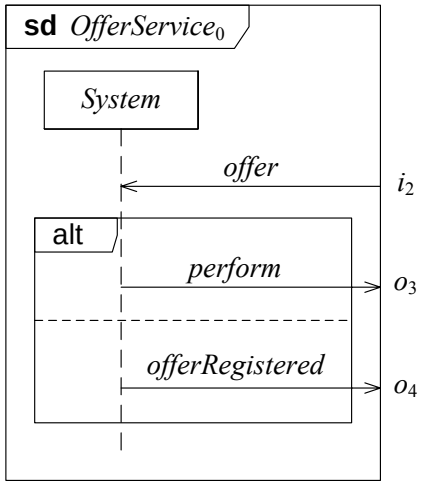


Figure 15.32: $OfferService_0$

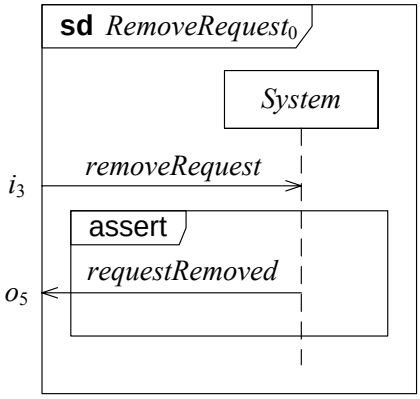


Figure 15.33: $RemoveRequest_0$

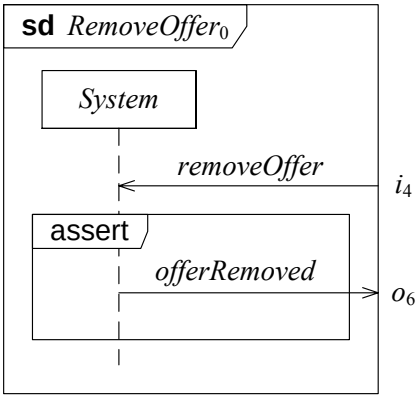


Figure 15.34: $RemoveOffer_0$

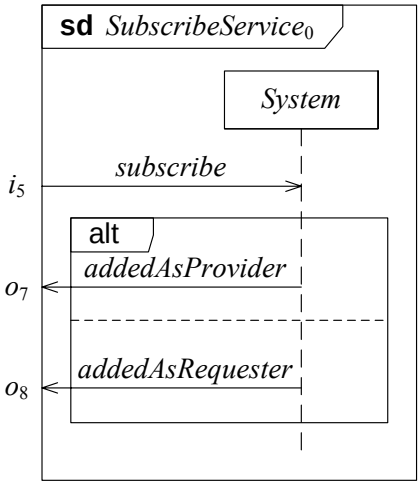


Figure 15.35: $SubscribeService_0$

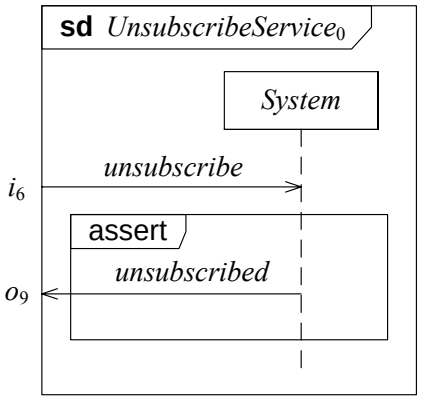


Figure 15.36: $UnsubscribeService_0$

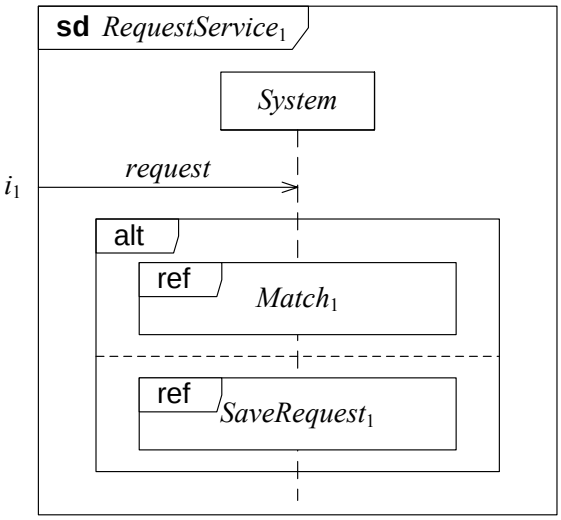


Figure 15.37: $RequestService_1$

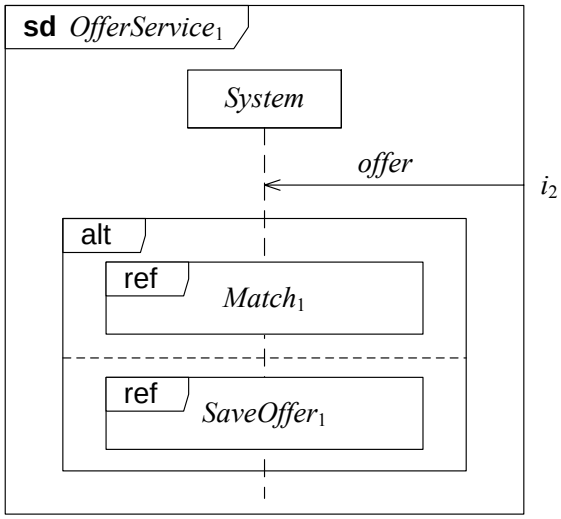


Figure 15.38: $OfferService_1$

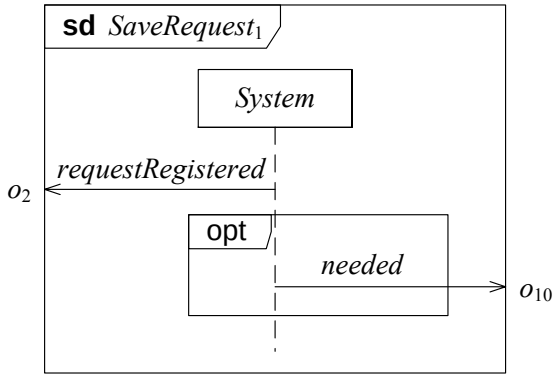


Figure 15.39: $SaveRequest_1$

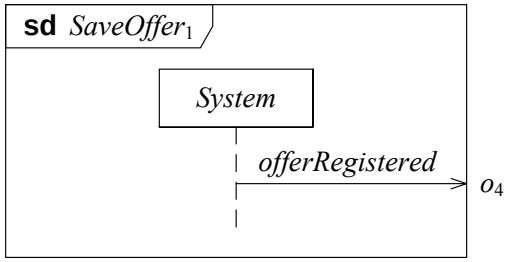


Figure 15.40: $SaveOffer_1$

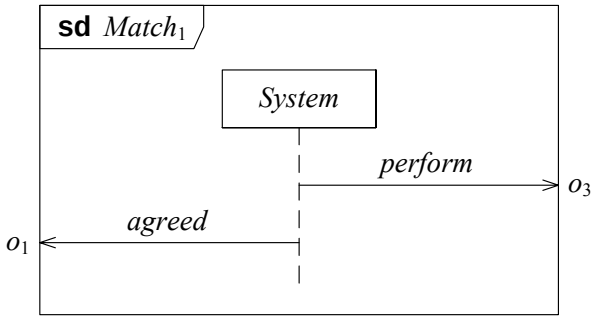


Figure 15.41: $Match_1$

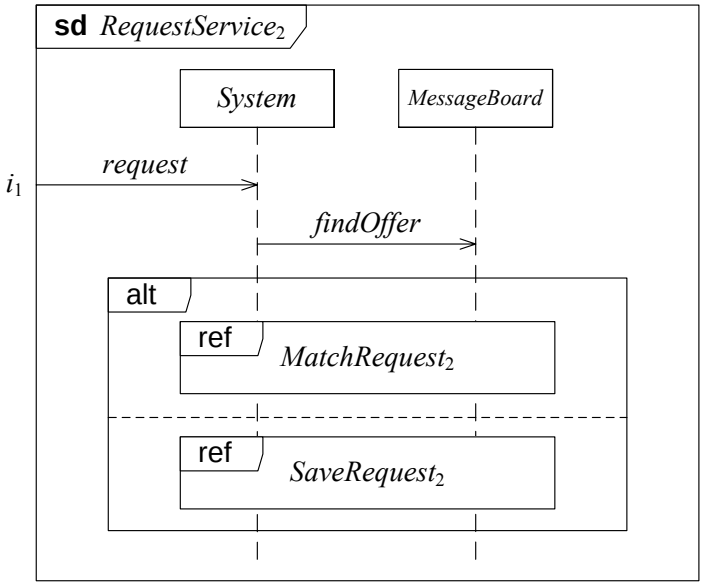


Figure 15.42: $RequestService_2$

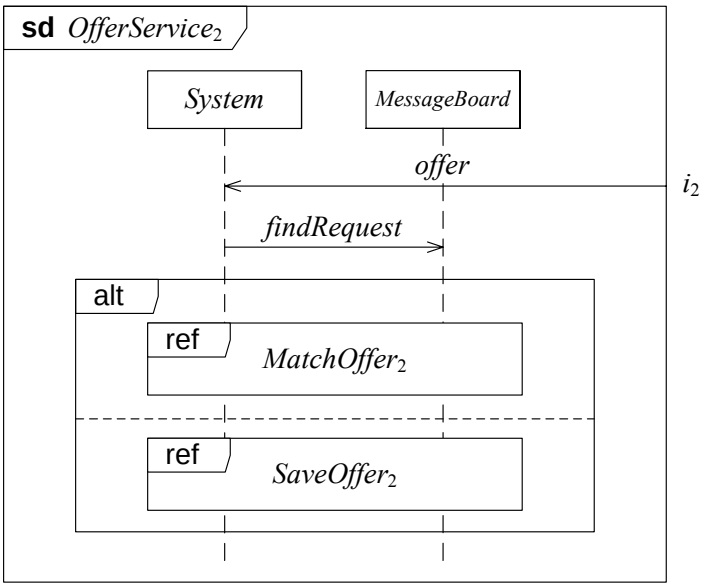


Figure 15.43: *OfferService*$_2$

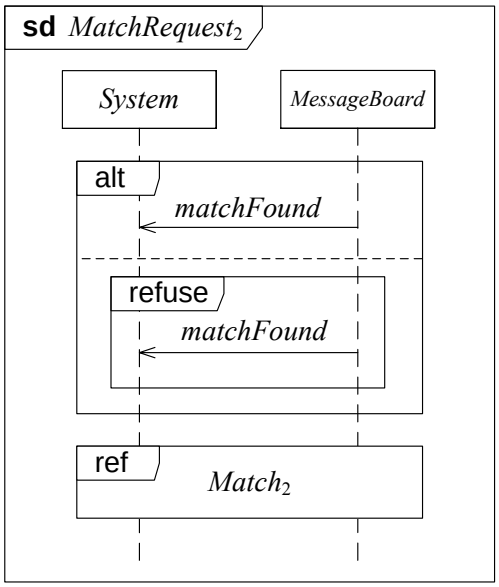


Figure 15.44: *MatchRequest*$_2$

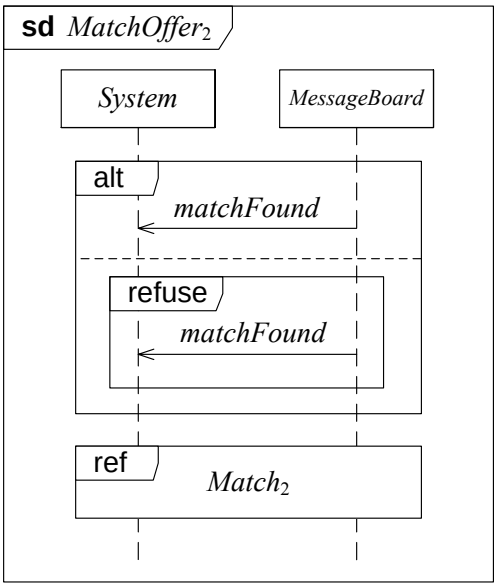


Figure 15.45: *MatchOffer*$_2$

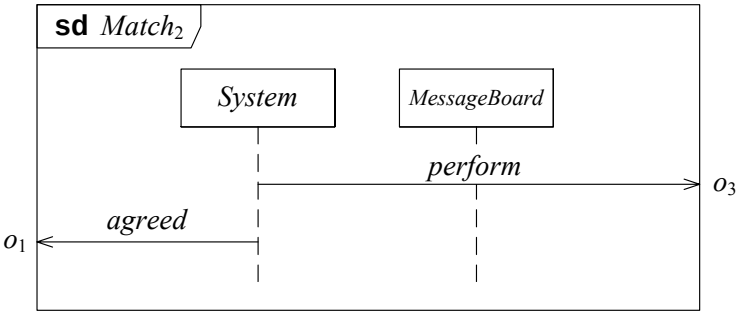


Figure 15.46: *Match*$_2$

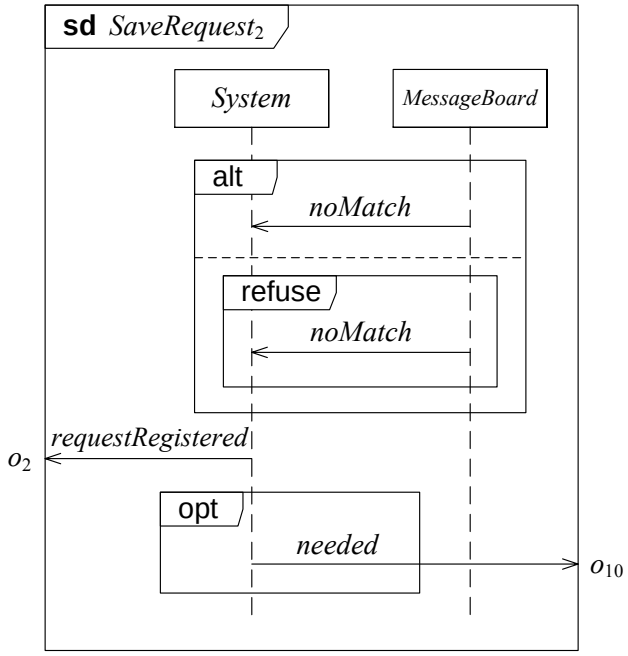


Figure 15.47: $SaveRequest_2$

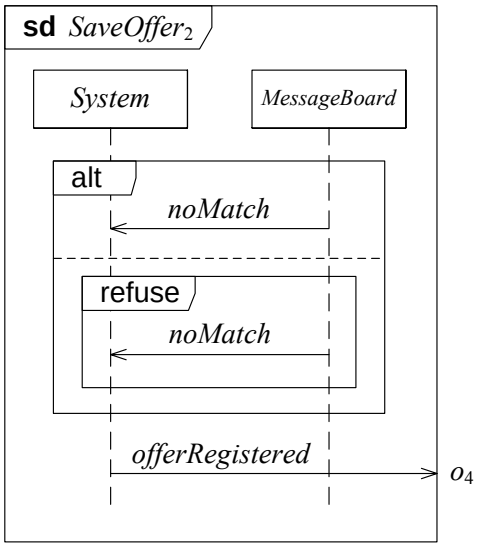


Figure 15.48: $SaveOffer_2$

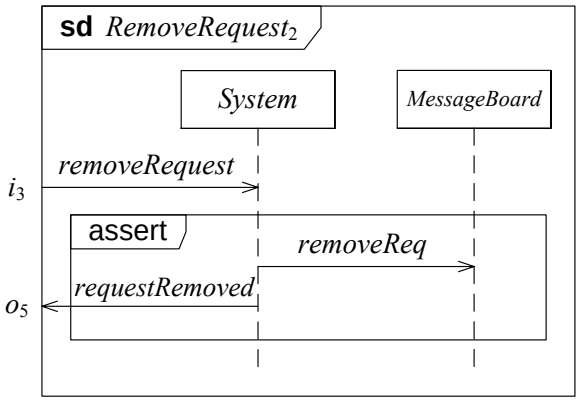


Figure 15.49: $RemoveRequest_2$

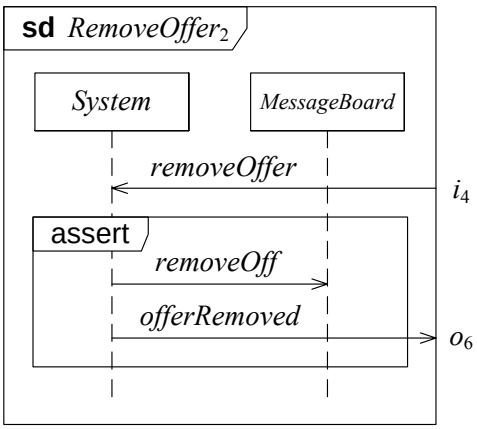


Figure 15.50: $RemoveOffer_2$

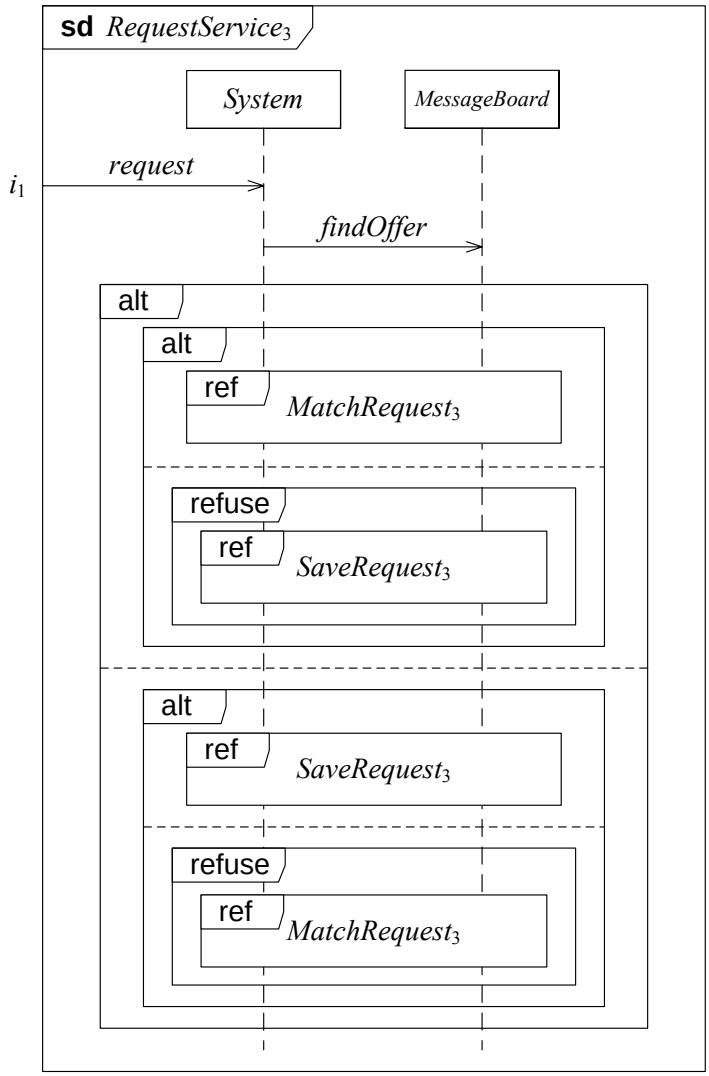


Figure 15.51: *RequestService*$_3$

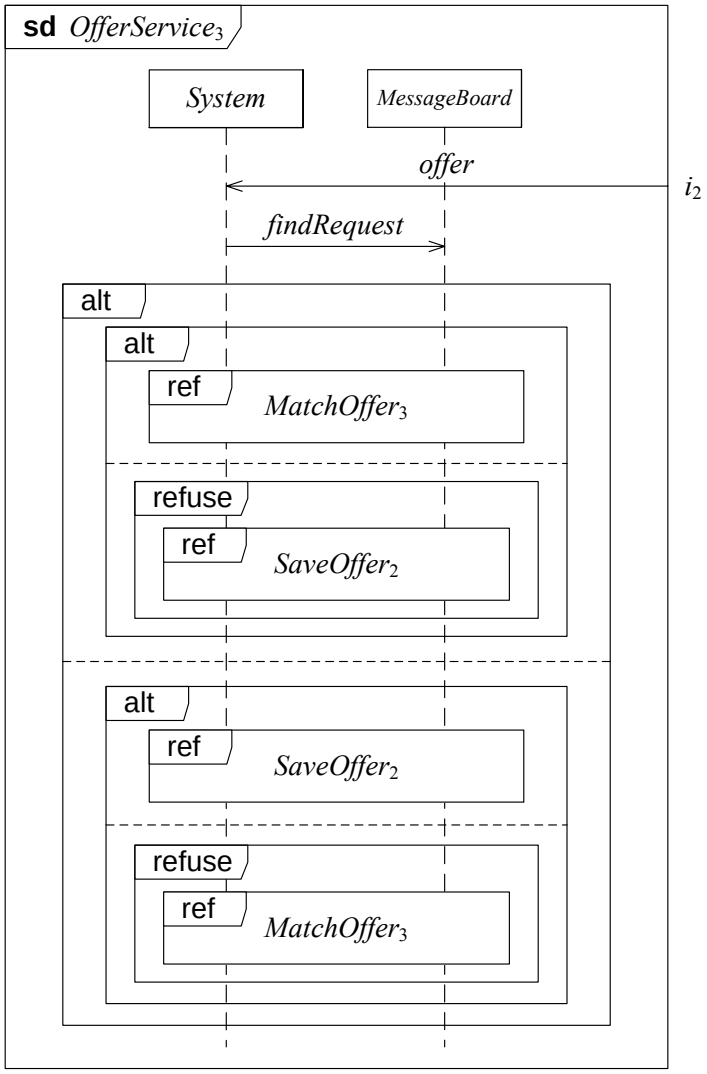


Figure 15.52: *OfferService*$_3$

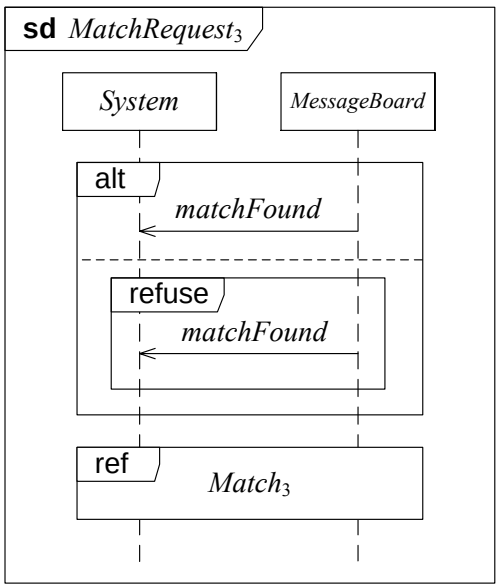


Figure 15.53: *MatchRequest*$_3$

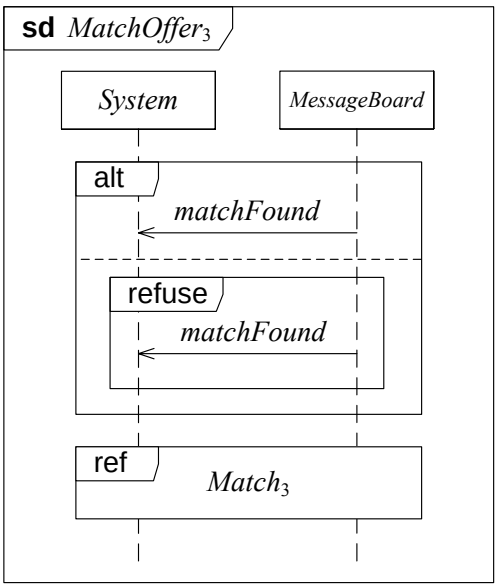


Figure 15.54: *MatchOffer*$_3$

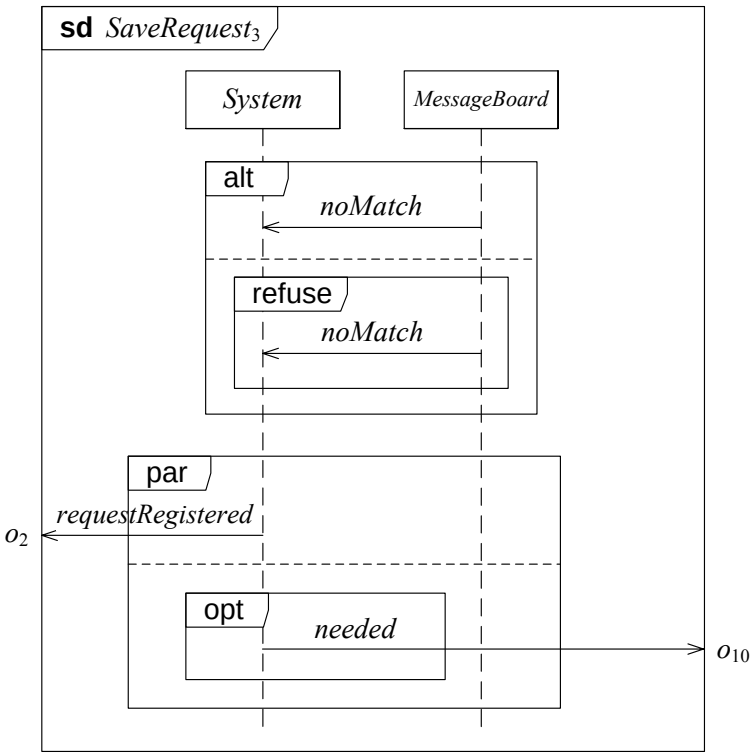


Figure 15.55: *SaveRequest*$_3$

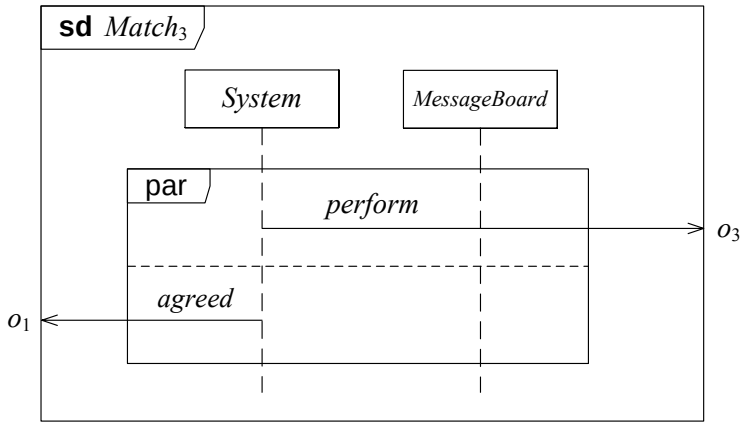


Figure 15.56: *Match*$_3$

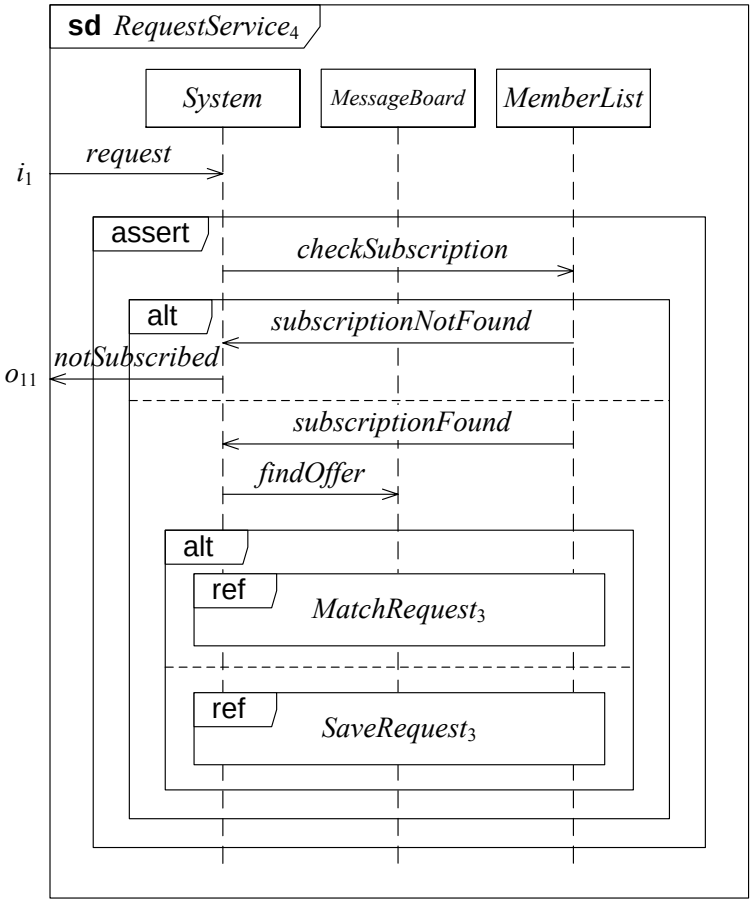


Figure 15.57: *RequestService*$_4$

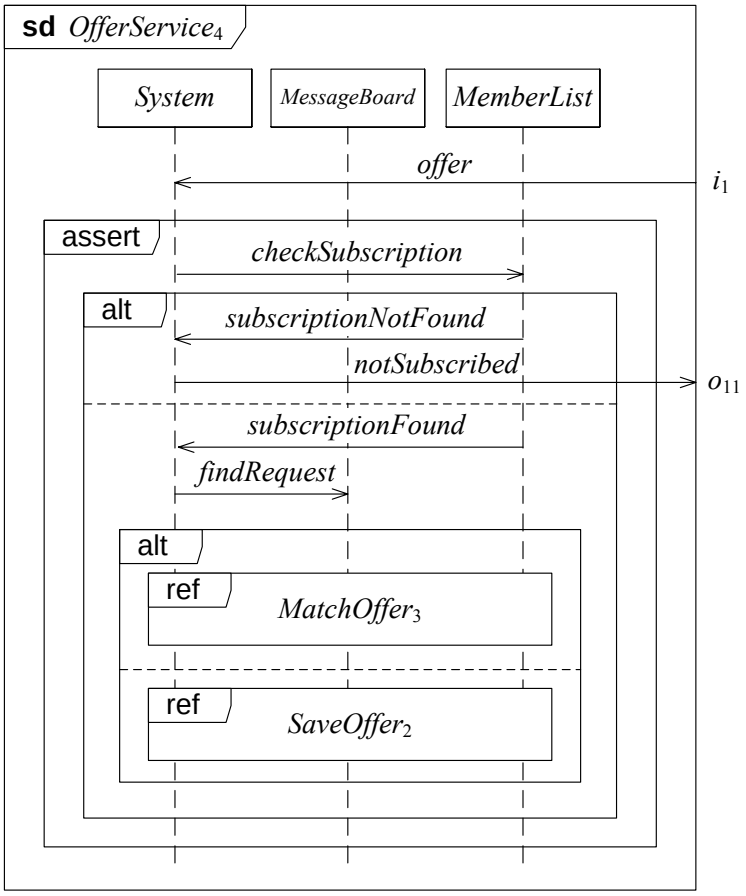


Figure 15.58: *OfferService*$_4$

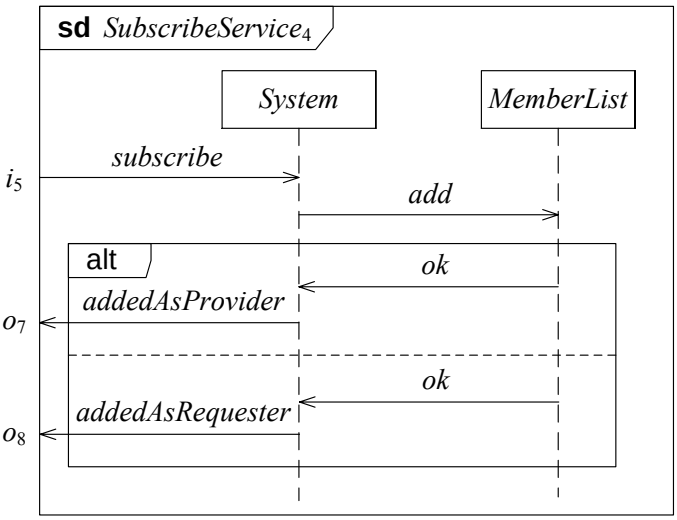


Figure 15.59: *SubscribeService*$_4$

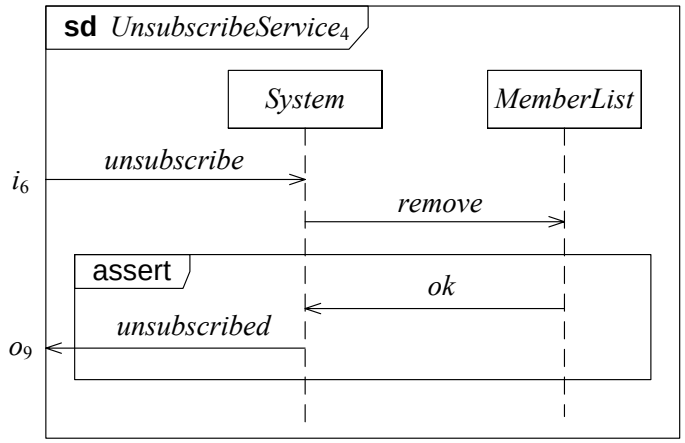


Figure 15.60: *UnsubscribeService*$_4$

15.3.3 Setup

As with the refinement verification part of the case, the refinement testing part was carried out with the prototype version of the Escalator tool available at the time of the conclusion of the work on this thesis, however with the following reservation: Some changes were made to the test generation and test execution (specifically some changes in the assignment of verdicts) after the refinement testing was conducted, and these changes affect the testing of some, but not all, of the refinements. Some of the results have therefore been updated according to these changes so that the results presented below reflect the state of the tool at the time of the conclusion of the thesis. This means that we present the results as they would have been if the refinement testing was conducted again using the same specification on the updated tool.

In the refinement testing part of the study we tested the same refinements as were checked by the refinement verification, with exception of the *Overview* diagrams. In the refinement testing we only handle diagrams with a single interaction obligation, and hence only check for refinement of interaction obligations, i.e. the relation $\rightsquigarrow_r$ (see section 5.4.1). For all pairs of diagrams tested we applied both "testing down" and "testing up". When doing refinement testing of a refinement $d \rightsquigarrow_r d'$, "testing down" means tests are generated from d and executed against d', while "testing up" means tests are generated from d' and executed against d. In the cases where a diagram that tests were executed against had a larger alphabet of external transmit events than the diagram the tests were generated from, these extra events were given as input to the test generation.

Refinement	**Direction**	**Tests**	**Runs**
OfferService$_2 \rightsquigarrow_r$ *OfferService*$_3$	*down*	5	5
RequestService$_3 \rightsquigarrow_r$ *RequestService*$_4$	*down*	5	3
RequestService$_3 \rightsquigarrow_r$ *RequestService*$_4$	*up*	5	3
RequestService$_3 \rightsquigarrow_r$ *RequestService*$_4$	*up*	1	5
OfferService$_3 \rightsquigarrow_r$ *OfferService*$_4$	*down*	5	3
OfferService$_3 \rightsquigarrow_r$ *OfferService*$_4$	*up*	5	3

Table 15.5: Exceptions to specification of number of tests and runs

The refinement testing is randomized, and therefore the number of tests and number of runs of each test must be specified. For the testing of most refinements we specified 10 tests to be generated for each testing direction and 5 runs of each test. The exceptions are shown in table 15.5. (Note that "testing up" of the refinement $RequestService_3 \rightsquigarrow_r RequestService_4$ was conducted twice. When presenting the results, the results of these two iterations are aggregated.) All refinement tests were carried out with random seeds.

Also a timeout must be specified when doing refinement testing. Due to a performance issue in the implementation of the refinement testing the time needed to carry out the refinement testing seems to increase exponentially with the number of operators in the specification. We believe this to be an implementation issue and not a fundamental flaw in our test generation and test execution, and in view of this we find the time used by the tool for the refinement testing uninteresting to report.

15.3.4 Results

The results of the refinement testing are shown in tables 15.6 and 15.7. Each generated test is given a name. The names numerate the tests and indicate whether they have been generated and executed during "testing down" or "testing up".

For each pair of diagrams that were tested we list the tests involved and the number

Diagrams		**Events**	**Tests**	**Runs**	**Verdict**
$RequestService_0$	$RequestService_1$	!*perform*, !*needed*	T_1^{down}	3	**pass**
			T_2^{down}	3	**pass**
			T_3^{up}	2	**fail**
			T_4^{up}	2	**fail**
$OfferService_0$	$OfferService_1$	!*agreed*	T_5^{down}	2	**pass**
			T_6^{down}	2	**pass**
			T_7^{up}	2	**fail**
			T_8^{up}	2	**fail**
$RequestService_1$	$RequestService_2$		T_9^{down}	2	**pass**
			T_{10}^{down}	3	**pass**
			T_{11}^{up}	3	**pass**
			T_{12}^{up}	3	**pass**
$OfferService_1$	$OfferService_2$		T_{13}^{down}	2	**pass**
			T_{14}^{down}	2	**pass**
			T_{15}^{up}	2	**pass**
			T_{16}^{up}	2	**pass**
$RemoveRequest_0$	$RemoveRequest_2$		T_{17}^{down}	1	**pass**
			T_{18}^{down}	1	**pass**
			T_{19}^{up}	1	**pass**
			T_{20}^{up}	1	**pass**
$RemoveOffer_0$	$RemoveOffer_2$		T_{21}^{down}	1	**pass**
			T_{22}^{down}	1	**pass**
			T_{23}^{up}	1	**pass**
			T_{24}^{up}	1	**pass**

Table 15.6: Testing results

Diagrams		Events	Tests	Runs	Verdict
$RequestService_2$	$RequestService_3$		T_{25}^{down}	3	**pass**
			T_{26}^{down}	3	**pass**
			T_{27}^{up}	3	**pass**
			T_{28}^{up}	3	**pass**
$OfferService_2$	$OfferService_3$		T_{29}^{down}	3	**pass**
			T_{30}^{down}	2	**pass**
			T_{31}^{up}	2	**pass**
			T_{32}^{up}	2	**pass**
$RequestService_3$	$RequestService_4$	!*notSubscribed*	T_{33}^{down}	3	**pass**
			T_{34}^{down}	2	**pass**
			T_{35}^{up}	4	**pass**
			T_{36}^{up}	3	**pass**
$OfferService_3$	$OfferService_4$	!*notSubscribed*	T_{37}^{down}	3	**pass**
			T_{38}^{down}	2	**pass**
			T_{39}^{up}	3	**pass**
			T_{40}^{up}	3	**pass**
$SubscribeService_0$	$SubscribeService_4$		T_{41}^{down}	2	**pass**
			T_{42}^{down}	2	**pass**
			T_{43}^{up}	2	**pass**
			T_{44}^{up}	2	**pass**
$UnsubscribeService_0$	$UnsubscribeService_4$		T_{45}^{down}	1	**pass**
			T_{46}^{down}	1	**pass**
			T_{47}^{up}	1	**pass**
			T_{48}^{up}	1	**pass**

Table 15.7: Testing results cont.

of different runs that were made with each test. The numbers reported here are lower that the specified number of tests and test runs because the same tests and test runs sometimes are generated more than once, due to the randomization. The tables also show the verdict of each test. These verdicts are the verdicts of the runs of a test seen together. The verdict **pass** is given if all the test runs gave verdict **pass** and **fail** is given if at least one test run gave the verdict **fail**. The tables also specify which extra events were given to the test generation algorithm.

As an example, the first line of the table 15.6 should be read: Test T_1^{down} was generated from $RequestService_0$ with the "down" option and the events !*perform* and !*needed* as additions to the output alphabet of the diagram. The test was executed against $RequestService_1$ with the "down" option, which resulted in three different runs that all gave the verdict **pass**. Further, line three should be read: Test T_3^{up} was generated from $RequestService_1$ using the "up" option and with no additional events in the output alphabet. It was executed against $RequestService_0$ with the "up" option, which resulted in two runs of which at least one gave the verdict **fail**.

15.3.5 Discussion

When we look at tables 15.6 and 15.7 we observe that the testing of all refinements returned the verdict **pass**, except for the "testing up" of the refinements *RequestSer-*

$vice_0 \rightsquigarrow_r RequestService_1$ and $OfferService_0 \rightsquigarrow_r OfferService_1$.

In the first of these cases we see that some of the test runs of T_3^{up} and T_4^{up} give the verdict **fail**. The actual test runs that give this verdict are the test runs

$$\langle \theta, !request, ?request, !agreed, ?agreed \rangle$$

and

$$\langle !request, ?request, !agreed, ?agreed \rangle$$

These runs both represent a trace that is present (positive) in $RequestService_0$ but is not present (inconclusive) in $RequestService_1$. Hence this is not a correct refinement according to the definition of $\rightsquigarrow_r$.

The case of T_7^{up} and T_8^{up} is analogous. The runs of these tests giving the verdict **fail** are

$$\langle \theta, !offer, ?offer, !perform, ?perform, \theta \rangle$$

and

$$\langle !offer, ?offer, !perform, ?perform, \theta \rangle$$

Both represent a trace that is positive in $OfferService_0$ but inconclusive in $OfferService_1$.

By inspecting the rest of the specification we find that the other tested refinements are indeed correct. We can therefore conclude that the refinement testing gave the expected results, and that the first of the success criteria is fulfilled.

From tables 15.6 and 15.7 we see that for each of the refinements, two tests were generated in the "testing down" and two tests were generated in the "testing up". If we look at the diagrams in the refinement testing part of the study, we observe that they all follow a pattern where the *System* lifeline first receives a message from the frame of the diagram and then transmits one or more messages back to the frame. Stated differently, all diagrams specify that the system receives an input and then does one or more outputs. If we see this characteristic of the diagrams together with the test generation algorithm, we realize that there are only two tests that can be generated from each of the diagrams. We can therefore consider the test generation in this case study to be complete.

The same is not true for test execution. Table 15.8 shows the potential (based on an analysis of the diagrams in the study) and actual test runs of each test. What we observe is that for most, but not all, of the tests, all possible test runs did appear in the test execution. By inspecting the test runs we discover that for tests T_9^{down}, T_{25}^{down}, T_{26}^{down}, T_{30}^{down}, T_{33}^{down}, T_{35}^{up}, T_{36}^{up} an T_{37}^{down} we still had all main branches (i.e. alt operands) covered. This leaves the tests T_{34}^{down} and T_{38}^{down}, where one branch each of the diagrams they were tested against was not covered. These branches were, however, covered by the test runs of tests T_{33}^{down} and T_{37}^{down}, which were part of testing the same refinement as T_{34}^{down} and T_{38}^{down}, respectively.

Based on this analysis of the test runs, we can safely say that the test execution had reasonable coverage of the specification. With complete test generation (i.e. all possible tests were generated) and reasonable coverage in the test execution, we can conclude that we have reasonable coverage of the specification in the refinement testing. This means we also have fulfillment of the second success criterion.

Test	Potential	Actual
T_1^{down}	3	3
T_2^{down}	3	3
T_3^{up}	2	2
T_4^{up}	2	2
T_5^{down}	2	2
T_6^{down}	2	2
T_7^{up}	2	2
T_8^{up}	2	2
T_9^{down}	3	2
T_{10}^{down}	3	3
T_{11}^{up}	3	3
T_{12}^{up}	3	3
T_{13}^{down}	2	2
T_{14}^{down}	2	2
T_{15}^{up}	2	2
T_{16}^{up}	2	2
T_{17}^{down}	1	1
T_{18}^{down}	1	1
T_{19}^{up}	1	1
T_{20}^{up}	1	1
T_{21}^{down}	1	1
T_{22}^{down}	1	1
T_{23}^{up}	1	1
T_{24}^{up}	1	1

Test	Potential	Actual
T_{25}^{down}	5	3
T_{26}^{down}	5	3
T_{27}^{up}	3	3
T_{28}^{up}	3	3
T_{29}^{down}	3	3
T_{30}^{down}	3	2
T_{31}^{up}	2	2
T_{32}^{up}	2	2
T_{33}^{down}	6	3
T_{34}^{down}	6	2
T_{35}^{up}	5	4
T_{36}^{up}	5	3
T_{37}^{down}	4	3
T_{38}^{down}	4	2
T_{39}^{up}	3	3
T_{40}^{up}	3	3
T_{41}^{down}	2	2
T_{42}^{down}	2	2
T_{43}^{up}	2	2
T_{44}^{up}	2	2
T_{45}^{down}	1	1
T_{46}^{down}	1	1
T_{47}^{up}	1	1
T_{48}^{up}	1	1

Table 15.8: Potential and actual test runs

Chapter 16

Related work on testing

Several approaches to generating tests from UML statecharts have been made (see e.g. [66, 93, 136]) as well as other state based formalisms (see e.g. [102, 177–179, 183] and also chapter 12). Several of these have been implemented in tools (e.g. [95, 180]). Characteristic for such approaches is that statecharts and state based formalisms are complete specifications, and further that they are methods for conformance testing of implementations. They can therefore be seen as establishing a relation between design and code, and further as operating on a lower level of abstraction that our sequence diagram based approach. Because of this, they can be seen as being orthogonal to our approach. We do not go any deeper into this, and in the following concentrate on sequence diagram or MSC based approaches.

The UML Testing Profile [133] is an extension of UML that defines a method for specifying tests as sequence diagrams. In [13] a methodology for model driven development with the profile is provided. The profile contains an informal notion of test execution as well as mappings from tests specified with this profile to tests in JUnit and Testing and Test Control Notation (TTCN-3). TTCN-3 is a language for specification of tests in a MSC like notation, as well as a textual notation, and can be seen as a predecessor of the UML Testing Profile (see e.g. [38, 160]). Neither do, however, specify or define test generation, and in this respect our work may be seen as complementary to the UML Testing Profile and TTCN-3.

Different approaches to generating tests from sequence diagrams and MSCs exist. In [167,168] a scheme for generating tests from UML 2.x sequence diagrams is described in an informal manner. The sequence diagrams are allowed to contain the operator neg in order to specify negative cases, but can not contain alt or other high-level operators. The approach is based on also having UML protocol state machines describing the system to obtain information on behavior that is inconclusive in the sequence diagrams.

In [52] a method for executing sequence diagrams as tests is described, and [22] describes a method for extracting tests from sequence diagram specifications. Also in these approaches the level of formality is not very high, and they seem to be mostly experience and heuristic based approaches. Further, they are based on UML 1.x and hence do not deal with high-level operators such as alt, assert and neg.

In [56] test generation from the Specification and Description Language (SDL) [86] with MSCs as test purposes is described, and [12] defines test generation from MSCs. These approaches are more formal than the above mentioned UML sequence diagram approaches, but still they are less general than our approach, because MSCs do not contain the assert and neg operators.

In [103] an approach for generating tests from MSCs is presented that is based on first translating MSCs to finite state machines and then apply other existing methods for test generation from state machines. This approach therefore considers MSCs as complete specifications. Further, the approach simplifies the MSC semantics by interpreting message passing as synchronous and by letting the lifelines synchronize.

Similar to our approach in some aspects is the approach of [143]. Tests are generated from UML models (statecharts, class diagrams and object diagrams) with sequence diagrams as test objectives. The sequence diagrams may characterize acceptance scenarios or rejection scenarios, and the partial nature of sequence diagrams is recognized. Both statecharts and sequence diagrams are translated into LTSs, and the ioco algorithm is used for test generation from these LTSs. The generated tests are then translated back to a sequence diagram like notation.

Also the approach of [126] have some similarities to our work. In this approach, abstract tests expressed as sequence diagrams are generated from a combination of formalized use cases and sequence diagrams. The sequence diagrams are based on UML 1.x and are interpreted with strict sequencing.

Our refinement testing differs from the above presented approaches in several ways. Our test generation is based purely on generating tests from sequence diagrams without any additional information. We stay loyal to the intended semantics of sequence diagrams by interpreting them as partial specifications and with weak sequencing, and we do not make simplifications to the semantics (e.g. by interpreting sequence diagrams as complete specification or with strict sequencing) to simplify the test generation. In addition, our approach is more general as we allow sequence diagrams to contain the operators neg and assert.

The approach of [177–179] and its implementation in the TorX tool [180] can probably be applied to validate ioco conformance between labeled transition systems, even though this is not necessarily the intended use of the theory and the tool. In earlier work [109–111] we have applied testing techniques to investigate refinements of SDL specifications. Except from these, we are not aware of any other approaches where test generation and test execution are applied for investigating the relation between specifications at different levels of abstraction. Our refinement testing is still unique as an approach that is developed specifically for the purpose of testing refinements and that is based on sequence diagrams.

Part V

Availability

Chapter 17

Introduction to availability

This part of the thesis is concerned with developing methods for specifying and analyzing availability based on specifications. An important trend in current system development is the focus on services, cf. service oriented architecture (SOA) (see e.g. [46,130]). We therefore find it natural to apdopt a service oriented view of availability. This works both ways; with a service oriented view on availability we find an application area for our approach to system specification and analysis, but on the other hand a sevice oriented view on system development creates an increasing demand for availability of services. This also fits our overall apporach to system specification, as there is a long tradition of using sequence diagrams and similar notations for specifying services.

In chapter 18 we present a conceptual model for service availability. There we argue that in order to properly specify and analyze availability, the notion of service availability must be decomposed into measurable properties. We view availability as being a collection of exclusivity properties and accessibility properties. Exclusivity properties are closely related to authentication, while accessibility is seen as a combination of timeliness properties and quality properties. While quality is related to quality of service (QoS), timeliness may be seen as a generalization of real-time properties. Within real-time, we may distinguish between hard real-time and soft real-time. Services with hard real-time are characterized by strict deadlines to which they should provided responses, while in services with soft real-time this requirement is relaxed so that reasonable deviations to the deadlines are allowed. What distinguishes soft real-time from hard real-time may be expressed by application of probabilities; where hard real-time specifies a deadline, soft real-time may specify a deadline with an associated probability, i.e., expressing that the deadline must be reached with a certain probability.

While soft real-time may be a more appropriate model for many services, hard real-time is easier to specify and analyze because we do not have to take the probabilities into consideration. In this thesis we concentrate on hard real-time properties, and do not go further into the topics of quality and soft real-time.

In chapter 19 we take a first step towards specifying and analyzing availability properties by extending our operational semantics with support for hard real-time. Then, in section 20, we investigate the possibility of applying our operational semantics extended with data and time to the specification and analysis of availability properties by conducting a case study. Chapter 21 provides related work with respect to specification and analysis of real-time properties.

Chapter 18

A conceptual model for service availability[1]

Availability is an important aspect of today's society. Vital functions as e.g. air traffic control and telecom systems, especially emergency telecommunications services, are totally dependent on available computer systems. The consequences are serious if even parts of such systems are unavailable when their services are needed.

Traditionally, the notion of availability has been defined as the probability that a system is working at time t, and the availability metric has been given by the "uptime" ratio, representing the percentage of time that a system is "up" during its lifetime [148]. This system metric has been applied successfully worldwide for years in the PSTN/-ISDN[2] telephony networks along with failure reporting methodologies [45]. This metric does not sufficiently characterize important aspects of service availability. Service availability needs a more enhanced metric in order to measure availability in a way that meets the demands of today's services which have been shown to have much more bursty patterns of use than traditional PSTN/ISDN services [33].

Indeed, as the environment where services are deployed becomes more and more complex [9] a more fine-grained view on "what is availability" is needed. Several global virus attacks have shown that availability is indeed affected by security breaches, e.g., when e-mail servers are flooded by infected e-mails, the availability for ordinary e-mails decreases. Another example is denial of service (DoS) attacks, for which a service is overloaded with requests with the only purpose of making the service unavailable for other users.

In order to model and analyze the availability of computerized systems in a world where the dependency on and the complexity of such systems are increasing, the traditional notion of availability is no longer sufficient. We present a conceptual model for service availability designed to handle these challenges. The core of this model is a characterization of service availability by means of accessibility properties and exclusivity properties. Further, we seek to preserve well-established definitions from our main sources of inspiration to the extent possible: security, dependability, real-time systems, and quality of service (QoS).

In section 18.1 we provide the basis of our investigation into availability, including different viewpoints on and approaches to availability, and other aspects in the fields

[1] This chapter is based on [149, 150] and represents joint work with Judith E. Y. Rossebø, Atle Refsdal and Knut Eilif Husa.

[2] Public Switched Telephone Network/Integrated Services Digital Network.

of security and dependability. In section 18.2 the properties of service availability are discussed, and in section 18.3 the overall conceptual model is presented. In section 18.4 a summary is provided.

18.1 Characterizing aspects of service availability

The setting for our investigation is derived from the fields of dependability and security, and we therefore strive to conform to the well-established concepts and definitions from these fields where there is a consensus. We also look to different approaches and viewpoints in dependability and security research to motivate and derive a set of requirements for a service availability concept model which enables an augmented treatment of availability that is more suited to securing availability in today's and future services.

18.1.1 Definitions of availability

Availability has been treated by the field of dependability and the field of security. The definitions of availability commonly used in these fields are:

1. Readiness for correct service [11].

2. Ensuring that authorized users have access to information and associated assets when required [80].

3. The property of being accessible and usable on demand by an authorized entity [78, 81].

We find the first of these definitions to be insufficiently constraining for practical application to design of services with high availability requirements. An integral part of securing availability is ensuring that that the service is provided to authorized users; this is not addressed by the first definition. It is, however, addressed by the second, but neither the first nor the second captures the aspect of a service being *usable*. The third definition captures all of these aspects, and therefore is the basis for our development of a more refined availability model.

In order to ensure service availability, it is essential to refine this notion to include addressing the aspect of ensuring that the service is provided to the authorized users *only*. For example, in order to calculate penetration and usage parameters, the total number of authorized users that are expected to access the service at a given time must be known. This is important to prevent the service from being overloaded.

As already argued, there is a need to provide an enhanced classification and model of service availability in order to thoroughly analyze and enable the rigorous treatment of availability throughout the design process depending on the requirements of the individual services.

18.1.2 Attributes, means and threats

The IFIP WG 10.4 view on dependability is provided in [11]. Figure 18.1 shows their concept model of dependability.

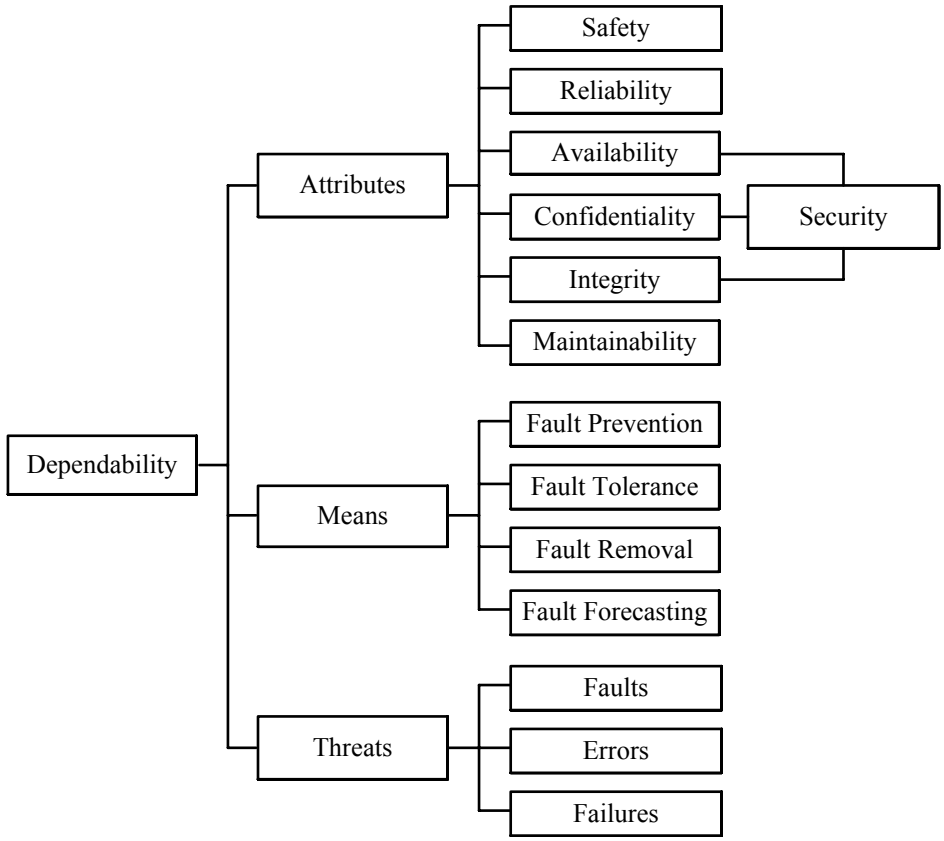


Figure 18.1: Conceptual model of dependability

This conceptual model of dependability consists of three parts: the *attributes* of, the *threats* to and the *means* by which dependability is attained. This is a nice approach which motivates us to use a similar approach in our classification of service availability. Clearly, threats to availability such as denial of service, and means to availability such as applying redundancy dimensioning techniques, have an important place in availability analysis.

In [11], the *means* by which dependability can be attained are fault prevention, fault tolerance, fault removal and fault forecasting. *Fault prevention*: how to prevent introduction of faults. *Fault tolerance*: how to deliver correct service in the presence of faults. *Fault removal*: how to reduce the number of faults, and finally *fault forecasting*: how to estimate the present number, future incidents and likely consequences of faults.

This approach does not address all of the means by which service availability can be obtained. This is because incidents resulting in loss of service availability do not necessarily transpire due to faults and therefore classification of means in terms of faults as in [11] is, in our view, insufficient for availability analysis. An example is the hijacking of user sessions by an attacker or group of attackers, preventing the authorized user or group of users from accessing the service. This incident results in loss of service availability for a set of users, without incurring a fault in the system.

An *unwanted incident* is defined in [171] as an incident such as loss of confidentiality, integrity and/or availability. A fault is an example of an unwanted incident. Therefore, in order to classify threats to availability and means to achieve availability in a security setting, we are also motivated by the approach used in the security field of risk analysis and risk management as in [40, 112].

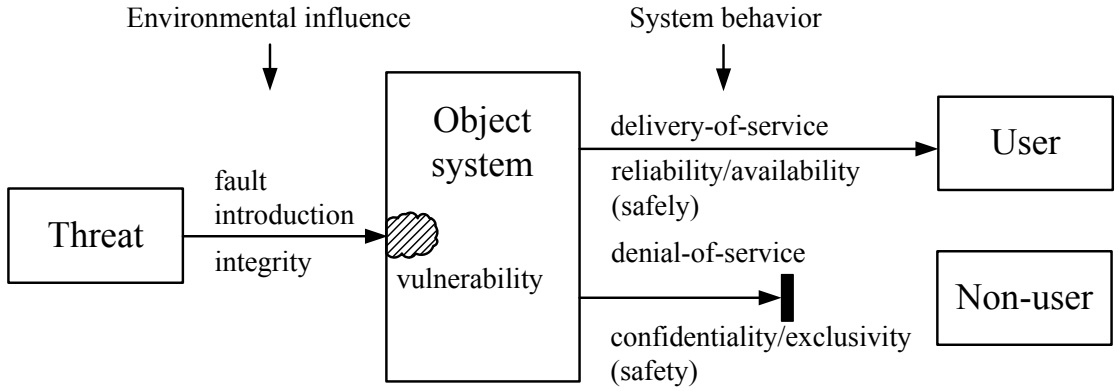


Figure 18.2: Jonsson's conceptual model

In [11], the *threats* to dependability are defined as faults, errors and failures, and these are seen as a causal chain of threats:

fault $\longrightarrow$ error $\longrightarrow$ failure

This understanding of threats serves nicely in the dependability model, however, we use the definition of threat as defined in [81]: a *threat* is a potential cause of an unwanted event, which may result in harm to a system or organization and its assets. Unlike [11], we do not consider such causal chains alone as the sole threats to availability, as service availability may be reduced by e.g. a denial of service (DoS) attack which reduces the service availability without causing a fault, error, or failure to the actual service itself.

18.1.3 Viewpoints

For our availability analysis, it is appropriate to evaluate whether we should consider a system from a black box or white box perspective. In [90], Jonsson provides a conceptual model for security/dependability with a black box view as shown in figure 18.2.

In this system model view, Jonsson considers availability to be a purely behavioral aspect related to the outputs of the system, solely with respect to the users. As can been deduced from figure 18.2, exclusivity is a means to ensure availability. This viewpoint is valid and useful for some aspects of availability analysis; however, we see the need for evaluating availability from other viewpoints as well. Availability aspects of the internal components of the system must also be analyzed.

We claim that aspects of availability must indeed be observed from both the input side and the output side as well as the internal components of the system. For example, denial of service attacks can be observed as malicious input to a system to flood the system.

It is also important to observe and analyze the internal behavior in the system in order to analyze the availability aspects of components, in particular service components which collaborate to deliver the service. Motivated by a service-oriented system view, only a white box view allows and facilitates the specification of the internal means to achieve availability and the examination of internal causes that affect availability.

Figure 18.3 summarizes the different concerns for analyzing availability. From the point of view of the user, the service is either available, or it is not. The system view is well understood in the dependability field, and as discussed above, Jonsson provides an evaluation from a system viewpoint and with a security point of view.

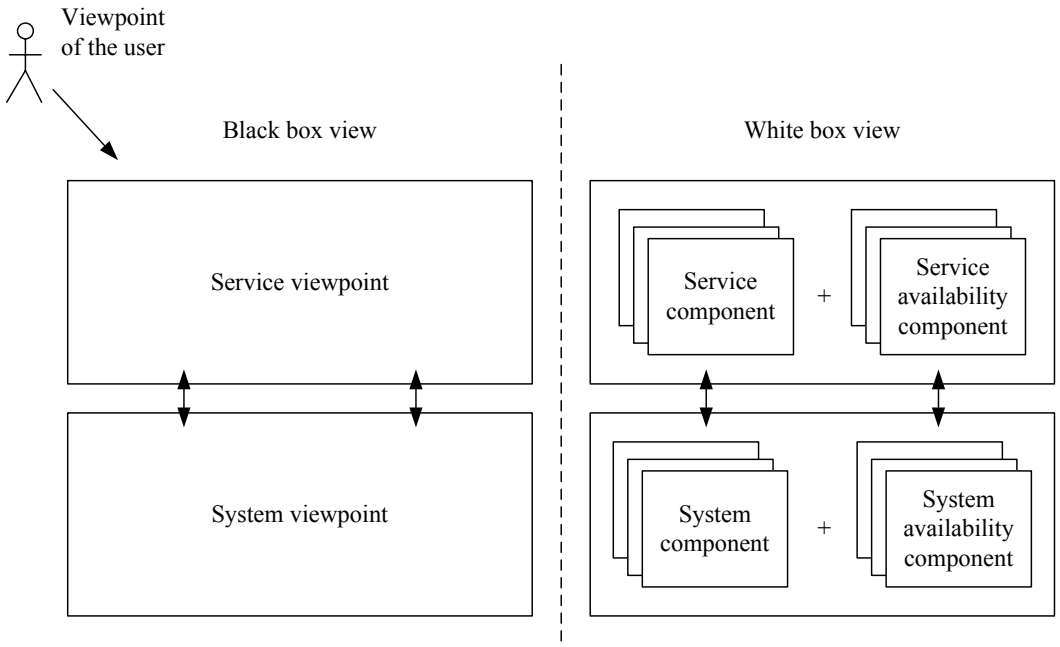


Figure 18.3: Viewpoints for analyzing availability

The Service Availability Forum (SAF) is working on standardizing middleware for the open interfaces between the layers [162], see figure 18.4. In our work on securing availability in service composition, we are analyzing availability from the decomposed service viewpoint, according to requirements of the users.

18.1.4 Diversity of services

In the current and future telecommunications market, there are many different types of services each of which may have different requirements with respect to availability. Telephony services, and in particular, emergency services, are examples of services with stringent availability requirements. Internet-based services, however, have somewhat different requirements. Requirements for what may be tolerated of delays or timing out of services are rather lax currently for e.g., online newspaper services.

For traditional telecommunications services, the requirement of $99,999\%$ availability is still valid, however, it does not sufficiently address all of the differentiated requirements with respect to service availability. More precisely, as advocated by SAF, there is also a need for a customer centric approach to defining availability requirements. The availability concern of the Service Availability Forum is readiness for correct service and in particular continuity of service, with a focus on the demands of the customers. Service availability as defined by the Service Availability Forum aims to meet the following demands:

- Customers demand and expect continuous service availability.
- Customers want always-on services and connections that are maintained without disruption – regardless of hardware, software, or operator-caused system faults or failures.

The availability concern of the Service Availability Forum is readiness for correct service and in particular continuity of service, with a focus on the demands of the customers.

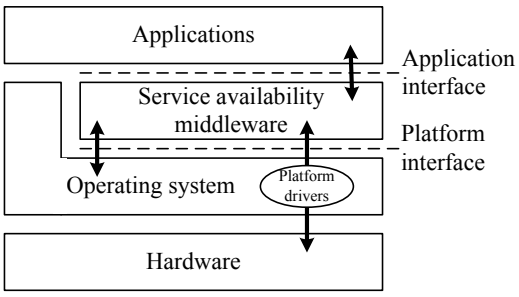


Figure 18.4: The Service Availability Forum framework

The Service Availability Forum is concerned with availability of today's systems from the dependability perspective providing a transition from the application of dependability to traditional telecommunications systems to current systems which are distributed.

We incorporate the ideas of the Service Availability Forum in our model, to enable customer oriented availability requirements, however, extending these to include the aspects of ensuring that unauthorized users cannot interrupt, hijack, or prevent the authorized users from accessing a service.

18.1.5 Measures

As discussed above, we need a more fine grained measure of availability than pure "up" or "down". Services can exist in numerous degraded but operational/usable/functional states between "up" and "down" or "correct" and "incorrect". It should be possible to describe various states of availability in order to specify just how much a reduction of service quality may be tolerated.

While both the Common Criteria [79] and Johnson [90] define security measures and provide techniques for measuring security in general, there is a need for a more fine grained metric for measuring service availability that takes into account, for example, measurement of how well user requirements are fulfilled, as well as a need for measuring the ability to adequately provision a service to all of the authorized users requiring the service at a given moment. Such a metric needs to take into account the appropriate set of parameters, not just the usual average based on the mean time to failure (MTTF) and the mean time to repair (MTTR).

18.2 Properties of service availability

We claim that service availability encompasses both exclusivity, the property of being able to ensure access to authorized users only, and accessibility, the property of being at hand and usable when needed. As such, contrary to, e.g. [11], which treats availability as an atomic property, we see service availability as a composite notion consisting of the following aspects:

- Exclusivity
- Accessibility

We elaborate on these two properties in the following sub-sections.

18.2.1 Exclusivity

By *exclusivity* we mean the ability to ensure access for authorized users only. More specifically, this involves ensuring that unauthorized users cannot interrupt, hijack, or prevent the authorized users from accessing a service. This aspect is essential to prevent the denial of legitimate access to systems and services. That is, to focus on prohibiting unauthorized users from interrupting, or preventing authorized users from accessing services. Our definition of exclusivity involves both users and non-users, i.e., ensuring access to users while keeping unauthorized users out. This is in order to properly address means of achieving exclusivity some of which will address ensuring access for authorized users and others will address techniques for preventing unauthorized users from accessing or interrupting services.

The goal with respect to exclusivity is to secure access to services for authorized users in the best possible way. Essentially this means:

- Secure access to services for the authorized users.
- Provide denial of service defence mechanisms. Here we focus on prohibiting unauthorized users from interrupting, or preventing users from accessing services.
- Ensure that unauthorized users do not gain access to services.

Note that attacks via covert channels or by eavesdropping can lead to loss of confidentiality without loss of exclusivity as the attacker is not accessing the service, but passively listening in on service activity. Confidentiality, however, consists of exclusivity and absence of unauthorized disclosure of information.

18.2.2 Accessibility

We define *accessibility* as the quality of being at hand and usable when needed. The notion of "service" is rather general, and what defines the correctness of a service may differ widely between different kinds of services. Accessibility is related to QoS [14, 135, 187], but what is considered relevant qualities vary from one domain to another. Furthermore, QoS parameters tend to be technology dependent. An example of this is properties like video resolution and frame rates [187], which are clearly relevant for IP-based multimedia services and clearly not relevant in other service domains, such as the Short Message Service (SMS) or instant messaging services.

What all services do seem to have in common is the requirement of being timely; for a service to be accessible it must give the required response within reasonable time. In addition to being timely, a service will be required to perform with some quality to be usable. Hence, we divide accessibility properties into two major classes of properties: *timeliness* properties and *quality* properties. Timeliness is the ability of a service to perform its required functions and provide its required responses within specified time limits. A service's quality is a measure of its correctness and/or how usable it is.

Consider an online booking service. From the viewpoint of a user at a given point in time, we could say that the quality of the service is either 1 or 0 depending on whether the user gets a useful reply (e.g. confirmation) or unuseful reply (e.g. timeout). (Over time this can be aggregated to percentages expressing how often one of the two kinds of responses will be given.)

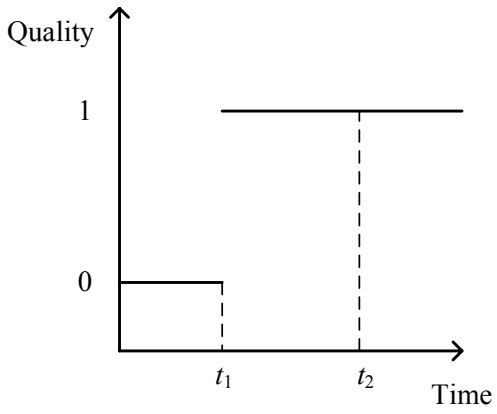


Figure 18.5: Quality vs. timeliness

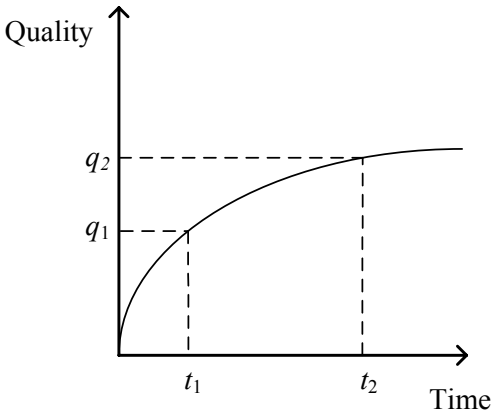


Figure 18.6: Quality vs. timeliness

In a multimedia service like video streaming, the frame rate may be seen as a timeliness property (each frame should be timely) while the resolution of each frame and the color depth are quality properties.

In both these examples we may see a dependency between timeliness and quality. In the first example (figure 18.5) we may assume a deadline t_2 for the response to the user for the service to be accessible. However, we must also assume some processing time t_1 for the service to be able to produce an answer. This means that the quality requirement enforces a lower bound on the timeliness; if the deadline is too short the user will always receive the timeout message. In other words we must have that $t_1 < t_2$ for the service to be accessible.

In the other example (figure 18.6) we may assume that higher quality requires more processing time per frame. This means that a required quality q_1 provides a lower limit t_1 on the processing time of each frame. Further, to get the required frame rate there must be a deadline t_2 for each frame, which provide an upper bound q_2 on the quality. This means the service must stay between this lower and upper bound to be accessible. This approach may be seen as an elaboration of Meyer's concept of *performability evaluation* [124].

These considerations motivate a notion of *service degradation*. We define service degradation to be reduction of service accessibility. Analogous to accessibility we decompose service degradation into timeliness degradation and quality degradation, and see that these are quantities mutually dependent on each other. For example, graceful degradation in timeliness may be a way of avoiding quality degradation if resources are limited, or the other way around. A combination of graceful degradation in timeliness and graceful degradation in quality may also be applied. Related to QoS, accessibility may actually be considered a QoS tolerance cut-off, i.e., the point at which the QoS deteriorates to a level where the service is deemed no longer usable, so that the service is considered unavailable.

18.3 The conceptual model

Based on the above discussion we propose the overall model presented in figure 18.7 (presented as a UML class diagram) and further explained in the following text.

In the figure the relationships between availability, threats and means are shown.

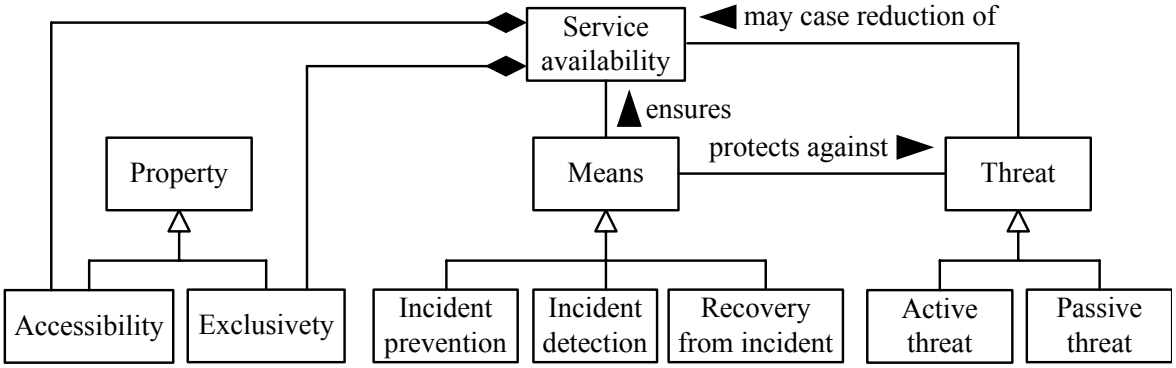


Figure 18.7: The overall picture

Availability is affected by means and threats. Means ensures availability by preventing and countering unwanted incidents and protects against threats. Threats may lead to unwanted incidents causing reduction of availability.

There are many different types of services, and they may have different requirements with respect to availability. Availability requirements should be flexible enough to address the different services consistently. We propose that availability is specified by the means of availability policies and predicates over measurable properties of services, and that these policies and predicates are decomposed in accordance to the decomposition of availability in the conceptual model. An availability policy consists of an accessibility policy (e.g., required resources) and an exclusivity policy (e.g., which entities have permissions to use the service or system).

18.4 Summary

In this chapter we have presented a conceptual model for service availability that takes into account a much broader spectrum of aspects that influence availability than previously addressed by work in this area. We have argued that exclusivity is an aspect of availability that has been generally neglected in the literature, and have shown where it fits in an enhanced notion of service availability. Further we have shown how QoS, real-time and dependability considerations may be integrated in the model and treated as accessibility properties.

Our conceptual model for availability embraces both a white box view as well as a black box view of availability and, hence, addresses both internal and external concerns of availability. The need for this is apparent in our current work on ensuring availability in service composition that encompasses a collaboration of roles, which are slices of behavior across distributed systems. These must be composed correctly in order to achieve a service with the required availability.

The model also establishes the relationship between threats, means and availability properties. Together these elements provide a framework in which all relevant views and considerations of availability may be integrated, and a complete picture of service availability may be drawn.

Chapter 19

Towards analyzing availability

This chapter takes a first step towards specifying and analyzing availability properties by extending our operational semantics with support for hard real-time. As explained below we choose an approach where time is treated as data. This means that we start by extending the operational semantics with data handling, and then apply the data handling mechanisms to extend the operational semantics with time. Data handling is presented in section 19.1 and time in section 19.2.

19.1 Operational semantics with data and variables

In this section we present an extension of the operational semantics with data and variables. Though inspired by STAIRS with data [156], there are certain differences. These differences are explained in section 21.1.1.

We find that the graphical syntax suggested by the UML standard is somewhat sparse for a consistent treatment of data. Imagine you want to specify a scenario where a lifeline l sends a value (e.g. 5) to a lifeline k, lifeline k squares the value and sends the result back to l, and l then performs a check of whether or not it received the square of the value it sent. This could be specified by the diagram $sqr5$ in figure 19.1. The problem with this diagram is that it lacks bindings between the data present in its various events. While this may be OK in a declarative setting we find this unsatisfactory when defining an operational semantics; we want to actually execute the diagram and

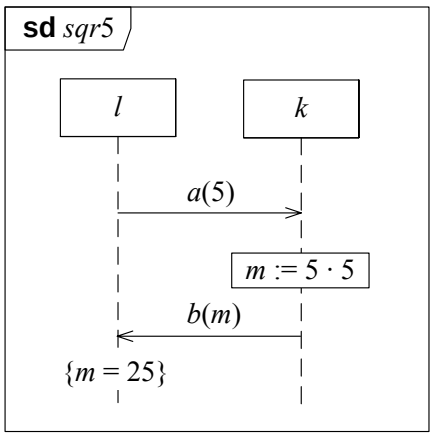


Figure 19.1: Diagram with data

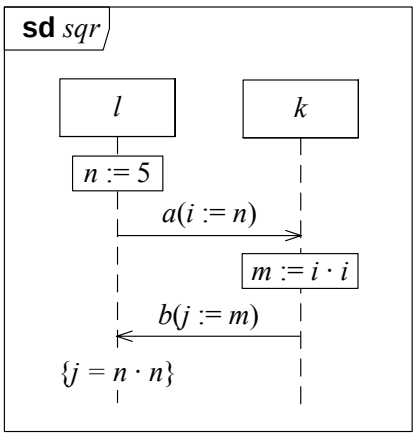


Figure 19.2: Diagram with data

perform the calculations. We therefore propose a more detailed notation, as exemplified by the diagram *sqr* in figure 19.2. In this diagram all bindings are explicit and we have the information necessary to do the calculations specified in the diagram.

All details of the textual syntax are given in section 19.1.2, after we have presented some necessary preliminaries on data and variables in section 19.1.1. In section 19.1.3 the extended operational semantics with data is presented, and in section 19.1.4 we provide a treatment of guarded choices.

19.1.1 Data and variables

We assume common data types such as Booleans, natural numbers, integers, reals, strings, etc. and use common operators for these types when building expressions. Further, we allow any data type as long as an axiomatic definition is provided, but we will not provide such definitions. In order to keep things simple we let the types be implicit and assume that all expressions are correct with respect to syntax and type.

Let *Var*, *Val* and *Expr* denote the sets of variables, values (constants) and expressions, respectively. Obviously we have that $Var \subset Expr$ and $Val \subset Expr$. Further let $var \in Expr \to \mathbb{P}(Var)$ be a function that returns the free variables of an expression. An expression $x \in Expr$ is closed if $var(x) = \emptyset$. We let *CExpr* denote the set of closed expressions, defined as:

$$CExpr \stackrel{\mathsf{def}}{=} \{x \in Expr \mid var(x) = \emptyset\}$$

We assume the existens of a function $eval \in CExpr \to Val$ that evaluates any closed expression to a value.

Let $\sigma \in Var \to Expr$ be a (partial) mapping from variables to expressions. We denote such a mapping $\sigma = \{v_1 \mapsto x_1, v_2 \mapsto x_2, \ldots, v_n \mapsto x_n\}$ for $v_1, v_2, \ldots, v_n \in Var$ and $x_1, x_2, \ldots, x_n \in Expr$. If $x_1, x_2, \ldots, x_n \in Val$ we call it a data state. We let Σ denote the set of all mappings and $\widehat{\Sigma}$ the set of all data states.

The empty mapping is denoted by $\emptyset$. $dom.\sigma$ denotes the domain of σ, i.e.:

$$dom.\{v_1 \mapsto x_1, v_2 \mapsto x_2, \ldots, v_n \mapsto x_n\} \stackrel{\mathsf{def}}{=} \{v_1, v_2, \ldots, v_n\}$$

For every $\sigma \in \Sigma$ we assert that $\#\sigma = \#(dom.\sigma)$. $\sigma[v \mapsto x]$ is the mapping σ except that it maps v to x, i.e.:

$$\{v_1 \mapsto x_1, \ldots, v_n \mapsto x_n\}[v \mapsto x] \stackrel{\mathsf{def}}{=} \begin{cases} \{v_1 \mapsto x_1, \ldots, v_n \mapsto x_n, v \mapsto x\} \\ \quad \textbf{if } v \neq v_i \textbf{ for all } i \in \{1, \ldots, n\} \\ \{v_1 \mapsto x_1, \ldots, v_i \mapsto x, \ldots v_n \mapsto x_n\} \\ \quad \textbf{if } v = v_i \textbf{ for some } i \in \{1, \ldots, n\} \end{cases}$$

We generalize $\sigma[v \mapsto x]$ to $\sigma[\sigma']$ in the following way:

$$\sigma[\{v_1 \mapsto x_1, v_2 \mapsto x_2, \ldots, v_n \mapsto x_n\}] \stackrel{\mathsf{def}}{=} \sigma[v_1 \mapsto x_1][v_2 \mapsto x_2] \cdots [v_n \mapsto x_n]$$

$\sigma(v)$ returns the expression that v maps to, and is only defined for $v \in dom.\sigma$, i.e.:

$$\{v_1 \mapsto x_1, \ldots, v_i \mapsto x_i, \ldots v_n \mapsto x_n\}(v_i) \stackrel{\mathsf{def}}{=} x_i$$

$x\{\sigma\}$ denotes the expression obtained from x by simultaneously substituting the free variables of x with the expressions that these variables map to in σ. For example

we have that $(y+z)\{\{y \mapsto 1, z \mapsto 2\}\} = 1+2$. In addition we let $\sigma'\{\sigma\}$ be the mapping where this process is applied to all expressions in the range of σ', i.e.:

$$\{v_1 \mapsto x_1, v_2 \mapsto x_2, \ldots, v_n \mapsto x_n\}\{\sigma\} \stackrel{\mathsf{def}}{=} \{v_1 \mapsto x_1\{\sigma\}, v_2 \mapsto x_2\{\sigma\}, \ldots, v_n \mapsto x_n\{\sigma\}\}$$

We also overload the function *eval* to evaluate all expressions in range of a mapping only containing closed expressions:

$$\mathit{eval}(\{v_1 \mapsto x_1, \ldots, v_n \mapsto x_n\}) \stackrel{\mathsf{def}}{=} \{v_1 \mapsto \mathit{eval}(x_1), \ldots, v_n \mapsto \mathit{eval}(x_n)\}$$

Similar to the mappings from variables to expressions we also operate with mappings $\Delta \in \mathcal{L} \to \widehat{\Sigma}$ from lifelines to data states. On these mappings we define operations $\Delta[l \mapsto \sigma]$, $\Delta(l)$ and $\mathit{dom}.\Delta$ identical to the definition of $\sigma[v \mapsto x]$, $\sigma(v)$ and $\mathit{dom}.\sigma$ given above.

19.1.2 Syntax

In our approach lifelines are non-synchronizing. In recognition of lifelines as independent entities, and messages as the only means of communication, we assume all variables to be local to a lifeline. Assignments to variables are events belonging to lifelines, and may also occur within parameterized messages. Formally, a variable of a message is a local variable in both the transmitter and the receiver lifeline of the message. This means that, for example in the diagram *sqr* of figure 19.2 above, both lifeline l and lifeline k have a local variable i because the message a has a parameter i, but the i of l and the i of k are not the same variable.

In the syntax defined in section 5.2, the set of signals $\mathcal{S}$ is merely a set of labels. In order to add parameters to messages we let $\mathcal{N}$ denote the set of all signal labels, and define the set of signals to be:

$$\mathcal{S} \stackrel{\mathsf{def}}{=} \mathcal{N} \times \Sigma$$

A signal (an element of $\mathcal{S}$) is then a pair (s, σ). This way, all other definitions related to messages carry over. Syntactically we represent a signal (s, σ) as $s(\sigma)$.

We extend the definition of sequence diagrams (see page 29) with two new syntactic constructs, assign and constr, to represent assignments and constraints. Let $\mathcal{A}$ and $\mathcal{C}$ denote the set of assignments and constraints, respectively. $\mathcal{A}$ and $\mathcal{C}$ are defined as

$$\begin{aligned} v \in \mathit{Var} \wedge x \in \mathit{Expr} \wedge l \in \mathcal{L} &\Rightarrow \mathsf{assign}(v, x, l) \in \mathcal{A} \\ b \in \mathit{Expr} \wedge l \in \mathcal{L} &\Rightarrow \mathsf{constr}(b, l) \in \mathcal{C} \end{aligned}$$

where we assume b in $\mathsf{constr}(b, l)$ to be a Boolean expression. We let $\mathcal{A} \subset \mathcal{D}$ and $\mathcal{C} \subset \mathcal{D}$. Note that $\mathcal{A}$, $\mathcal{C}$, $\mathcal{E}$ and $\mathcal{T}$ are pairwise disjoint. We let the function $l.\_$ also range over assignments and constraints:

$$\begin{aligned} l.\mathsf{assign}(v, x, l) &\stackrel{\mathsf{def}}{=} l \\ l.\mathsf{constr}(x, l) &\stackrel{\mathsf{def}}{=} l \end{aligned}$$

With these new syntactical constructs, the textual representation of the diagram

sqr of figure 19.2 becomes:

$$\begin{array}{rcl} sqr & = & \mathsf{assign}(n, 5, l)\ \mathsf{seq} \\ & & (!, (a(\{i \mapsto n\}), l, k))\ \mathsf{seq} \\ & & (?, (a(\{i \mapsto n\}), l, k))\ \mathsf{seq} \\ & & \mathsf{assign}(m, i \cdot i, k)\ \mathsf{seq} \\ & & (!, (b(\{j \mapsto m\}), k, l))\ \mathsf{seq} \\ & & (?, (b(\{j \mapsto m\}), k, l))\ \mathsf{seq} \\ & & \mathsf{constr}(j = n \cdot n, l) \end{array}$$

We let the function *var* be overloaded to also range over diagrams in such a way that it returns the variables manipulated or accessed by a diagram:

$$\begin{array}{rcl} var((!, (s(\varphi), tr, re))) & \stackrel{\mathsf{def}}{=} & \bigcup_{v \in dom.\varphi}(\{v\} \cup var(\varphi(v))) \\ var((?, (s(\varphi), tr, re))) & \stackrel{\mathsf{def}}{=} & dom.\varphi \\ var(\mathsf{assign}(v, x, l)) & \stackrel{\mathsf{def}}{=} & \{v\} \cup var(x) \\ var(\mathsf{constr}(x, l)) & \stackrel{\mathsf{def}}{=} & var(x) \\ var(\mathsf{skip}) & \stackrel{\mathsf{def}}{=} & \emptyset \\ var(\mathsf{op}\ d) & \stackrel{\mathsf{def}}{=} & var(d),\ \mathsf{op} \in \{\mathsf{neg}, \mathsf{refuse}, \mathsf{opt}, \mathsf{assert}, \mathsf{loop}\} \\ var(d_1\ \mathsf{op}\ d_2) & \stackrel{\mathsf{def}}{=} & var(d_1) \cup var(d_2),\ \mathsf{op} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{strict}, \mathsf{alt}, \mathsf{xalt}\} \end{array}$$

There is an asymmetry in the definition of the variables of transmit events and receive events. The reason for this is that we assume the expressions in the parameters of messages to be evaluated at the transmission point, and therefore the variables of these expressions are not accessed by the receiver lifeline.

We let $\pi\_(\_) \in \mathcal{L} \times \mathcal{D} \to \mathcal{D}$ be a function that projects a diagram on a given lifeline such that, e.g.:

$$\begin{array}{rcl} \pi_l(sqr) & = & \mathsf{assign}(n, 5, l)\ \mathsf{seq} \\ & & (!, (a(\{i \mapsto n\}), l, k))\ \mathsf{seq} \\ & & (?, (b(\{j \mapsto m\}), k, l))\ \mathsf{seq} \\ & & \mathsf{constr}(j = n \cdot n, l) \\ \pi_k(sqr) & = & (?, (a(\{i \mapsto n\}), l, k))\ \mathsf{seq} \\ & & \mathsf{assign}(m, i \cdot i, k)\ \mathsf{seq} \\ & & (!, (b(\{j \mapsto m\}), k, l)) \end{array}$$

We then have:

$$\begin{array}{rcl} var(\pi_l(sqr)) & = & \{n, i, j\} \\ var(\pi_k(sqr)) & = & \{m, i, j\} \end{array}$$

This provides an unambiguous assignment of the variables visible in a diagram to lifelines.

19.1.3 Semantics

The execution system of the operational semantics provides us with execution states, and this is something we make use of in our handling of data. We extend the execution states to also hold mappings from lifelines to data states:

$$[\beta, d, \Delta] \in \mathcal{B} \times \mathcal{D} \times (\mathcal{L} \to \widehat{\Sigma})$$

Δ contains a data state for each lifeline in d. In the initial state of an execution, each of these data states should map all variables of the lifeline to values, i.e. all variables should be defined at the start of the execution. The set of possible initial mappings from lifelines to data states, given a diagram d, can then be expressed by the following function:

$$initData(d) \stackrel{\mathsf{def}}{=} \{\Delta \in \mathcal{L} \to \widehat{\Sigma} \mid \forall l \in ll.d : \Delta(l) \in \widehat{\Sigma} \wedge var(\pi_l(d)) \subseteq dom.(\Delta(l))\}$$

The rules of the projection system remain unchanged, with the exception of two new rules for handling assignments and constraints:

$$\Pi(L, \beta, a) \xrightarrow{a} \Pi(L, \beta, \mathsf{skip}) \textbf{ if } a \in \mathcal{A} \wedge l.a \in L \tag{19.1}$$

$$\Pi(L, \beta, c) \xrightarrow{c} \Pi(L, \beta, \mathsf{skip}) \textbf{ if } c \in \mathcal{C} \wedge l.c \in L \tag{19.2}$$

The rules of the execution system, on the other hand, are redefined to handle the data states, the assignments and the constraints. Rule (8.6) for executing silent events does not really change, but is adapted to the extended execution states:

$$[\beta, d, \Delta] \xrightarrow{\tau} [\beta, d', \Delta] \textbf{ if } \Pi(ll.d, \beta, d) \xrightarrow{\tau} \Pi(ll.d, \beta, d') \wedge \tau \in \mathcal{T} \tag{19.3}$$

For executing assignments and constraints we define separate rules and special events. For assignments we introduce the special event $write(\Delta, \Delta')$ where Δ and Δ' are the data states of the lifelines before and after the assignment. The rule for assignment is defined as:

$$[\beta, d, \Delta] \xrightarrow{write(\Delta,\Delta')} [\beta, d', \Delta'] \textbf{ if } \Pi(ll.d, \beta, d) \xrightarrow{\mathsf{assign}(v,x,l)} \Pi(ll.d, \beta, d') \tag{19.4}$$

with

$$\begin{array}{rcl} \sigma & = & \Delta(l) \\ \Delta' & = & \Delta[l \mapsto \sigma[v \mapsto eval(x\{\sigma\})]] \end{array}$$

What happens is that values from σ, the data state of lifeline l, are substituted for the free variables in x, and x is then evaluated to a value. This value then replaces the old value of v in the data state, and the new data state replaces the old data state of l.

The *write* event may be seen as superfluous since the execution states contain data states. When we still produce this event it is for two reasons: 1) It provides us with the possibility of doing analysis based on the produced traces alone. 2) It makes the connection to the denotational semantics stronger, since the denotational semantics with data operates with this kind of *write* events.

For constraints we introduce the special event $check(b)$ with $b \in \mathbb{B}$. The rule is defined as:

$$[\beta, d, \Delta] \xrightarrow{check(b)} [\beta, d', \Delta] \textbf{ if } \Pi(ll.d, \beta, d) \xrightarrow{\mathsf{constr}(x,l)} \Pi(ll.d, \beta, d') \tag{19.5}$$

with

$$\begin{array}{rcl} \sigma & = & \Delta(l) \\ b & = & eval(x\{\sigma\}) \end{array}$$

I.e., σ is the data state of lifeline l, and values from σ are substituted for free variables in x, and then x is evaluated to the Boolean value b. The event $check(\mathbf{true})$ is produced if the constraint evaluates to true in σ and $check(\mathbf{false})$ if the constraint evaluates to

false. This is a property we may use to distinguish positive from negative traces with a suitable meta-level, i.e. a meta-level that treats the occurrence of $check(\mathbf{false})$ in the same way as an occurrence of τ_{refuse}.

The rule for executing events is divided into two rules; one for transmit events and one for receive events. When a transmit event occurs, the expressions representing the parameters of the event's message is evaluated with the values of the data state of the transmitter lifeline. The evaluated data of the message is then used to update the data state of the lifeline:

$$[\beta, d, \Delta] \xrightarrow{e} [update(\beta, e), d', \Delta'] \textbf{ if } \Pi(ll.d, \beta, d) \xrightarrow{(!,(s(\varphi),tr,re))} \Pi(ll.d, \beta, d') \qquad (19.6)$$

where

$$\begin{aligned} \sigma &= \Delta(tr) \\ \varphi' &= eval(\varphi\{\sigma\}) \\ e &= (!, (s(\varphi'), tr, re)) \\ \Delta' &= \Delta[tr \mapsto \sigma[\varphi']] \end{aligned}$$

This means that when the message is transmitted it's variables no longer maps to expressions, but to values, and these are the values that the message contains when it is placed in the communication medium.

When a receive event is executed we would like this event's message to contain the evaluated values. Therefore we include in the set of functions that manipulate the communication medium a function $getData \in \mathcal{B} \times \mathcal{M} \to \widehat{\Sigma}$ that retrieves the data from a message in the communication medium. As with the other functions that manipulate the communication medium (add, rm and $ready$), this function must be defined specifically for each communication model.

In the rule for executing a receive event $(?, m)$, we retrieve the parameter values of the message m from the communication medium and place them in the event instead of the expressions we get from the syntactical representation of the event. The retrieved parameter values are also used to update the data state of the receiver lifeline. The rule is hence:

$$[\beta, d, \Delta] \xrightarrow{e} [update(\beta, e), d', \Delta] \textbf{ if } \Pi(ll.d, \beta, d) \xrightarrow{(?,(s(\varphi),tr,re))} \Pi(ll.d, \beta, d') \qquad (19.7)$$

where

$$\begin{aligned} \sigma &= \Delta(re) \\ \varphi' &= getData(\beta, (s(\varphi), tr, re)) \\ e &= (?, (s(\varphi'), tr, re)) \\ \Delta' &= \Delta[re \mapsto \sigma[\varphi']] \end{aligned}$$

19.1.4 Guards

In the UML standard, the choice operator alt may be guarded, as illustrated in figure 19.3. Following [158] we also allow the xalt operator to have guards, as in figure 19.4.

For both alt and xalt we choose the same approach as [158] and view guarded choices as special applications of the constraint construct. There is, however, a subtle difference in the interpretation of alt and xalt; in a guarded alt, execution is allowed to skip the alt altogether if none of the guards of the operands evaluate to true, while in a guarded xalt this is not the case. The rationale for this with respect to alt is to stay close to the UML standard. The reason for not having the option of skipping an xalt

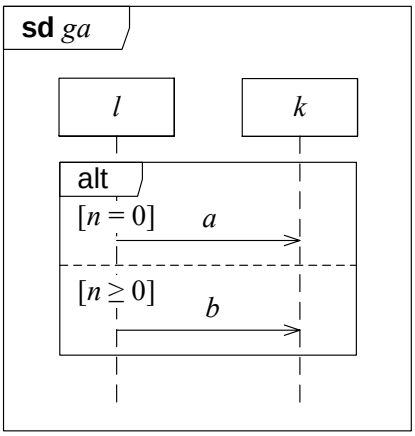


Figure 19.3: Guarded alt

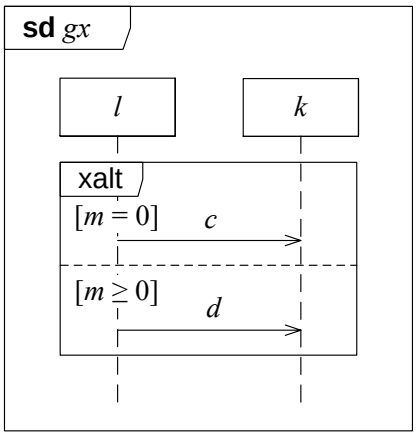


Figure 19.4: Guarded xalt

when all guards evaluate to false is that xalts are used for specifying choices that must be present in an implementation, and it is then not desirable to have implicit operands of the operator.

We do not introduce any special textual syntax for guarded choices. Hence, the textual syntax of the two diagrams is:

$$\begin{aligned} ga &= (\mathsf{constr}(n = 0, l)\ \mathsf{seq}\ !a\ \mathsf{seq}\ ?a)\ \mathsf{alt} \\ &\quad (\mathsf{constr}(n \geq 0, l)\ \mathsf{seq}\ !b\ \mathsf{seq}\ ?b)\ \mathsf{alt} \\ &\quad \mathsf{constr}(\neg(n = 0 \vee n \geq 0), l) \end{aligned}$$

$$\begin{aligned} gx &= (\mathsf{constr}(m = 0, l)\ \mathsf{seq}\ !c\ \mathsf{seq}\ ?c)\ \mathsf{xalt} \\ &\quad (\mathsf{constr}(m \geq 0, l)\ \mathsf{seq}\ !d\ \mathsf{seq}\ ?d) \end{aligned}$$

The operational semantics has already been defined since guarded choices are just macros on top of the syntactic constructs already dealt with above.

What should be noted is that we allow guards of different operands of the same choice to be true at the same time. Further, we do consider the lifeline a guard is placed on as relevant, and therefore assign the constraint representing a guard to the lifeline the guard in placed on in the diagram.

19.2 Operational semantics with time

In this section we show how the operational semantics with data defined above may be extended with time. The same way as the operational semantics with data is inspired by STAIRS with data, the operational semantics with time is inspired by timed STAIRS [70]. Though inspired by timed STAIRS, there are also differences. These differences are discussed in section 21.1.2.

Section 19.2.1 introduces our handling of time, while section 19.2.2 provides our approach to timestamps of events. In section 19.2.3 we provide an example of how our time and data enhanced sequence diagrams can be applied.

19.2.1 Time

In order to model time we introduce a special variable $\boldsymbol{now} \in \mathit{Var}$.[1] Because we only have local variables, this variable is placed in the data state of every lifeline in a diagram. We do, however, consider it a global variable in the sense that we make sure that all the local $\boldsymbol{now}$ variables are updated simultaneously and with equal increments, i.e. that the time of all lifelines are synchronized. Except for increments by a special tick rule, the $\boldsymbol{now}$ variables are read only, something that may be ensured by the following syntax constraint:

$$v = \boldsymbol{now} \Rightarrow \mathsf{assign}(v, x, l) \notin \mathcal{D}.$$

Because $\boldsymbol{now}$ is a variable in the data state of each lifeline of a diagram, it must be initialized at the start of an execution. Further, it is necessary that all the $\boldsymbol{now}$ variables are initialized with the same value. This can be obtained by constraining the set of possible initial mappings from lifelines to data states:

$$\mathit{initDataTime}(d) \stackrel{\mathsf{def}}{=} \{\Delta \in \mathit{initData}(d) \mid \exists r \in \mathbb{R}_+ : \forall l \in ll.d : \Delta(l)(\boldsymbol{now}) = r\}$$

where $\mathbb{R}_+$ denotes the positive non-zero real numbers.

We introduce a special event $\mathit{tick}(r)$, with $r \in \mathbb{R}_+$, that represents elapse of time with increment r. Further we add to the execution system a rule that allows time to elapse:

$$[\beta, d, \Delta] \xrightarrow{\mathit{tick}(r)} [\beta, d, \mathit{nowInc}(\Delta, r)] \tag{19.8}$$

with

$$r = \delta(\beta, d, \Delta)$$

being the time increment.

$\mathit{nowInc} \in (\mathcal{L} \to \widehat{\Sigma}) \times \mathbb{R}_+ \to (\mathcal{L} \to \widehat{\Sigma})$ is a function that increments the $\boldsymbol{now}$ variable in every data state with the time increment, so that the time of all lifelines remain synchronized:

$$\mathit{nowInc}(\Delta, r) \stackrel{\mathsf{def}}{=} \{l \mapsto \sigma[\boldsymbol{now} \mapsto (\sigma(\boldsymbol{now}) + r)] \mid l \in \mathit{dom}.\Delta \wedge \sigma = \Delta(l)\}$$

$\delta \in \mathcal{B} \times \mathcal{D} \times (\mathcal{L} \to \widehat{\Sigma}) \to \mathbb{R}_+$ is a function that determines the time increment based on the current state. The idea is then that the time model we choose to use in an execution may be specified by the function δ. This can be either discrete or dense time, with fixed or randomized time increments, dependent or independent of the state of execution, and with lazy or eager time. The simplest imaginable case may be a discrete time model with unit time increments. In this case δ is defined as:

$$\delta([\beta, d, \sigma]) \stackrel{\mathsf{def}}{=} 1$$

We do not go further into this, other than recognizing that there exists a variety of ways to define the time increment function.

There are some properties one might desire of a time model to avoid a counter-intuitive representation of time and anomalies. The first of these is progress, i.e. that time does not stop, but eventually advances. (Because advancement of time is alway

[1] For a discussion on difference approaches to modeling time, see section 21.2.

enabled, progress and fairness become equivalent.) The other is to avoid Zeno's paradox, i.e. that even if time always advances, it never reaches a given point in time. Both these properties can be ensured by placing a requirement on execution similar to what we did to ensure weak fairness of the operational semantics in section 8.4. The requirement basically states that for every execution, at some point it will overtake every possible points in time:

$$\forall[\beta_1, d_1, \Delta_1] \xrightarrow{x_1} [\beta_2, d_2, \Delta_2] \xrightarrow{x_2} [\beta_3, d_3, \Delta_3] \xrightarrow{x_3} \cdots \in \Xi, r \in \mathbb{R}_+ : \\ \exists i \in \mathbb{N} : \forall l \in ll.d_i : \Delta_i(l_i)(\boldsymbol{now}) > r$$

This requirement also becomes a requirement on the time increment function δ; the possibility of defining δ is restricted to definitions that allow the requirement on executions to be fulfilled.

19.2.2 Events with timestamps

In timed STAIRS, events have timestamps. This is defined so that in the syntactic representation of an event (i.e. in a diagram), it contains a timestamp tag, and in the semantics representation of the event (i.e. in a trace) the timestamp tag is mapped to a real number representing the point in time of the occurrence of the event.

It must be noted that timed STAIRS operates with three events: transmission, reception and consumption. The difference between reception and consumption is that reception is the event that a message is placed in the input buffer of the receiver, and consumption the event that the message is removed from the input buffer and the receiver starts processing the message. In the untimed version of the operational semantics we have not dealt explicitly with this difference, but in the context of time constraints this becomes relevant because we may want to place time constraints explicitly on the communication medium or on a lifeline. In the following we assume the receive events of our operational semantics to be the receive events of timed STAIRS, and hence that the consumption events are hidden. However, we assume that this does not affect the ordering of events.

The timestamp tags can be considered variables. Hence, in the operational semantics, events are triples of kinds, messages and variables in the syntactic representation:

$$(k, m, t) \in \mathcal{K} \times \mathcal{M} \times \mathit{Var}$$

In the following we use $\mathcal{E} \stackrel{\mathsf{def}}{=} \mathcal{K} \times \mathcal{M} \times \mathit{Var}$ as the definition of the set of events $\mathcal{E}$. We let all functions on the old definition of $\mathcal{E}$, e.g., $k.e$, $m.e$ and $l.e$ carry over to this new definition. We define a function $t._- \in \mathcal{E} \to \mathit{Var}$ that returns the timestamp tag of an event:

$$t.(k, m, t) \stackrel{\mathsf{def}}{=} t$$

In addition we put the constraint on diagrams that each timestamp tag is only allowed to occure ones.

In the semantic representation of events, i.e. events that are produced in an execution step, they are tuples of kinds, messages, variables and reals:

$$(k, m, t \mapsto r) \in \mathcal{K} \times \mathcal{M} \times \mathit{Var} \times \mathbb{R}_+$$

We write $t \mapsto r$ to emphasize that we view r as a value that t maps to. In timed STAIRS the set of "semantic events" is denoted $[\![\mathcal{E}]\!]$. It should be noted that while

the operational semantics as defined in chapter 8 produces traces $h \in (\mathcal{E} \cup \mathcal{T})^\omega$, the redefined operational semantics with data and time produces traces $h \in (\llbracket \mathcal{E} \rrbracket \cup \mathcal{T} \cup \{write(\Delta, \Delta') \mid \Delta, \Delta' \in \mathcal{L} \to \widehat{\Sigma}\} \cup \{check(b) \mid b \in \mathbb{B}\} \cup \{tick(r) \mid r \in \mathbb{R}_+\})^\omega$.[2]

In addition to having these timestamp tags as part of events, we also consider them variables of the lifelines on which the events occur. Hence, the function *var* is redefined for events to be:

$$\begin{array}{rcl} var((!, (s(\varphi), tr, re), t)) & \stackrel{\mathsf{def}}{=} & \{t\} \cup \bigcup_{v \in dom.\varphi}(\{v\} \cup var(\varphi(v))) \\ var((?, (s(\varphi), tr, re), t)) & \stackrel{\mathsf{def}}{=} & \{t\} \cup dom.\varphi \end{array}$$

In the initial data states, these timestamp variables will have arbitrary values as any other variable. When an event is executed, two things happen: The timestamp tag t of the event is assigned the current value of ***now***, and at the same time an implicit assignment assigns the value of ***now*** to the variable t of the lifeline on which the event occurs. Thus, rule (19.6) for execution of transmit events is redefined to:

$$[\beta, d, \Delta] \xrightarrow{e} [update(\beta, e), d', \Delta'] \textbf{ if } \Pi(ll.d, \beta, d) \xrightarrow{(!,(s(\varphi),tr,re),t)} \Pi(ll.d, \beta, d') \qquad (19.9)$$

where

$$\begin{array}{rcl} \sigma & = & \Delta(tr) \\ \varphi' & = & eval(\varphi\{\sigma\}) \\ e & = & (!, (s(\varphi'), tr, re), t \mapsto \sigma(\textbf{\textit{now}})) \\ \Delta' & = & \Delta[tr \mapsto \sigma[\varphi'][t \mapsto \sigma(\textbf{\textit{now}})]] \end{array}$$

As can be seen, the only difference from rule (19.6) is that the timestamp tag and variable t is set to have the value of ***now***.

The rule (19.7) for receive events is redefined analogously:

$$[\beta, d, \Delta] \xrightarrow{e} [update(\beta, e), d', \Delta] \textbf{ if } \Pi(ll.d, \beta, d) \xrightarrow{(?,(s(\varphi),tr,re),t)} \Pi(ll.d, \beta, d') \qquad (19.10)$$

where

$$\begin{array}{rcl} \sigma & = & \Delta(re) \\ \varphi' & = & getData(\beta, (s(\varphi), tr, re)) \\ e & = & (?, (s(\varphi'), tr, re), t \mapsto \sigma(\textbf{\textit{now}})) \\ \Delta' & = & \Delta[re \mapsto \sigma[\varphi'][t \mapsto \sigma(\textbf{\textit{now}})]] \end{array}$$

Because the time of all lifelines are synchronized, these rules produce traces where the events are ordered accoring to their timestamps.

19.2.3 Example

In order to illustrate the use of our formalism we apply an example from the UML standard. The UML standard allows specification of time constraints in sequence diagrams in a declarative fashion, as shown in the following example [134, p. 509]:

> The Sequence Diagram in Figure [19.5] shows how time and timing notation may be applied to describe time observation and timing constraints. The *User* sends a message *Code* and its duration is measured. The *ACSystem* will send

[2] We use A^ω to denote the set $A^* \cup A^\infty$, i.e. the set of finite and infinite traces of elements from a set A.

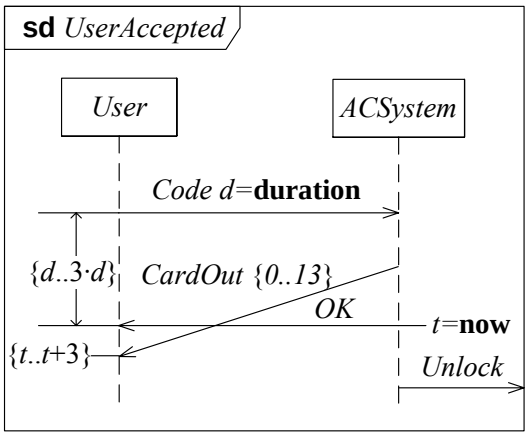


Figure 19.5: Sequence diagram with time constraints

two messages back to the *User*. *CardOut* is constrained to last between 0 and 13 time units. Furthermore the interval between the sending of *Code* and the reception of *OK* is constrained to last between d and $3 \cdot d$ where d is the measured duration of the *Code* signal. We also notice the observation of the time point t at the sending of *OK* and how this is used to constrain the time point of the reception of *CardOut*.

By using just our mechanisms for recording the points in time that messages are transmitted and received, and for manipulating and putting constraints on data, we are able to express the same time constraints with our time and data enhanced sequence diagrams. Assume the events of the diagram have the following timestamp tags:

$$\begin{aligned} t.!Code &= t_1 & \qquad t.?Code &= t_2 \\ t.!CardOut &= t_3 & \qquad t.?CardOut &= t_4 \\ t.!OK &= t_5 & \qquad t.?OK &= t_6 \\ t.!Unlock &= t_7 \end{aligned}$$

Syntactically we place these timestamp tags along the lifelines, beside the events to which they belong. The diagram *UserAccepted*′ of figure 19.6 then expresses the same time constraints as in *UserAccepted*.

The point in time of transmission of message *Code* is stored in the variable t_1 and is also transmitted to *ACSystem* by having $c := \textbf{\textit{now}}$ as a parameter in the message. Upon reception of the message *Code*, the point in time of the reception is assigned to the variable t_2. The duration of the message is calculated by the assignment $d := t_4 - c$, and this duration is transmitted back to *User* by the variable d' in the message *OK*. The timestamp tag of the reception of *OK* is t_6, so the time constraint between the transmission of *Code* and reception of *OK* is expressed by the constraint $\{d' \leq t_6 - t_1 \leq 3 \cdot d'\}$. t, which is the point of time for transmission of *OK* is obtained by having $t := \textbf{\textit{now}}$ as parameter to *OK*. Similarly, the point in time of transmission of *CardOut* is made available to *User* by having $co := \textbf{\textit{now}}$ as a parameter in *CardOut*. The timestamp tag for reception of *CardOut* is t_4. The time constraint on the reception of *CardOut* is then expressed by the constraint $\{t \leq t_4 \leq t{+}3\}$, and the time constraint on the duration of *CardOut* is expressed by the constraint $\{0 \leq t_4 - co \leq 13\}$.

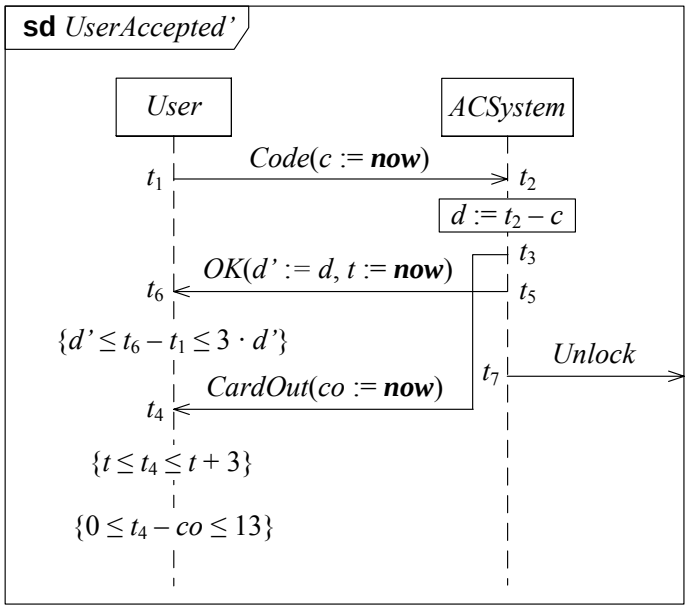


Figure 19.6: Time constraints expressed with variables, timestamps and constraints

Chapter 20

Case study

This chapter reports on a case study on the use of the time enhanced sequence diagrams defined in chapter 19. The study focuses on the expressiveness of our approach to specify timing properties in sequence diagrams. A number of different approaches and notations for expressing real-time properties in sequence diagrams and similar specification languages exists. Our hypothesis is that our approach is sufficiently general to capture real-time properties expressed by other approaches and that it has sufficient expressiveness to capture interesting real-time properties.

The case study consisted of two parts. In the first part a literature survey was conducted. In the survey a number of publications containing sequence diagram or MSC specifications with time constraints or other characterizations of real-time properties were identified. A collection of specifications were chosen from these publications and harmonized in their notation (with the exception of their notation for time constraints and properties). In the second part of the study we made new versions of the specifications of the collection where we applied our approach to specifying time constraints and properties, as defined in chapter 19.

In section 20.1 we present assumptions and success criteria. Section 20.2 presents the results of the study. In section 20.3, the results are discussed.

20.1 Assumptions and success criteria

In this study there are two main assumptions. The first assumption is that the specifications we have been able to identify through the literature survey are representative for the variety of techniques and notation for expressing time constraints and real-time properties one may expect to be applied to sequence diagrams and similar specification languages.

The second assumption is that our collection of specifications also represents specification of what we in the hypothesis called interesting real-time properties. By this we mean that we expect the authors of the publications we have surveyed to present specifications that illustrate how they intend their notation for real-time properties to be used, and that they present examples of real-time properties they consider interesting to specify.

With these assumptions we can concretize the hypothesis of this case study in two success criteria:

1. *Our approach is sufficiently general to capture real-time properties expressed by*

the approaches identified through the literature survey.

2. *Our approach has sufficient expressiveness to capture the real-time properties in the collection of specifications identified in the literature survey.*

20.2 Results

This section presents the results of the case study. Section 20.2.1 presents the results of the literature survey. In order to make the presentation easier to follow, we do not present the collection of specifications identified in the survey separately, but present them together with our new versions of the specifications. Each specification in the collection, together with our version of the specification, is given a separate sub-section following section 20.2.1. After these presentations, a summary of the results is provided in section 20.2.14.

20.2.1 Results of the literature survey

The literature survey had four kinds of sources:

- Publications on STAIRS.
- Scientific papers with titles containing words like "UML", "MSC", "sequence diagrams", "interactions", "time", "timing" and "real-time".
- Text books on using UML or MSC in development of real-time systems.
- Standards defining notation for time constraints or real-time properties for UML sequence diagrams or MSCs.

The result of the survey is a collection of specifications containing time constraints or other characterizations of real-time properties. The specifications in the collection

Description	**Source**	**Figure**
Request-reply 1	[153, p. 14]	20.1
Request-reply 2	[53, p. 42]	20.2
Delayed message	[53, p. 41]	20.5
Absolute and relative delay	[192, p. 87]	20.7
Specification of timer	[7]	20.9
Use of timer	[192, p. 96]	20.11
Examples of timing constraints	[106, p. 662]	20.13
Pacemaker setup scenario	[41, p. 239]	20.15
Protocol for an audio/video component	[51, pp. 655–657]	20.17
Heartbeat packet emission	[71, p. 710]	20.19
ATM example	[145, p. 2] & [144, p. 33]	20.21
AMT test case	[133, p. 54]	20.23
ATM specification	[17, pp. 15–17] & [18, pp. 103–104]	20.25–20.31

Table 20.1: Specifications with real-time properties

are presented in table 20.1. A short description to identify each specification is given, together with the source of the specification and a reference to the figure(s) where we present it.

In addition to this collection, we also identified some specifications with real-time properties that we have decided to keep out of the study. The reason for not including them in the collection is mainly to keep the study to a manageable size. The specifications were selected to facilitate a large variation in notation for real-time properties, large variation in the real-time properties characterized in the specifications and large variation in the types of the sources. In addition we wanted at least one specification with some considerably size (i.e. more than one diagram). The criterion for size is fulfilled by the ATM specification, which is the only specification in the collection consisting of more than one diagram.

The specifications with real-time properties we identified, but did not chose for the collection, are found in [70, 71, 77, 132, 134, 193–195]. In [132] there are two examples of "proposed notation", but the same notation is used by similar examples from [41], which are included in the collection. In [194] we find more or less the same examples as in [192], with the exception of a concept called "instance delay" that only applies to high-level MSCs. In [70, 77, 193, 195] there are examples of real-time properties, but they use techniques and notations that we feel are covered by other specifications in the collection. In [71] there are some larger specifications, but they are based on using only MSC timers, which we also find well covered by our selection. Also in the UML standard [134] there is an example. We consider this example as an attempt to present as may features as possible at the same time, and hence not an "interesting" specification as such. We have, however, used it as an example in section 19.2.3.

In the following we present the specifications of the collection together with our versions. In order to make the presentation more consistent we have done some harmonizing of the notation in the diagrams. Most notably we have "translated" MSCs into UML sequence diagrams and we have removed irrelevant elements from some of the specifications. In the specification from [17] we have made some simplifications. We have, however, kept the notation for time constraints and real-time properties as they are in the original, with the exception that we have removed the units of measurement such as seconds and milliseconds. Despite the changes we have implemented to harmonize the notation, we believe we have managed to stay faithful to the time related parts. We apply the convention that the name of a diagram is the same as the name it has in the source, or a suitable descriptive name if it lacks a name in the source. Our version of a diagram uses the same name, but followed by an apostrophe mark '.

20.2.2 Request-reply

A commonly used example of time constraints in sequence diagrams is what we may call "request-reply" examples. What these examples have in common is that a lifeline receives a message and should relpy to the sender within some time limit. As representatives for these we have chosen an example from [153, p. 14], reproduced in figure 20.1, and an example from [53, p. 42], reproduced in figure 20.2. In the first of these specifications we see a notation where the time constraint is placed on the lifeline, restraining the time span between reception of the message *request* and transmission of message *response* in a declarative fashion. In the other example, the messages *a* and

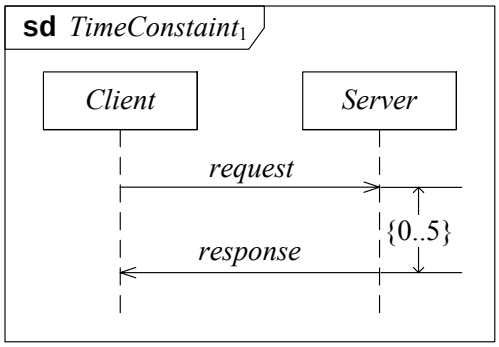


Figure 20.1: Request-reply

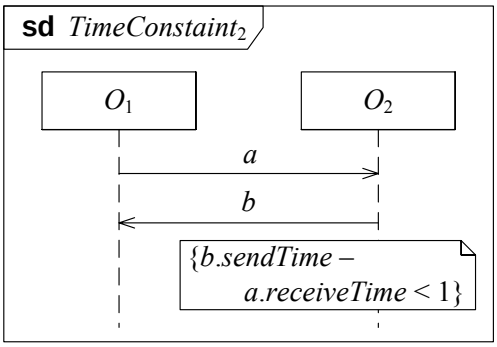


Figure 20.2: Request-reply

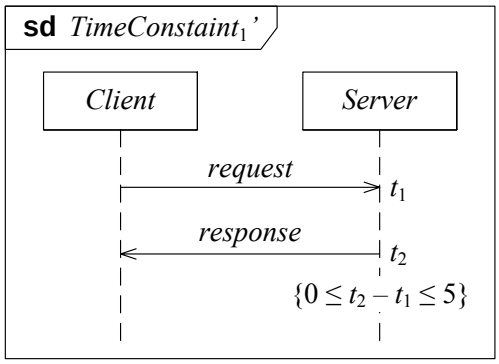


Figure 20.3: Request-reply

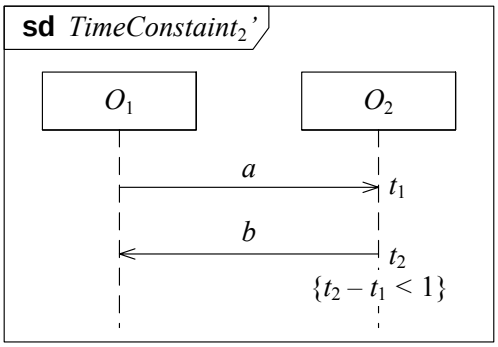


Figure 20.4: Request-reply

b are given attributes *sendTime* and *receiveTime* that are used for expressing a time constraint between reception of a and transmission of b that is placed on the diagram. It is, however, not clear whether there is any semantics related to the fact that the constraint graphically is placed on the lifeline O_2.

In our approach, the time constraints of both these specification are treated in the same way. We introduce timestamp tags on the events that are involved in the time constraints, and use these timestamp tags to express the time constraints as constraint events on the lifelines on which the time constraints apply.[1] Our versions of the specifications of figures 20.1 and 20.2 are shown in figures 20.3 and 20.4, respectively.

20.2.3 Delayed message

In [53, p. 41], the same mechanism as used in the specification of figure 20.2 is applied for specifying a message with time delay. The specification is found in figure 20.5. The attributes *sendTime* and *receiveTime* of a message c is used in a constraint to say that the message has transmission time greater than zero.

Our version of the specification is shown in figure 20.6. The transmit event and send event of the message are given the timestamp tags t_1 and t_2 respectively, but because a timestamp tag must belong to a lifeline, the timestamp tag t_1 is not available in the constraint that we place on lifeline O_2. We handle this by giving the message a parameter t which is assigned the value of ***now*** at the transmission. The variable t is

[1] In these and in later specifications in this chapter, we sometimes hide the timestamp tags that are not used in constraints.

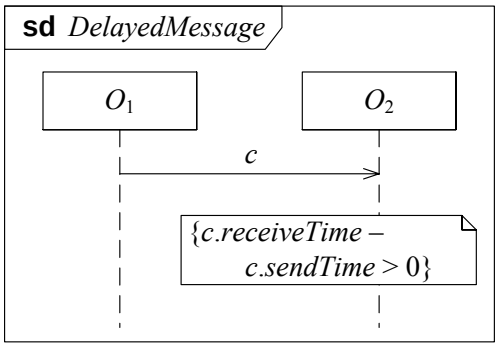


Figure 20.5: Delayed message

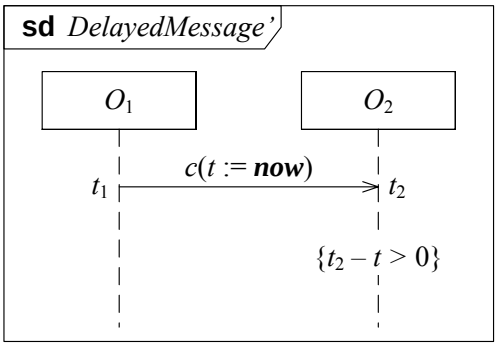


Figure 20.6: Delayed message

available for O_2 and holds the same value as t_1, and can then be used to express the time constraint on the message.

20.2.4 Absolute and relative delay

In a specification from [192, p. 87], reproduced in figure 20.7, time constraints on the lifelines are used for specifying both time delay of messages and time delay between events at the same lifeline. In [192] this is referred to as relative delay. In addition, the specification gives constraints on the absolute points in time of occurrences of events, referred to as absolute delay. Absolute time is denoted by a @. Both relative and absolute delay are features of the MSC-2000 standard [85].

In the specification, each event is given a label. In our version of the specification, shown in figure 20.8, we use these labels as timestamp tags. Because we have these timestamp tags that are assigned the point in time that an event occur, we can easily express both the absolute and relative delays using the same kind of constraints; $\{3 \leq e - a \leq 4\}$ expresses the relative delay between transmission of m_1 and transmission of m_3, while $\{3 \leq e \leq 4\}$ expresses the absolute time delay of the transmission of m_3. The delay between transmission and reception of m_1 is specified using the same

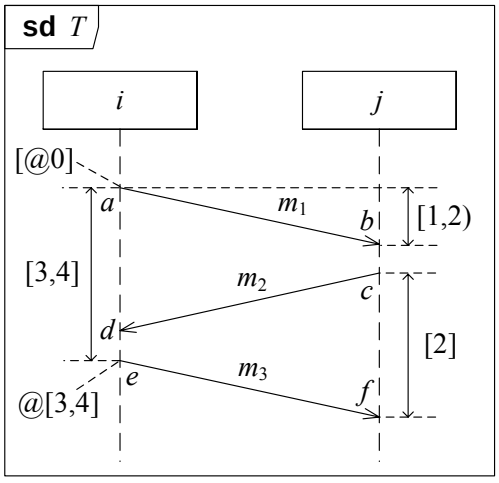


Figure 20.7: Absolute and relative delay

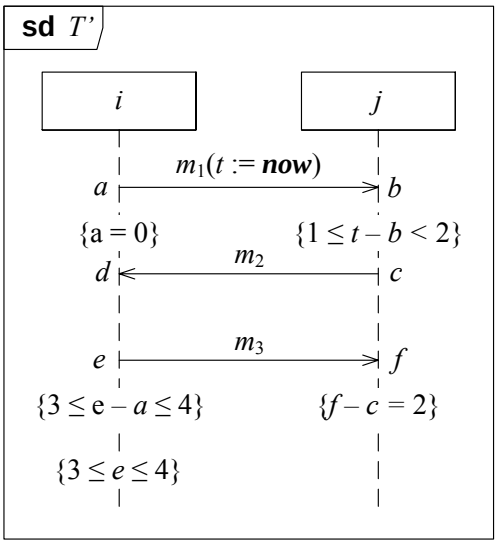


Figure 20.8: Absolute and relative delay

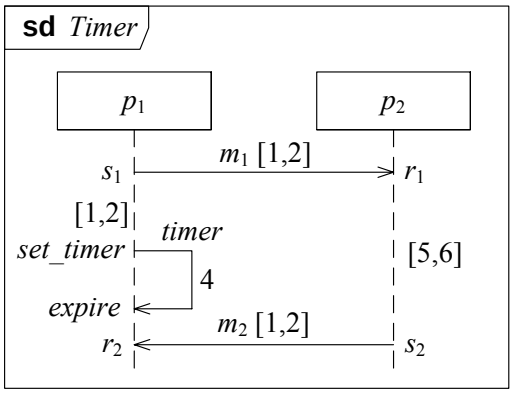


Figure 20.9: Specification of a timer

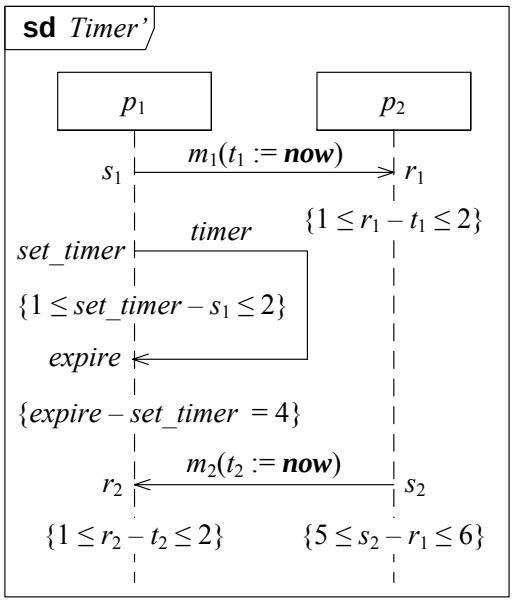


Figure 20.10: Specification of a timer

method as we did for the delayed message in figure 20.5 above.

20.2.5 Specification of timer

In [7], the MSC-2000 mechanism for specifying time constraints is applied to specify a timer, in addition to specifying time constraints on messages and lifelines. The specification is reproduced in figure 20.9. (The original is without labels on the messages; in figure 20.9 these are added by us.)

As in the specification of figure 20.7 the events are labeled. In our version, given in figure 20.10, we use these labels as timestamp tags. We express the time constraints in the same fashion as we did in figure 20.8. However, because the timer is specified as a message with the lifeline p_1 as both sender and receiver, we do not have to pass the point in time that the timer was sent along with the message.

20.2.6 Use of timer

In [192, p. 96], an example of use of the timer construct from MSC-2000 is given. This example is shown in figure 20.11, with the difference that the original is made with a high-level MSC while in our reproduction we use the high-level operator xalt instead. In the example a timer T_1 is set, with five time units, by lifeline i after the transmission of the message m_1. The choice specifies two alternatives: In the first alternative the timer is stopped after lifeline i has received a message n, while in the other, the timer expires.

When we make our version of this specification, shown in figure 20.12, we choose to specify the timer as a lifeline, and not as a message as in figure 20.10. The new lifeline T_1 represents the timer, and again we use the event labels in the original specification as our timestamp tags. According to the explanation of the example in [192], the timer specifies a time delay of 5 between r and s, and a time delay in the interval $(0, 5)$ between r and t. We could easily have specified this by placing constraints $\{s - r = 5\}$

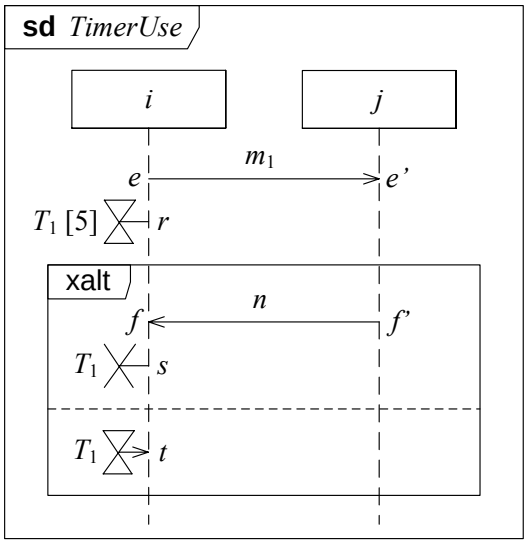


Figure 20.11: Use of a timer

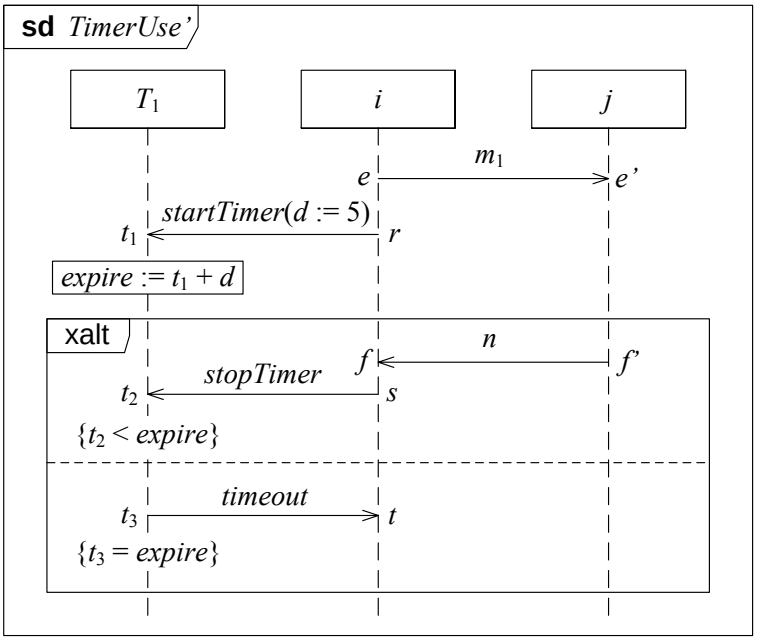


Figure 20.12: Use of a timer

and $\{0 < t - r < 5\}$ on lifeline i. We do, however, chose a more operational approach to specifying timers: The timer is set by sending a message *startTimer* with the delay of 5 time units assigned to a parameter d. A variable *expire* is assigned the point in time the timer should expire as the sum of the timestamp of the reception of *startTimer* and the delay d. In the first alternative, lifeline i sends a message *stopTimer* to T_1, and the reception of *stopTimer* on T_1 is followed by a time constraint stating that the timestamp of the event should be less that the value of *expire*. In the second alternative, the timer sends a message *timeout* to lifeline i, followed by a constraint specifying that the timestamp of transmission of *timeout* should be equal to the value stored in *expire*.

This is obviously not the only way of specifying a timer, but in an operational

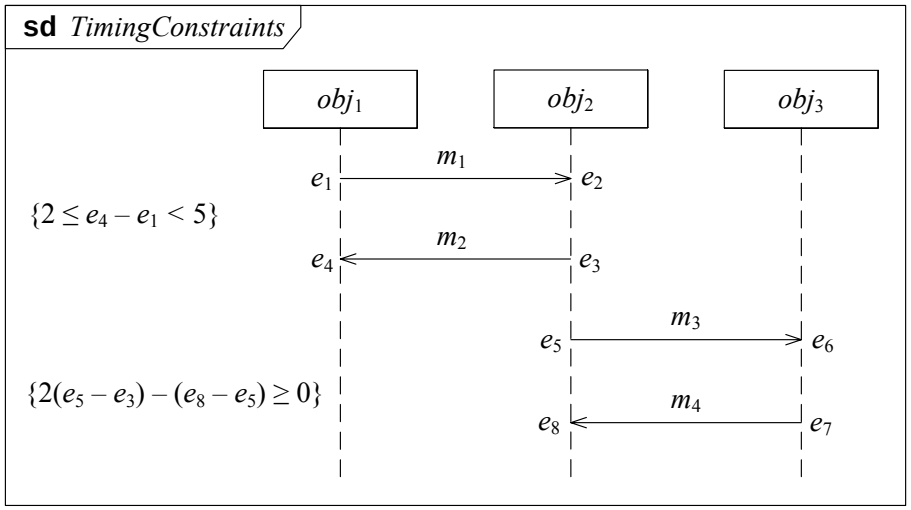


Figure 20.13: Examples of time constraints

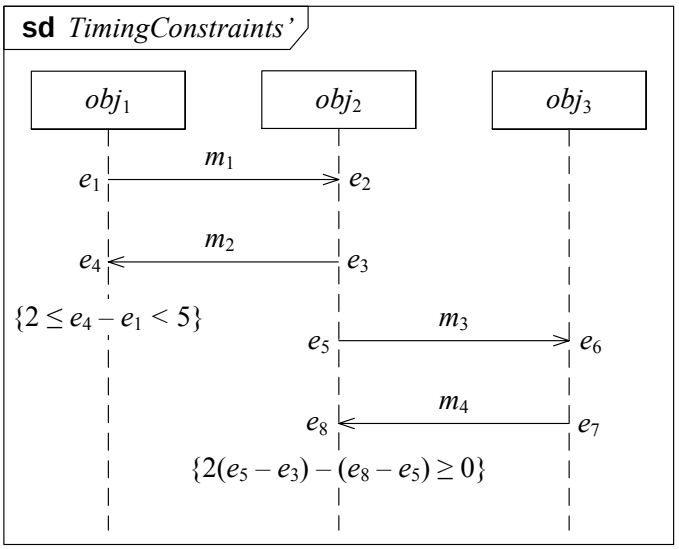


Figure 20.14: Examples of time constraints

setting we find this the most natural way. This is because it recognizes the timer as an independent entity in the specification, and because it places all the responsibilities of a timer on the timer itself.

20.2.7 Examples of timing constraints

In [106, p. 662] an example of a sequence diagram with time constraints is given. We reproduce the diagram in figure 20.13. Each event in the diagram is given a label representing the point in time of its occurrences, and two time constraints using these labels are placed on the diagram.

Our version of the diagram is given in figure 20.14. The event labels are consistent with our timestamp tags, so we simply have to place the time constraints as constraint events on the appropriate lifelines to express the same restrictions on the timing of the events.

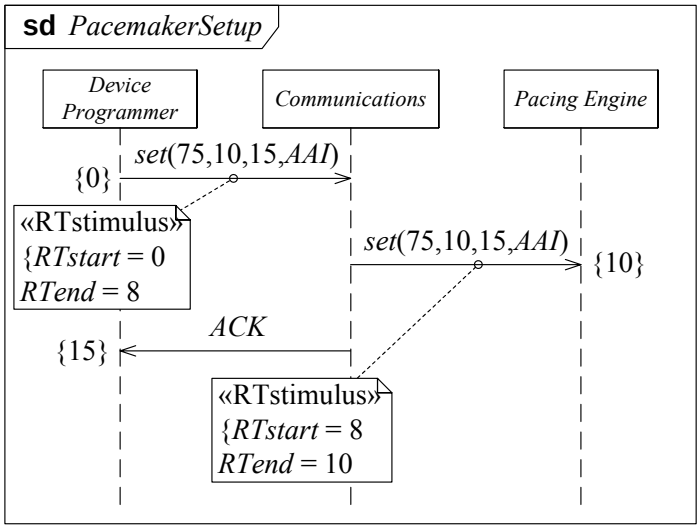


Figure 20.15: Pacemaker setup scenario

20.2.8 Pacemaker setup scenario

In [41, p. 239] a specification of a setup scenario for a pacemaker is provided. The specification uses constructs (stereotypes) from the *UML Profile for Schedulability, Performance, and Time* (often abbreviated UML-RT, for "UML Real-Time") [132]. We reproduce the specification in figure 20.15. A simplification is made in that use of executions specifications, a UML construct we do not deal with in our formalism, and time constraints on these are removed. Time constraints on messages are given by constraints labeled «RTstimulus» attached to the messages. In these constraints the points in time of the transmission and reception of a message are given by the variables *RTstart* and *RTend*, respectively. In addition, constraints on the occurrence of events are given beside the lifelines.

Because all the time constraints in the specification are simple references to points in time, we can simply introduce timestamp tags and constraints on timestamps after

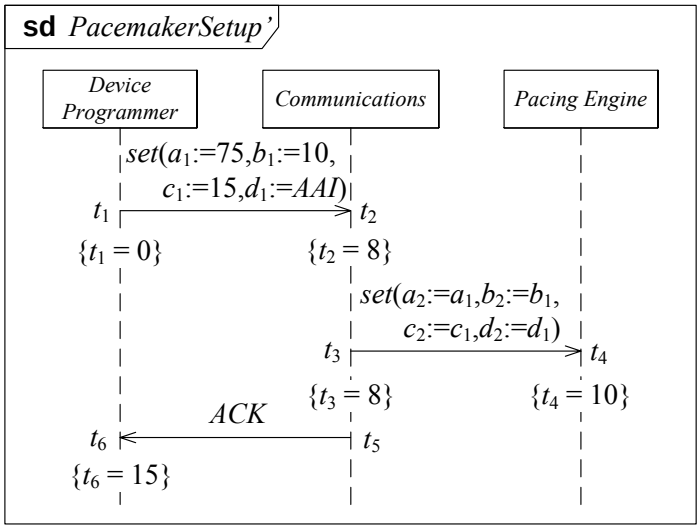


Figure 20.16: Pacemaker setup scenario

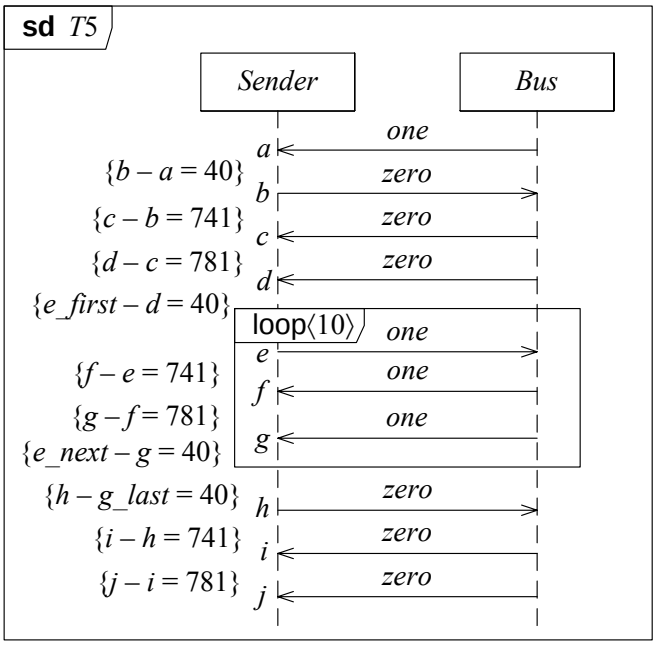


Figure 20.17: Part of a protocol for an audio/video component

the occurrences of transmit or receive events. In order to stay consistent with our handling of data, we have also introduced variables in the messages to hold the values that are passed with the messages. Our version of the diagram is shown in figure 20.16.

20.2.9 Protocol for an audio/video component

In [51, pp. 655–657] a specification of a simple protocol for communication within an audio/video component is presented. The specification consists of five diagram, but because they are all very similar we choose to use only one of them in this study. This diagram is reproduced in figure 20.17. The way of specifying time constraints in this specification is similar to what we saw in figure 20.13; events are given labels and constraints over these labels are placed on the diagram along one of the lifelines. A difference, however, is that this diagram contains a loop, and in the constraints, *x_first*, *x_next* and *x_last* are used to denote the first, next and last occurrence of a label x within the loop.

In our version of the specification, shown in figure 20.18, we use the label as timestamp tags and place the time constraints as constraint events on the lifeline *Sender*. Because of our operational view of sequence diagrams, and because timestamp tags are variables that are assigned values when their events are executed, the elimination of the *x_first*, *x_next* and *x_last* constructs almost resolves itself. Only some clever use of a variable to handle the time constraint containing *e_next* is needed.

20.2.10 Heartbeat packet emission

In [71, p. 710] an MSC specification of Reliable Multicast Transport Protocol (RMTP2) is provided. We have chosen a diagram from this specification. In this diagram, reproduced in figure 20.19, a new construct for specifying a special kind of time constraint is introduced. The message *heartbeat* inside the loop will be transmitted repeatedly

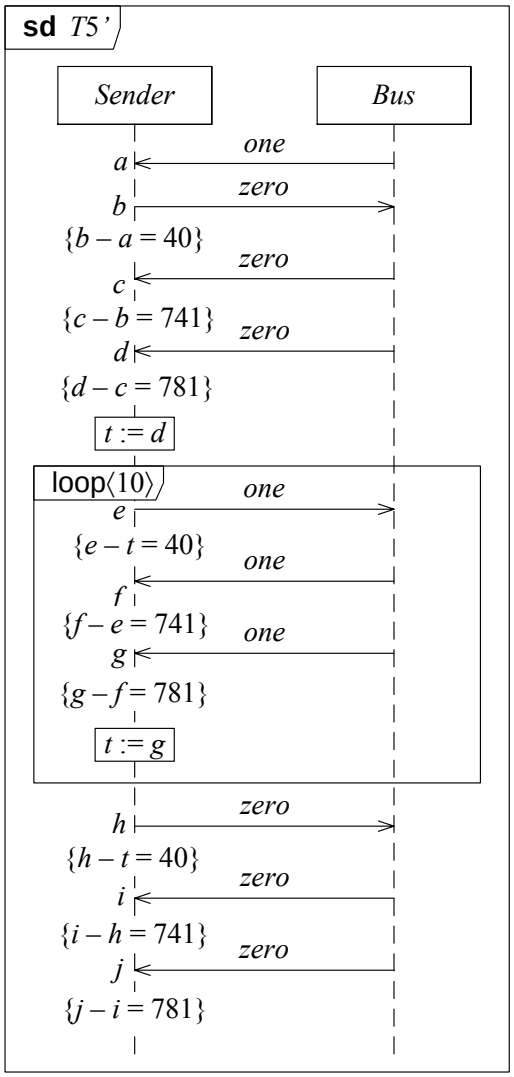


Figure 20.18: Part of a protocol for an audio/video component

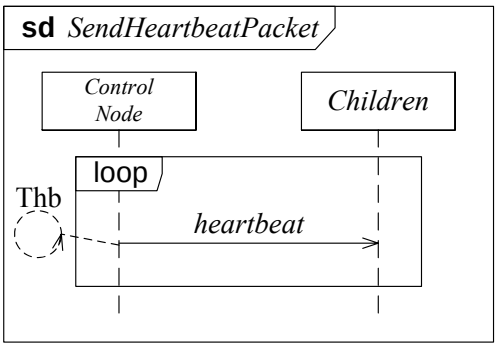


Figure 20.19: Heartbeat packet emission

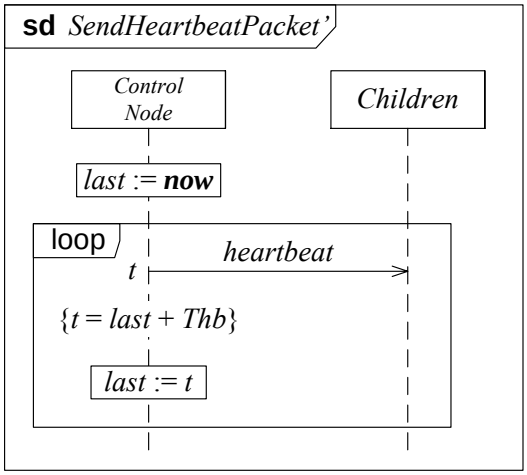


Figure 20.20: Heartbeat packet emission

by the lifeline *Control Node*. The circle attached to the transmit event of the message means that there should be *Thb* time units between each transmission of the message.

Because of our operational approach to sequence diagrams we can specify the time delay between the repeated transmission of the message by means of a variable that is assigned the timestamp of the transmit event after each transmission, and a suitable constraint event. Our version of this diagram is presented in figure 20.20.

20.2.11 ATM example

An example ATM specification is found in [145, p. 2] (and also in [144, p. 33]). In this specification, shown in figure 20.21, some of the events are labeled by timestamp tags and time constraints are expressed by means of notes that are attached to the last of the events which timestamp is used in the constraint.

In our formalism we can easily express the same time constraints by placing them as constraint events on the appropriate lifeline, using the same timestamp tags. Our version of the specification is found in figure 20.22.

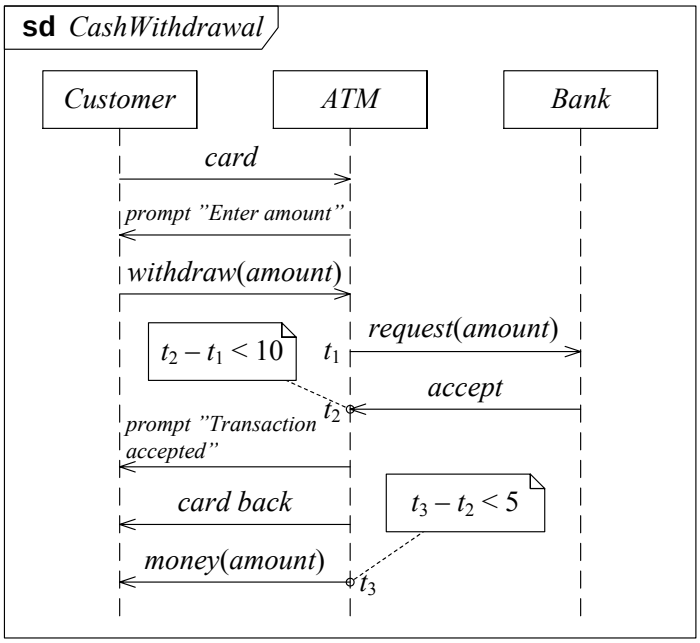


Figure 20.21: ATM example

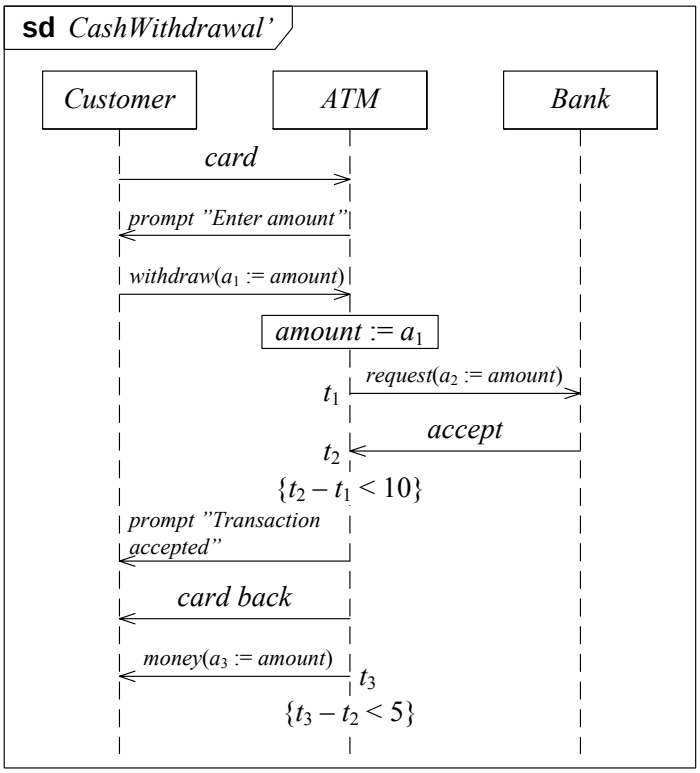


Figure 20.22: ATM example

20.2.12 AMT test case

In the *UML Testing Profile* (UML-TP) [133, p. 54] an example specification of a test case with time constraints is given. We reproduce the specification in figure 20.23, with some simplification; testing specific information and execution specifications have been removed, and return messages have been replaced by ordinary messages.

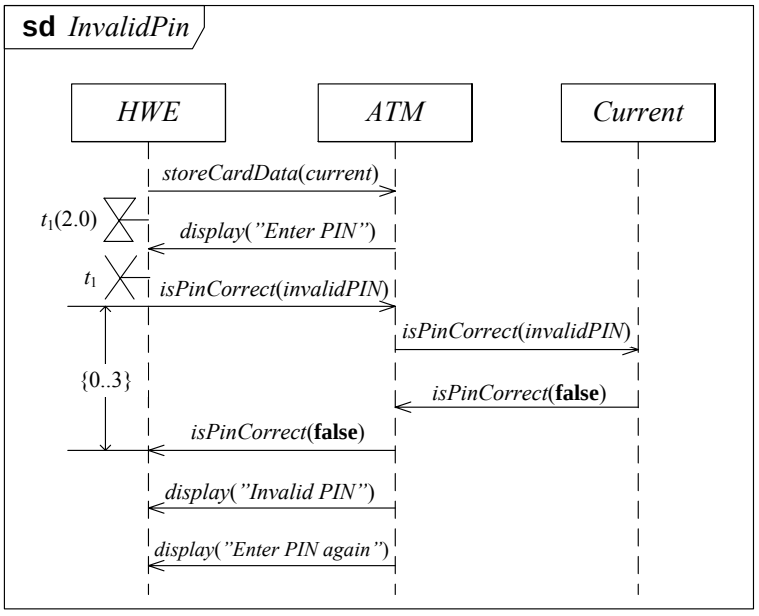


Figure 20.23: ATM test case

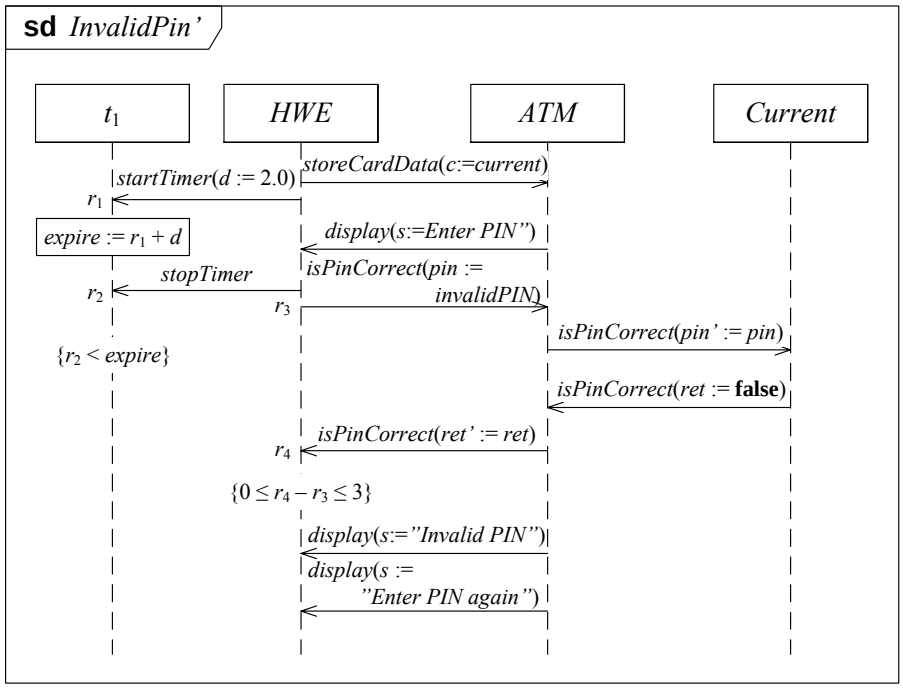


Figure 20.24: ATM test case

The specification uses both a timer and a time constraint between events at the lifeline *HWE*. When we make our version of the specification, shown in figure 20.24, we use the same method for specifying the timer as we did for the timer in figure 20.11 and use the same method for specifying the time constraint as we did for the time constraint in figure 20.1.

20.2.13 ATM specification

From [17, pp. 15–17] (and also [18, pp. 103–104]) we get a larger specification of an example ATM that uses both timers and time delays. We reproduce this specification in figures 20.25–20.31, however with several changes. The specification is originally made by means of high-level MSCs while we use high-level operators and references. In the original specification there are at some places specified time delays between the top of a diagram and the first event, or between the last event and the bottom of the diagram. In the transformation from MSCs to UML sequence diagrams, the structure has been somewhat altered so that time delays are always between events. We have also made sure that the starting, stopping and expiring of a timer always are present in the same diagram. In addition we have removed all loops in the specification, some inconsistencies have been corrected, and we have added a new message (the *welcome_msg* message) in order to make the specification more consistent. Despite these changes, all the main functionality and all time constraints of the specification are intact.

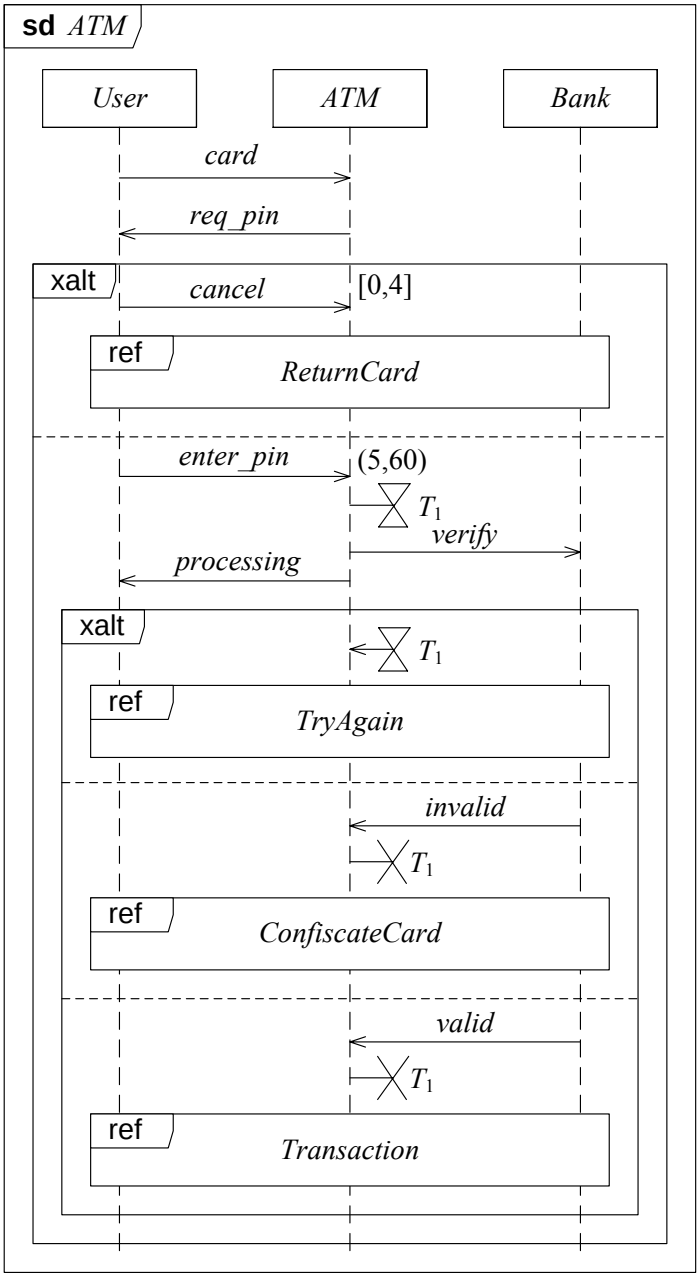


Figure 20.25: ATM specification

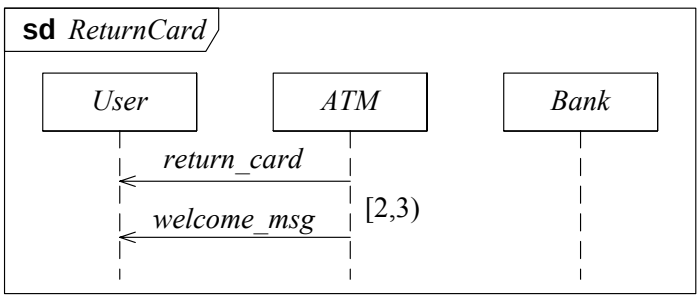


Figure 20.26: *ReturnCard*

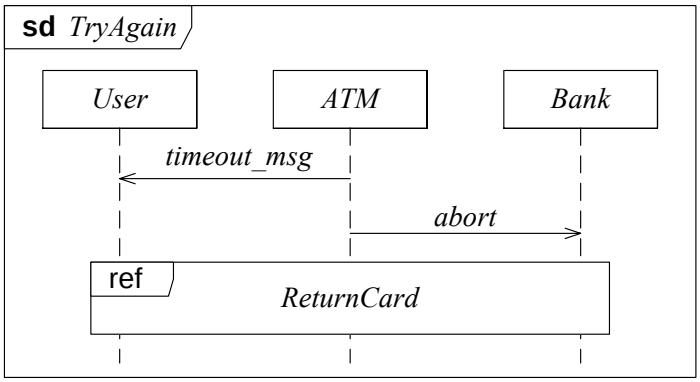


Figure 20.27: *TryAgain*

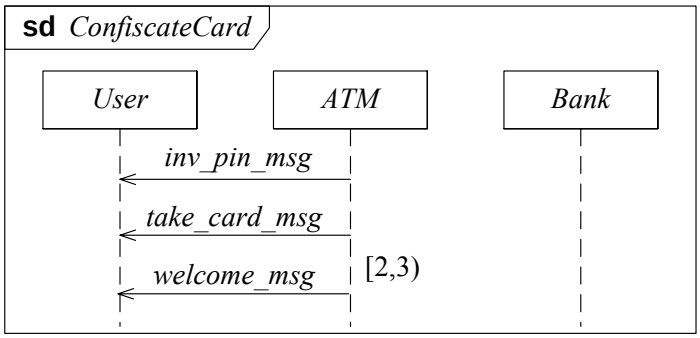


Figure 20.28: *ConfiscateCard*

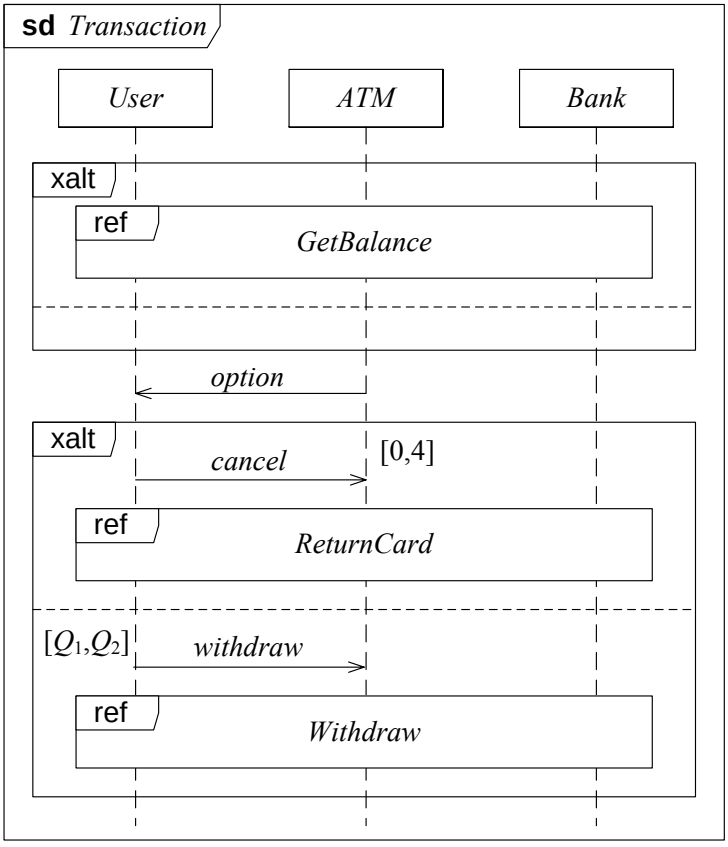


Figure 20.29: *Transaction*

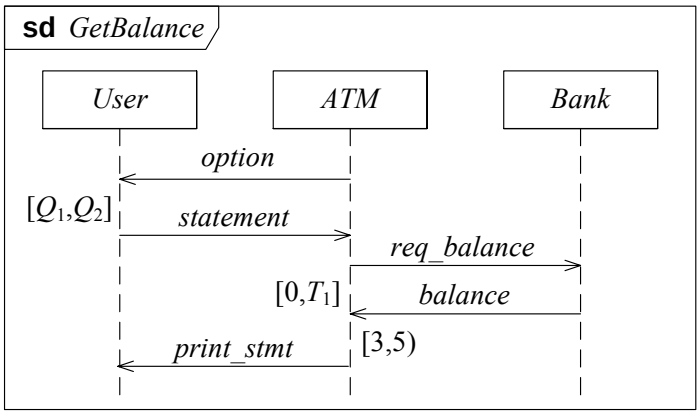


Figure 20.30: *GetBalance*

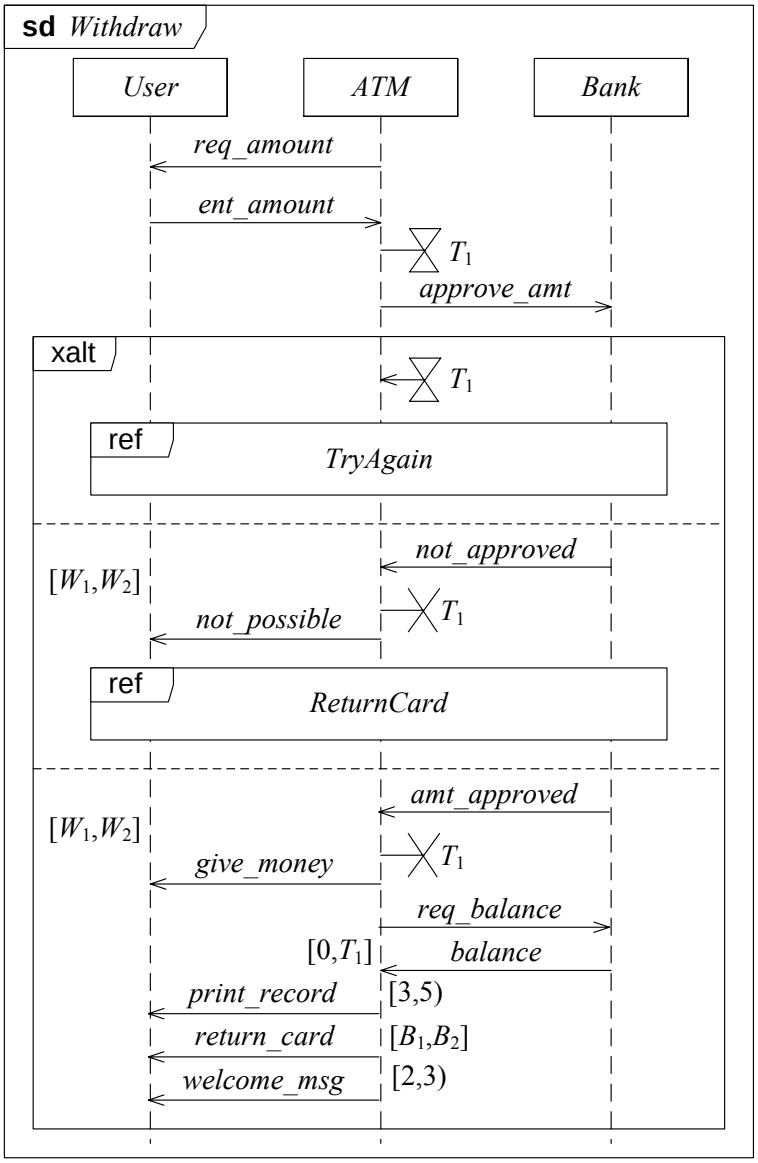


Figure 20.31: *Withdraw*

Our version of this specification is presented in figures 20.32–20.38. The time constraints are handled by introduction of timestamp tags and constraint events over the timestamps. The timers are handled in the same manner as the timers of figures 20.11 and 20.23. In the original, the timers are labeled by T_1, and it is somewhat unclear whether this is the name of the timer or the time delay of the timer. Since T_1 also occurs in time constraints in the specification, we interpret it as a constant value and assume the timer to be anonymous. In our version of the specification we simply call the timer lifeline for *Timer*.

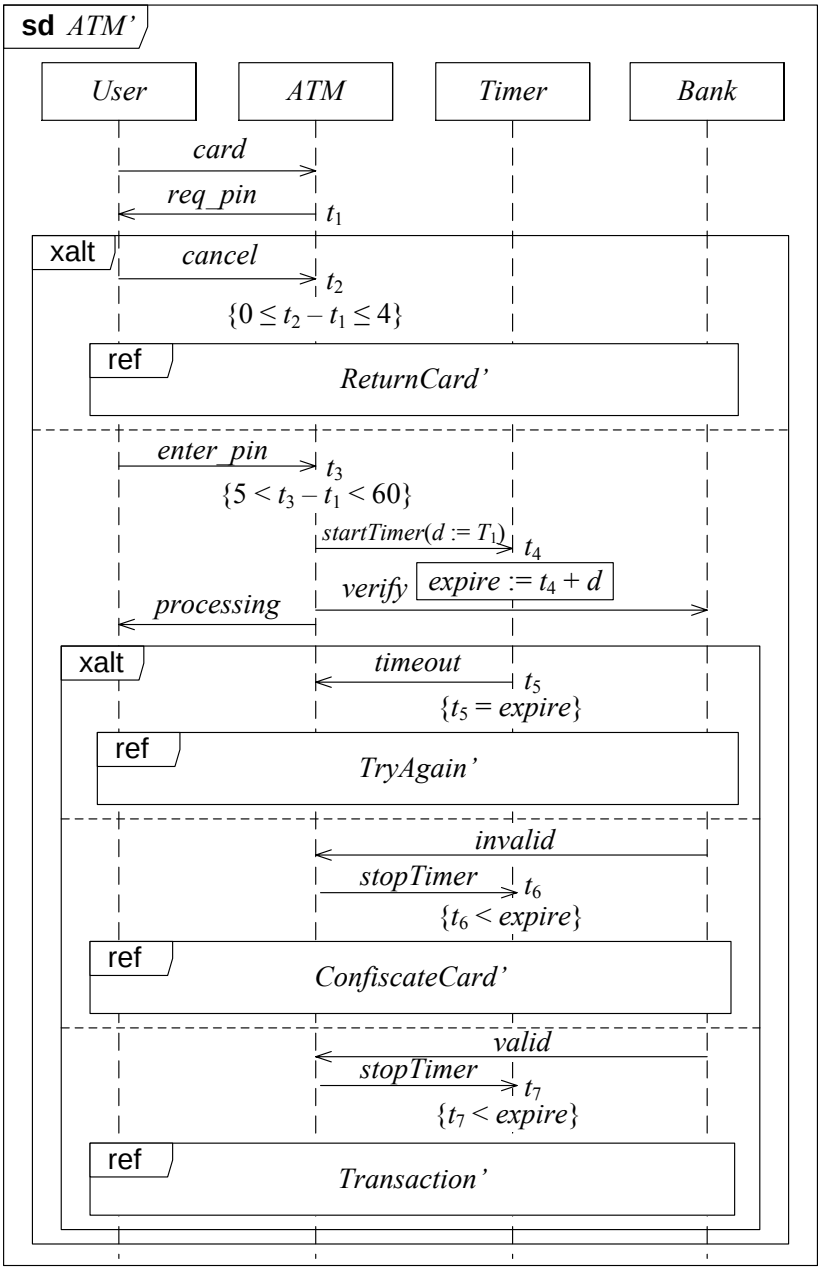


Figure 20.32: ATM specification

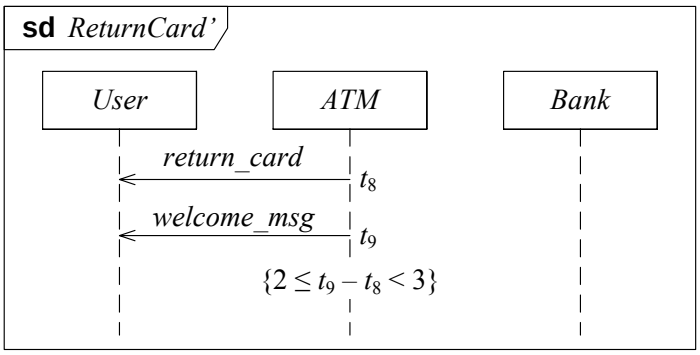


Figure 20.33: *ReturnCard'*

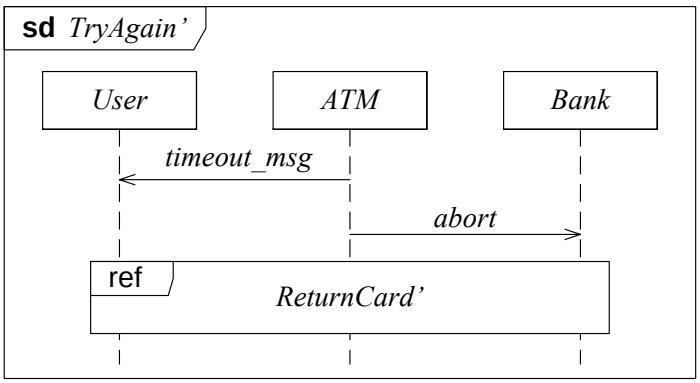


Figure 20.34: *TryAgain'*

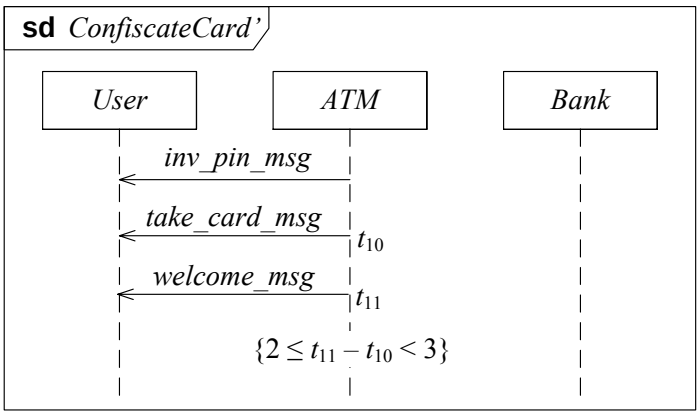


Figure 20.35: *ConfiscateCard'*

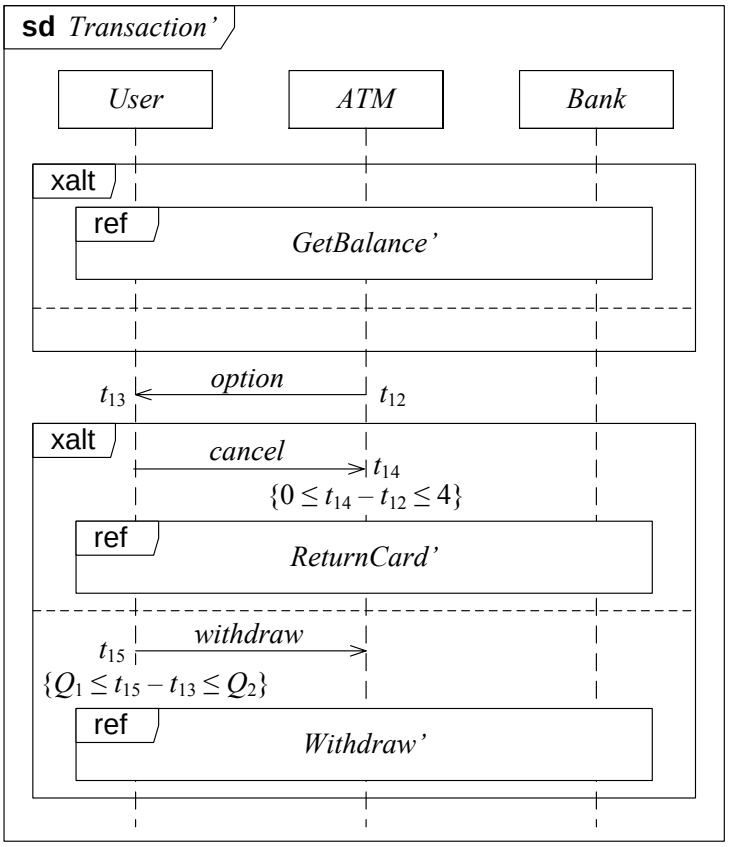


Figure 20.36: *Transaction'*

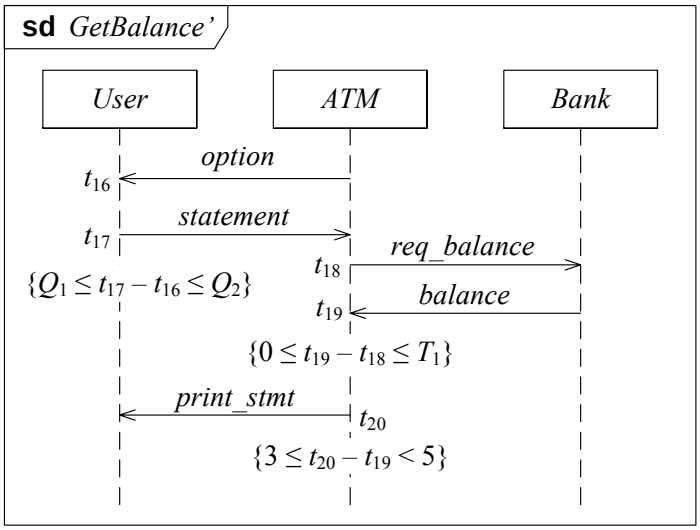


Figure 20.37: *GetBalance'*

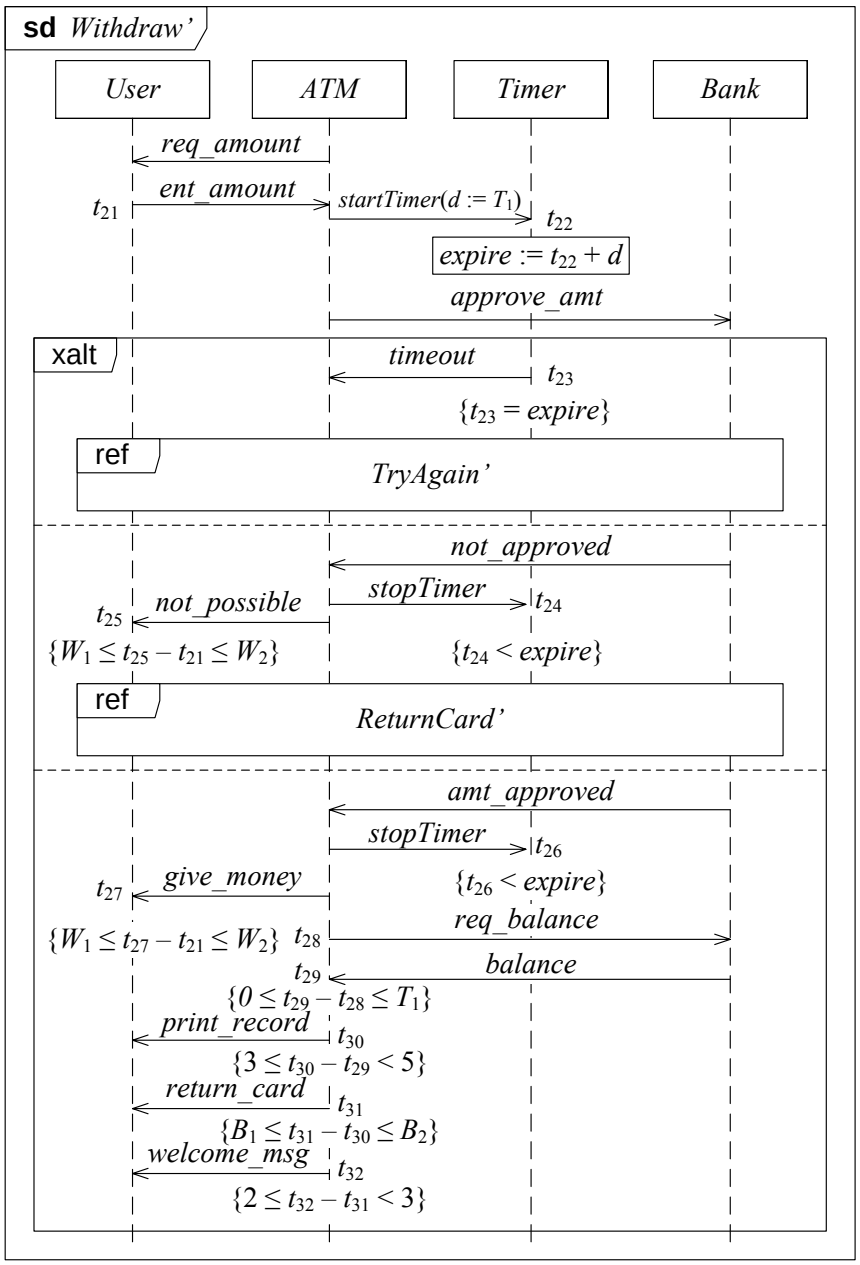


Figure 20.38: *Withdraw'*

Description	Original	Our version	Result
Request-reply 1	20.1	20.3	√
Request-reply 2	20.2	20.4	√
Delayed message	20.5	20.6	√
Absolute and relative delay	20.7	20.8	√
Specification of timer	20.9	20.10	√
Use of timer	20.11	20.12	√
Examples of timing constraints	20.13	20.14	√
Pacemaker setup scenario	20.15	20.16	√
Protocol for an audio/video component	20.17	20.18	√
Heartbeat packet emission	20.19	20.20	√
ATM example	20.21	20.22	√
AMT test case	20.23	20.24	√
ATM specification	20.25–20.31	20.32–20.38	√

Table 20.2: Results of the case study

20.2.14 Summary of results

In table 20.2 we summarize the results of the case study. Each specification in the collection has one row in the table. In each row we give the short description of the specification, a reference to the figure(s) where we have reproduced the original specification, and a reference to the figure(s) with our version. In the last column of the table we have checked of the specifications where we were able to express the real-time properties of the original with our time enhanced sequence diagrams.

20.3 Discussion

In section 20.1 we formulated two success criteria for the case study. The first criterion was that we, with our time enhanced sequence diagrams, have sufficient generality to express the real-time properties formalized by the variety of techniques and notations identified trough our literature survey. This criterion must be said to be fulfilled. As is shown by table 20.2 we were able to express all real-time properties in the collection, even though there is a large variation in the approaches to specifying the properties. Further, we can say that in all but one of the cases (the "protocol for an audio/video component" specification) we found a general way "translating" the specification, which may be applied to similar cases. From this we cannot draw the conclusion that we have the generality to express every real-time property that can be express by all of the other approaches. It is most likely possible to construct examples we cannot handle, specifically because we only have local variables and local constraints. We do, however, have the generality to express all of the real-time properties the authors of the publications in the survey considered interesting to present, and most of them in a schematic fashion.

The second success criterion stated that our time enhanced sequence diagrams should have the expressiveness to capture the real-time properties of the specifications in the collection. As shown by table 20.2, we were able express all the real-time properties present in the collection of specification, and can say that the criterion is

fulfilled. However, as explained below, this conclusion must be restricted to hard real-time properties expressed as time constraints strictly over transmit and receive events of messages.

It is possible to criticize the collection of specifications made on basis of the literature survey. As explained in section 20.2.1, we do consider the collection representative of the specifications found in the literature. But this only applies to time constraints strictly over events of messages, as there are things we deliberately have overlooked because they are out of scope. Examples of this are the time constraints on execution specification we removed from the specification we got from [41], and specifications in [144, 145] where time constraints are given certain probabilities with which they apply.

We clearly have no guarantee that there are not any publications that we have overlooked or that our selection based on the surveyed publications is not biased. Further, we do not have any guarantee that such a literature survey reveals all interesting cases. Despite these shortcomings, we have managed to sample and handle a large class of the existing approaches for specifying real-time properties in sequence diagrams, and a large class of what must be considered interesting real-time specifications. Hence, we believe we can conclude that the case study has given support to the hypothesis that our approach to specifying real-time properties has the generality to capture real-time properties expressed by other approaches and the expressiveness to capture interesting real-time properties.

Chapter 21

Related work on timed specification and analysis

This chapter presents work related to our operational semantics with data and time. We start by discussing the relation to STAIRS in section 21.1. Then, in section 21.2 we have a look at other formalisms for timed specification and analysis. Finally, in section 21.3 we discuss other approaches to data and time in sequence diagrams.

21.1 Relation to STAIRS

In the following we discuss the relation between STAIRS and the operational semantics with data and time. In section 21.1.1 the extended operational semantics is related to STAIRS' treatment of data, and in section 21.1.2 we do the same with respect to timed STAIRS.

21.1.1 Relation to STAIRS with data

STAIRS with data is defined in [158]. The operational semantics with data is inspired by STAIRS with data, but we have at some points deviated from it. In the following we have a look at the similarities and differences, and provide rationale for our choices where the two semantics differ.

An obvious difference is that the variables are local to the lifeline in the operational semantics, while in the denotational semantics, variables are global. By naming conventions and constraints on the syntax one can easily simulate local variables by global variables, so in this sense the denotational semantics is more general than the operational semantics. This is however not the main difference, and we do not consider the local/global distinction any further in the following discussion.

The main difference between the operational semantics with data and STAIRS with data is the intention with which they have been defined. In [158] the intention is to define an extension to the denotational semantics of STAIRS in which earlier established results with respect of refinement of STAIRS specifications still hold. In contrast, our intention when defining the operational semantics with data, was to make a semantics where we actually do manipulation of data while executing a diagram. This difference in intention results in some differences between the denotational and our operational semantics.

The denotational semantics does not have states in the sense of the execution states of the operational semantics. An assignment is therefore characterized by an event $write(\sigma, \sigma')$ that holds the data state σ before the assignment and that data state σ' after the assignment. The denotational definition of assignment is:

$$[\![\, \mathsf{assign}(v, x, l) \,]\!] \stackrel{\mathsf{def}}{=} \{(\{\langle write(\sigma, \sigma') \rangle \mid \sigma'(v) = eval(x\{\sigma\}) \wedge \forall v' \in Var : (v = v' \vee \sigma'(v') = \sigma(v'))\}, \emptyset)\}$$

This provides one trace for each possible pair of data states that satisfy the assignment. In difference, the operational semantics will only provide the data states before and after assignment as they exist in the execution states. This means that in a trace of the denotational semantics the values in the data state may implicitly change at any time between assignments, while this will never happen in the operational semantics. Hence, the operational semantics is sound, but not complete, with respect to the denotational semantics.

The difference with respect to constraints is similar. In the denotational semantics, a constraint is characterized by an event $check(\sigma)$ where σ represent the data state at the time of the evaluation of the constraint. The denotational definition is:

$$[\![\, \mathsf{constr}(x, l) \,]\!] \stackrel{\mathsf{def}}{=} \{(\{\langle check(\sigma) \rangle \mid eval(x\{\sigma\}) = \mathbf{true}\}, \{\langle check(\sigma) \rangle \mid eval(x\{\sigma\}) = \mathbf{false}\})\}$$

A positive trace with an event $check(\sigma)$ is produced for every σ that satisfies the constraint (i.e. evaluates the constraint to true) and a negative trace is produced for every σ that does not satisfy the constraint. In the operational semantics, however, we evaluate the constraint based on the data state at that point in the execution and produce $check(\mathbf{true})$ or $check(\mathbf{false})$ depending on whether it was evaluated to true or false.

In summary we can say that the difference is that the operational semantics does exactly the data manipulation that is specified in the diagram (and nothing more), while the denotational semantics characterizes the traces of all possible implicit data manipulations that satisfy the explicit specification of data manipulation.

A more visible difference in addition to this, is that the operational semantics allows messages to carry data, while STAIRS with data has no syntax for this. This is a result of the difference described above, because in STAIRS with data, messages may implicitly carry data, while all data handling must be explicit in the operational semantics.

21.1.2 Relation to timed STAIRS

Timed STAIRS is defined in [70].[1] In timed STAIRS there is a distinction between syntactic and semantic events. In the syntax an event is a tripple $(k, m, t) \in \mathcal{E}$ of a kind k, message m and timestamp tag t. The semantic events are defined as:

$$[\![\, \mathcal{E} \,]\!] \stackrel{\mathsf{def}}{=} \{(k, m, t \mapsto r) \mid (k, m, t) \in \mathcal{E} \wedge r \in \mathbb{R}\}$$

[1] As explained in section 19.2.2, timed STAIRS is a three event semantics that for each message m defines three events $(!, m)$, $(\sim, m)$ and $(?, m)$, where $(\sim, m)$ denotes that m is placed in the receiver's input buffer. We do not operate with explicit input buffers in the lifelines in our operational semantics and regard our receive events $(?, m)$ as the same as the receive events $(\sim, m)$ of timed STAIRS.

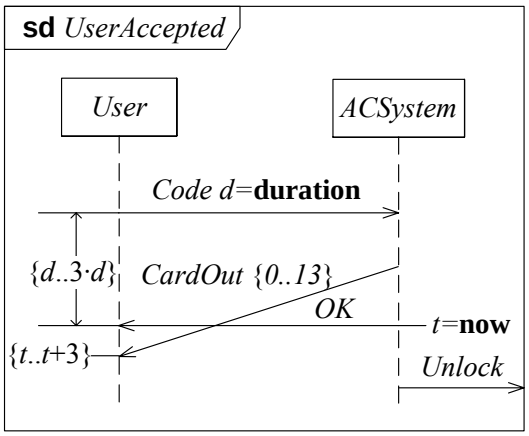


Figure 21.1: Sequence diagram with time constraints

In other words events where the timestamp tag maps to a timestamp represented by a real number. With the exception that we apply variables as timestamp tags, our definitions of events are identical to those given by timed STAIRS. In timed STAIRS a requirement is placed on traces to ensure that time increases monotonic in every trace. In the operational semantics, this is obtained by letting the clock variable ***now*** of all lifelines be synchronized by the tick rule, and by asserting that time increments always are positive numbers.

A time constraint in timed STAIRS is a Boolean expression over the timestamp tags of the events of a diagram. The denotation of a time constraint is defined as

$$[\![d \ \mathsf{tc} \ x]\!] \stackrel{\mathsf{def}}{=} \{(p \wr x, n \cup (p \wr \neg x)) \mid (p, n) \in [\![d]\!]\}$$

where

$$s \wr x \stackrel{\mathsf{def}}{=} \{h \in s \mid h \models x\}$$

and $h \models x$ expresses that x is true when the timestamp tags of x is replaced by the timestamps they map to in the events of trace h.

Let us again consider the diagram *UserAccepted* from section 19.2.3 as an example. The diagram is shown in figure 21.1. In timed STAIRS, the diagram may be expressed as

$$\begin{aligned} \mathit{UserAccepted} \ = \ & (!\mathit{Code} \ \mathsf{seq} \ ?\mathit{Code} \ \mathsf{seq} \ !\mathit{CardOut} \ \mathsf{seq} \ !\mathit{OK} \\ & \mathsf{seq} \ ?\mathit{OK} \ \mathsf{seq} \ ?\mathit{CardOut} \ \mathsf{seq} \ !\mathit{Unlock}) \ \mathsf{tc} \ T \end{aligned}$$

with timestamp tags

$$\begin{aligned} t.!\mathit{Code} &= t_1 & \qquad t.?\mathit{Code} &= t_2 \\ t.!\mathit{CardOut} &= t_3 & \qquad t.?\mathit{CardOut} &= t_4 \\ t.!\mathit{OK} &= t_5 & \qquad t.?\mathit{OK} &= t_6 \\ t.!\mathit{Unlock} &= t_7 \end{aligned}$$

and the time constraint

$$\begin{aligned} T \;=\; & (d = t_2 - t_1 \wedge \\ & d \leq t_6 - t_1 \wedge \\ & t_6 - t_1 \leq 3 \cdot d \wedge \\ & t = t_5 \wedge \\ & t \leq t_4 \wedge \\ & t_4 \leq t + 3 \wedge \\ & 0 \leq t_4 - t_3 \wedge \\ & t_4 - t_3 \leq 13) \end{aligned}$$

The resulting denotation of the diagram is

$$[\![\, \mathit{UserAccepted} \,]\!] = \{(p, n)\}$$

where p contains all the traces of the diagram that has assignments of real numbers to $t_1, t_2, \ldots, t_7$ that satisfy T, and n contains the traces with assignments of real numbers to $t_1, t_2, \ldots, t_7$ that do not satisfy T.

In the operational semantics we have the timestamp tags available as variables. We can therefore simulate the time constraints of timed STAIRS with the use of constraint events with constraints over these variables, as already illustrated in section 19.2.3. The *check* events produced by executing these constraints will determine whether the traces are positive or negative. The monotonicity of time is ensured by the tick rule, and every possible assignment of timestamps to timestamp tags can be obtained by providing a suitable general time model.

The only real difference between the time enhanced operational semantics and timed STAIRS is then that the time constraints of timed STAIRS are global while the constraint events of the operational semantics are local to the lifelines. This means that the constraint events of the operational semantics do not have the full generality of the time constraints of timed STAIRS. However, as illustrated by the example diagram *UserAccepted'* of figure 19.6 in section 19.2.3, some of the limitation in having local, rather than global, constraints may be overcome by clever use of data transmission by messages.

21.2 Formalisms for timed specification and analysis

Several approaches to making timed specifications exist. Best known is maybe the approach of timed automata found in [4]. Timed automata accepts timed words, i.e. sequences of labels where each label has been assigned a timestamp. A timed automaton has a finite set of clocks. These clocks are incremented implicitly and simultaneously according to global time by giving each state in a run of the automaton a clock interpretation that assigns time values to the clocks. This means time is incremented at the transitions of the automaton. Transitions may reset one or more of the clocks to zero, and transitions may have constraints over the clocks so that a transition may only fire if its clock constraints are evaluated to true.

In [21], a formalism for timed labeled transitions systems (LTSs) is applied. In this formalism, each state holds a time value, and special transitions labeled with time delays specify the advancement of time. We have, in chapter 13, made an analogy

between our operational semantics and LTSs. By interpreting our ***now*** variables as the time value of the states and our tick rule as the time delayed transitions, we can make a similar analogy between our operational semantics with time and timed LTSs.

In [140, 141], a timed rewrite theory is defined. The rewrite theory is implemented as Real-Time Maude [138]. In this theory a special type *Time* for time variables is provided and several time domains – such as discrete or dense time – defined. Special timed rewrite rules can be defined. These rules provide time increments that are in accordance with the time domain chosen, and that may or may not be restricted by constraints. In addition, methods for timed analysis, such as timed search are provided.

In [1, 101] a simple approach to real-time specifications is suggested where an ordinary variable $now \in \mathbb{R}$ is added to the specification, and advancement of time is modeled by a special transition rule incrementing this variable. Time related concepts such as timers and time constraints are then expressed by use of expressions over the variable *now* and possibly other variables. A similar approach is used in SDL [86] where a special expression **now** gives access to the system clock, i.e. returning the absolute system time.

This is the approach we have followed, finding this to be the most suitable for our purposes. This allows us to be more explicit in advancement of time than, e.g., the timed automata approach of [4]. Further, the timed automata are very focused on the transitions of the system, which in our approach are more implicit. In our approach we are in any case able to simulate clocks and clock constraints. Much of the same applies to timed LTSs. The timed rewrite logic of [140, 141], while providing a solid framework for making real-time specification in rewrite logic, and has tool support [138], becomes too technical for our purposes. We are mostly concerned with enabling specification of real-time properties in sequence diagrams, and we only apply rewrite logic indirectly through the implementation of our operational semantics. Still, this approach is relevant to our work because it offers the configurability of time models that we are aiming at with our time extended operational semantics.

Several methods for timed analysis based on the formalisms discussed above exist. These include analysis of clock regions (see [4]), timed search (see [140, 141]), timed model checking (see [101, 141]), and timed testing (see [21]). As with the time models we do not go into details on these analysis methods, but recognize that timed analysis has been defined for approaches to timed specification similar to ours. We see a clear potential in applying such methods for timed analysis to our operational semantics for sequence diagrams with time. The most obvious approach for further work on this would be an adaption of the implementation of the operational semantics to Real-Time Maude [138, 140, 141] and make use of the time models and timed analysis methods provided there.

21.3 Sequence diagrams with data and time

The syntax for sequence diagrams defined by the UML standard [134] allows parameterized messages where values are assigned to named or anonymous attributes of the message. However, the semantics of this is unclear. This handling of data may appear to be more flexible then our approach, but what we loose in flexibility we gain in control over the parameters.

In the MSC standard [85] a treatment of data is defined that is similar to, but

more general than, ours. Variables are owned by lifelines and each lifeline maintains a state that maps the variables to values. An assignment, called a binding, binds an expression to a variable and in the new state the expression is evaluated based on the values of the old state. Bindings can also occur in messages, in which case the variable to which the expression is bound must be owned by the receiving lifeline.

What makes this approach more general than ours is that the bindings may contain wildcards, on both sides of the assignment symbol, in order to facilitate refinement. However, a binding with a wildcard cannot be evaluated without identifying the wildcards, e.g. by substituting arbitrary values. Modifying our data handling to allow a version of the wildcards in bindings in MSC would require rather elaborate rules for updating the data states, as expressions cannot always be evaluated, but it could be a topic for further work.

As is obvious from the case study presented in chapter 20, a number of approaches to specifying real-time properties in sequence diagrams and MSCs exist. The MSC standard [85] defines three timing concepts: The timer that we saw applied in [192] (see section 20.2.6) and [17, 18] (see section 20.2.13), the relative time constraints or relative time delays we saw in [192] (see sections 20.2.4), [7] (see section 20.2.5) and [17, 18] (see section 20.2.13), and the absolute measure or timing that we saw in [192] (see section 20.2.4). The UML standard [134] defines four timing concepts: Duration observation, duration constraint, time observation and time constraint. These are demonstrated by the example reproduced in figure 21.1. The examples from [153] (see section 20.2.2), [106] (see section 20.2.7), [144,145] (see section 20.2.11), and partly the example from [51] (see section 20.2.9), can be considered variants over the duration constraints specified in the UML standard. The timing concepts of the UML Testing Profile [133] are a combination of the timing concepts from the UML standard and the timers from MSC. These are illustrated by the example reproduced in figure 20.23 (see section 20.2.12). In the UML Profile for Schedulability, Performance, and Time [132] timing is specified by timestamps on events. An example of this we get from [41] (see section 20.2.8). This is conceptually similar to the *sendTime* and *receiveTime* attributes of messages we saw in [53] (see sections 20.2.2 and 20.2.3). The UML profile also has the notion of a timer and a notion of a system clock that can produce interrupt events. In addition to the timing concepts listed above, we did in the case study also note two constructs for specifying time constraints in loops. They are found in [51] (see section 20.2.9) and [71] (see section 20.2.10).

What most of these approaches have in common is that they provide basically declarative treatments of time. Even where timing concepts that may be seen as more operational are provided, such as the timers in MSC and the time observation in UML, the semantics given is in a denotational fashion, i.e. without any indication of how time advances.

An exception is [51] where sequence diagrams are translated to timed automata, but in the process they make considerably deviations from what we consider standard UML or MSC semantics. Also in [108], timed MSCs are translated to timed automata, with the restrictions it imposes to translate MSCs to finite automata. Another denotational approach is the formal sematics for Time Sequence Diagrams presented in [47].

There also exist some other more operational approaches. In [62] a time extension to LSCs (see chapter 11) is presented. This is similar to our extension to the operational semantics, in that a clock variable *Time* is added to the formalism. Time is then treated as data and time constraints can be expressed by means of ordinary variables.

In [38], time extensions to the graphical notation of TTCN-3 are introduced, including a **now** variable. In [96] timed MSCs are formalized in a timed version of Maude. This approach, however, only deals with timers, and as explained in chapter 11 their formalization makes restrictions on the MSC semantics.

Some of the publications cited above also describe methods for timed analysis. When sequence diagrams are translated to timed automata they may be subjected to timed model checking, as is done in [51]. In [7] a tool is described that analyzes an MSC for timing conflicts. In [17, 18] an algorithm for time consistency analysis is presented. The work is based on [7], but extends the approach to high-level MSCs. The algorithm is based on constructing weighted graphs from (high-level) MSCs and analyzing the paths of the graphs. A somewhat stronger variant of this kind of analysis of timed MSCs is presented in [195]. In [106] an algorithm is presented that checks time consistency of paths of high-level MSCs by means of linear programming.

In our view, these kinds of analysis are somewhat restricted, as they only analyze consistency of time constraints in sequence diagrams/MSCs. We have more belief in a direction where an operational semantics is connected to more general approaches to timed analysis, as described in section 21.2.

Part VI

Discussion and conclusions

Chapter 22

Discussion

In chapter 2 we formulated the goal for the work of this thesis as follows:

Develop a method and tool to analyze refinements and decompositions of specifications expressed in a diagram based language. The methods and tool should:

- *Handle functional behavior as well as aspects of availability.*
- *Have a sound formal foundation.*
- *Apply testing techniques.*
- *Be practical in use also for people without background in formal methods.*

In addition we formulated 10 success criteria as requirements for the method and tool referred to in the goal.

In this thesis we have developed an operational semantics for sequence diagrams, a refinement verification technique and a refinement testing technique for sequence diagrams, all based on the operational semantics, and an extension of the operational semantics with data and time. These artifacts must together be considered the method referred to in the goal. The operational semantics, the refinement testing and the refinement verification have been implemented in the Escalator tool, described in chapter 14. Escalator is obviously the tool in the goal.

In the following we reproduce the success criteria formulated in chapter 2, and evaluate each of them with respect to the method, as defined above, and the tool. At the end of this chapter, we also review the overall hypothesis we formulated in chapter 2.

Success criterion 1 *The method should support model driven development by offering the capability of analyzing the relation between models or specifications at different levels of abstraction.*

The method provides two techniques for analyzing the relation between specifications at different levels of abstraction, namely the refinement verification and the refinement testing. Specifically, the specifications are UML 2.x sequence diagrams and the relation is refinement as defined in STAIRS [70,159]. The case study documented in chapter 15 demonstrates that our method can be applied to model driven development.

While we have not done work in that direction, we believe the tests we generate from sequence diagram specifications can be applied to more than testing other sequence diagram specifications. The obvious candidates are testing of state machine/statechart specifications, and testing of code (by transforming our abstract tests into tests for testing of code). We would also consider such kinds of application of our tests as "analyzing the relation between models or specifications at different levels of abstraction".

Success criterion 2 *The method should offer the capability of analyzing aspects of availability.*

In chapter 18 we analyzed the aspects of service availability and found the main properties of availability to be exclusivity – broken down to securing access for authorized users, providing denial of service defense mechanisms and ensuring exclusion of unauthorized users – and accessibility – decomposed to quality and timeliness. Our method includes a technique for specifying real-time properties in sequence diagrams and an operational semantics for sequence diagrams with time. These real-time properties are closely related to timeliness. The generality and expressiveness of this technique have been demonstrated in a case study, presented in chapter 20. While we have not implemented any analysis of real-time properties, we believe, as outlined in chapter 21, that existing techniques for doing real-time analysis can be applied to our the operational semantics with time.

We have not done any work in the direction of exclusivity. We have, however, great belief in sequence diagrams as a language for specifying protocols and believe our method can be applied for analysis of authentication and authorization protocols, which are related to exclusivity.

Success criterion 3 *The method should make use of functional models.*

Our method is based on UML 2.x sequence diagrams. Sequence diagram specifications must be considered functional models, and hence we can say that our method make use of functional models.

Success criterion 4 *The method should handle modularity.*

The method does not have any particular support for modularity except from what is inherited from STAIRS. This does not mean that there is no modularity in the method. The refinement relations in STAIRS [70, 159] are modular. Because the refinement verification is based directly on these refinement relations, this makes the refinement verification modular. This is demonstrated by the refinement verification case study documented in chapter 15, by the fact that refinement verification of composed diagrams (the *Overview* diagrams) gave the same result as applying the refinement verification on the composites separately.

This must still be considered modularity in a weak sense because the case study was related to modularity of services and not modularity of components. We do see a potential for extending our method with stronger support for modularity via the refinement testing. In earlier work [109–111] we have demonstrated how testing techniques can be applied for analyzing decomposition of components specified in a contract ori-

ented[1] fashion. We believe something similar can be done based on our refinement testing.

Success criterion 5 *The method should be based on practical testing techniques.*

Part of the method, namely the refinement testing, is based on testing techniques. More specifically, both the test generation and the test execution are based on the testing theory often referred to as ioco theory [177–179]. The refinement testing is implemented in the Escalator tool where both the test generation and the test execution are carried out automatically. The case study on the use of this functionality, documented in chapter 15, demonstrates the practicality of the testing techniques, by demonstrating that the refinement testing can be applied in practice.

Success criterion 6 *The method should be formally sound.*

The operational semantics has been proven to be sound and complete with respect to the denotational semantics of STAIRS. (The proofs are found in appendix A.) The refinement verification is based on the definitions of refinement given in STAIRS. These refinement relations have in [70, 158, 159] been proved to be transitive and monotone, properties that our refinement verification inherits by applying the same definitions. The refinement verification applies the "generate all traces" meta-strategy defined in chapter 9. The soundness of this strategy has been given additional justification in appendix B. The refinement testing is based a formal testing theory [177–179], and in chapter 13 we provide arguments for its soundness with respect to refinement. Regarding the data and time extensions of the operational semantics we have not done any proofs, but in chapter 21 we present arguments for its soundness with respect to STAIRS with data and time. While we do not have formal proofs and irrefutable support for the soundness of all the aspects of the method, the proofs and justifications we have listed above should still provide strong support for its soundness.

Success criterion 7 *The output from the method should useful.*

The method provides four kinds of output:

- Execution traces from the operational semantics with or without data and time. In addition these traces may be categorized as positive and negative traces, or may be structured in interaction obligations.
- From the refinement verification, the answer "yes" or "no" to the question of whether a sequence diagram is a correct refinement of another diagram, or not.
- Tests generated from sequence diagrams.
- Verdicts of test executions with test runs represented as traces. These verdicts also answers the question of whether or not a sequence diagram is a correct refinement of another diagram.

[1] See e.g. [2, 172]. Contract oriented specifications are also called assumption/guarantee specifications, rely/guarantee specifications and assumption/commitment specifications.

In the case study on the use of the Escalator tool, documented in chapter 15, we saw that both the refinement verification and the refinement testing gave results that determined whether the refinement steps of the specification were correct or not. Further we saw that the reasons for unexpected results (i.e. incorrect refinements) could be identified by manual analysis of traces and test runs generated by the tool. In the setting of development by stepwise refinement, these outputs must be considered useful. In the case study we also got the expected results with respect to a manual analysis of the specification, and we have not been able to identify any anomalies in the output.

We also believe the tests themselves can be made useful by applying them to conformance testing, i.e. testing of code. As discussed in chapter 12, iterative development methodologies such as RUP [97] and agile methods [8, 88] recommend scenario based specifications and a short way from specification to testing. In such settings, tests generated from sequence diagrams could be useful.

Because we have not defined or implemented any availability analysis beyond trace generation from sequence diagrams with data and time, it is hard to say something about the usefulness of the output of our method with respect to availability.

Success criterion 8 *The method should be supported by a fully or partially automatic tool.*

As documented in chapter 14, the refinement verification and the refinement testing have been implemented in the Escalator tool. As part of the former, also the operational semantics, with its capabilities of producing execution traces, is implemented. Further, the tool includes a graphical sequence diagram editor and implements a transformation from sequence diagrams in the graphical editor to the internal textual representation of sequence diagrams. The whole process starting with diagrams (made in the editor) to the presentation of results is carried out by the tool without intervention from the user. The only part of the method that is not fully automated is the setup of and analysis of results from the refinement testing, since the "testing down" part and the "testing up" part must be specified and carried out separately.

Success criterion 9 *Application of the method at early stages of the system development improves the quality of systems with availability requirements.*

We have no direct support for this from the work of this thesis. However, we did carry out a case study (see chapter 15) that demonstrates that our method can be applied to stepwise development of sequence diagram specifications, something that would typically be part of the early stages of a system development. There are reasons to believe that this can improve the quality of the specifications and that this again can improve the quality of systems. A real investigation of this would however require empirical studies of a much larger scale than the case studies we have carried out in our work.

Success criterion 10 *Application of the method makes development of systems with availability requirements more efficient and less costly.*

We have no real support to say whether this statement is true or not, as this would require empirical studies at a much larger scale than we have had time and resources to conduct within the work of this thesis.

In chapter 2 we stated our overall hypothesis as follows:

System development will benefit (become more effective and less costly, and produce better software systems) from:

- *Methods and tools for doing analysis of systems at the model or specification level.*
- *An integration of practical or pragmatic methods with formal methods.*

We cannot say that our work implies the validity of this hypothesis, but we can suggest that our work demonstrates the feasibility of making methods and tools for doing analysis at the specification level, and the feasibility of making such methods and tools by the integration of practical methods and formal methods.

Chapter 23

Conclusions

The main contributions of this thesis are:

- An operational semantics for UML 2.x sequence diagrams, including a characterization of how execution strategies for this operational semantics can be defined.
- A method for refinement verification of sequence diagram specifications.
- A method for refinement testing of sequence diagram specifications, including a test generation algorithm and a characterization of test execution.
- A prototype tool for analysis of sequence diagram specifications implementing the operational semantics, the refinement verification and the refinement testing.
- A characterization of the basic aspects of availability and an extension of the operational semantics with support for one of them: real-time properties.

In section 23.1 below we go into more detail about the main contributions. In section 23.2 we discuss the possibilities for further work.

23.1 Main contributions

We are not aware of any other operational semantics for UML 2.x sequence diagrams or MSCs with the same strength and generality as ours. Several other approaches exist, but all with significant shortcomings.

Our operational semantics is simple and is defined with extensibility and variation in mind. An example of this is the possibility of defining the communication model separately from the operational semantics. This extensibility also made it possible to make the data and time extensions with only minor adjustments. It does not involve any translation or transformation of the diagrams into other formalisms, which makes it easy to use and understand. It is sound and complete with respect to a reasonable denotational formalization of the UML standard.

A major insight in the work with the operational semantics was identification of the need for a meta-level to assign properties to the executions and give structure to sets of executions. This is in particular due to the semantic structure of STAIRS with interaction obligations of positive and negative traces. For this reason the operational semantics has a formalized meta-level. This meta-level is used for such things as distinguishing positive from negative traces, and distinguishing between traces of different

interaction obligations. In addition, this meta-level can be be used for defining execution strategies. We have provided two such meta-strategies, one for generating all traces of a diagram and one for generating an arbitrary trace and assigning properties to it. The former is applied in the refinement verification and the latter in the test generation algorithm.

The operational semantics is implemented in the term rewriting language Maude. Maude is itself a formal language, and using Maude for the implementation reduces the risk of unintentionally changing the semantics while doing the implementation. This implementation is one of the cornerstones of the Escalator tool.

In order to analyze real-time properties, the operational semantics was given a simple, but powerful, extension for the handling of data and time. A case study in specification of real-time properties showed that a broad range real-time properties defined in a wide range of approaches to timed sequence diagrams can be specified by our time enhanced sequence diagrams. The operational semantics with time was defined with the same idea of variability as the untimed version of the operational semantics. Therefore the time model can be defined separately from the operational semantics itself, allowing a variety of time models to be implemented.

Based on the operational semantics we have defined and implemented a test generation algorithm and a characterization of test execution. What is special with our approach is that tests are generated from sequence diagram specifications, the tests are themselves sequence diagrams and the tests are executed against sequence diagram specifications. Together this forms a method for refinement testing, i.e. a method for investigating whether or not a sequence diagram is a correct refinement of another sequence diagram by application of testing techniques. We are not aware of any other methods that can do this.

The test generation algorithm is an adaption of a well-known algorithm for generation of tests for conformance testing from LTS specifications. What distinguishes our algorithm is that it takes sequence diagrams that may contain the operators neg/refuse and assert as input. Furhter, it is adapted to the semantic model of UML 2.x which defines sequence diagrams as partial specifications with positive, negative and inconclusive behavior. The output of the test generation algorithm are tests in the format of sequence diagrams. With a small adaption we can use the same operational semantics for tests as for ordinary sequence diagrams. Not surprisingly, test execution is defined by a meta-strategy for this purpose. Also the test execution is adapted to the semantic model with positive, negative and inconclusive traces.

The Escalator tool offers an implementation of the analysis methods that we have defined. The main functionality is obviously the refinement verification and the refinement testing methods, but also trace generation, test generation and test execution are available independently of these. The backbone of the tool is the Maude implementation of the operational semantics. The motivation behind the tool has been to make the analysis methods easily available and practical in use. For this reason the tool integrates a sequence diagram editor that allows the user to work with sequence diagrams in their graphical notation rather than the abstract textual syntax of STAIRS or the Maude representation of diagrams we use as input to the Maude implementation of the operational semantics. All parts of trace generation, refinement verification, test generation and test execution are automated. This means the user only has to care about diagrams in the graphical editor and the results of the analysis, and does not have to be bothered with internal representations and processes. While the tool is a

prototype and is far from industrial standard, the case study we carried out using the tool demonstrated the feasibility of our approach.

23.2 Further work

There are several possible direction for further work. With respect to the operational semantics the most obvious is defining a repository of execution strategies to support a wider variety of analysis. One suggestion could be an execution strategy that searches for a particular trace or traces of a particular pattern. Another possibility of further work on the operational semantics is to define a repository of more sophisticated communication models. An example of this could be defining a lossy communication medium, which could be useful in different kinds of analysis, e.g. protocol analysis.

With respect to the operational semantics with data, a possibility for further work is to make the data handling more general in the direction of the data handling in MSC. Regarding the operational semantics with time, the obvious continuation of the work is to define a more sophisticated time model, or a repository of time models. This would support the work on defining and implementing timed analysis of sequence diagram specifications based on the timed operational semantics, which is the natural next step with respect to availability analysis. Real-Time Maude seems to be a good candidate for implementing time models and timed analysis for our timed operational semantics.

Another possibility of further work with availability is to extend the operational semantics with a notion of probabilities, which would support specification and analysis of soft real-time properties, and not only hard real-time properties. We also believe a possibly interesting direction could be specification and analysis of authentication and authorization protocols, in order to support the exclusivity aspects of availability, and not only the accessibility aspects.

Also with respect to testing there are several possibilities for continuation of the work. At the moment the test generation and test execution, and hence also the refinement testing, do not support the mandatory choice operator xalt from STAIRS. An investigation into the possibilities of extending the testing approach to support xalt, e.g. by some use of test purpose, could be feasible. Another way of increasing the scope of our approach to testing could be to define execution of testing against state machines/statecharts and do refinement testing between sequence diagram specifications and state machine/statechart specifications. We could also define transformations from our abstract tests to implementation specific tests and do conformance testing against code. Both these variants could include establishing a formal relation to the UML Testing Profile.

There are other, more sophisticated, extensions we could make to the refinement testing. A possibility for further work is to define a testing strategy to support the special class of refinements that decompositions constitute. Such a testing strategy should be based on principles from contract oriented specifications (assumption/guarantee specifications) to handle the challenges related to mutually dependencies between the resulting components. Other possibilities that can be considered are to extend the refinement testing with support for data or do timed testing of refinements based on the operational semantics with data and time.

Further work on the Escalator tool will obviously depend on further work on the

analysis methods, and the implementations of these. In addition to this there are several things we could do with the tool at the more practical level. There is a performance problem in the implementation of the refinement testing, and solving this problem is the most pressing issue with respect to improving the quality of the tool. Other obvious improvements of the tool would be to implement such things as syntax checking of diagrams, and improving error messages and other feedback to the user. With respect to implementing new functionality, other than new methods of analysis, we consider import of UML models from third party UML tools, increasing the configurability (e.g. by allowing the users to define their own communication media and execution strategies) and making the tool more interactive (e.g. by providing animations of executions) interesting possibilities.

Finally, a possible direction for further work is to strengthen the empirical support for our hypothesis. A way to do this could be to carry out a larger industrial case study or other experiments with the tools. This would of course require some of the proposed improvements of the tool suggested above.

Bibliography

[1] M. Abadi and L. Lamport. An old-fashion recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, 1994.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[3] S. Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53:225–241, 1987.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[5] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. *IEEE Transactions on Software Engineering*, 29(7):623–633, 2003.

[6] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. *Theoretical Computer Science*, 331(1):97–114, 2005.

[7] R. Alur, G. J. Holzmann, and D. Peled. An analyzer for Message Sequence Charts. In *2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'96)*, number 1055 in Lecture Notes in Computer Science, pages 35–48. Springer, 1996.

[8] S. W. Ambler. *The object primer: Agile model-driven development with UML 2.0*. Cambridge University Press, 3rd edition, 2004.

[9] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of vulnerability: A case study analysis. *IEEE Computer*, 33(12):52–59, 2000.

[10] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based product line engineering with UML*. Addison-Wesley, 2002.

[11] A. Avižienis, J.-C. Laprie, B. Randel, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[12] P. Baker, P. Bristow, C. Jervis, D. King, and B. Mitchell. Automatic generation of conformance tests from Message Sequence Charts. In *SDL and MSC workshop (SAM'02)*, number 2599 in Lecture Notes in Computer Science, pages 170–198. Springer, 2003.

[13] P. Baker, Z. R. Dai, J. Grabowski, Ø. Haugen, I. Schieferdecker, and C. Williams. *Model-driven testing. Using the UML Testing Profile*. Springer, 2008.

[14] M. Barbacci, M. H. Klein, T. A. Longstaff, and C. B. Weinstock. Quality attributes. Technical report CMU/SEI-95TR-021, Software Engineering Institute, Carnegie Mellon University, 1995.

[15] V. R. Basili. The role of experimentation in software engineering: Past, current and future. In *18th International Conference on Software Engineering (ICSE'96)*, pages 442–449. IEEE Computer Society, 1996.

[16] B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold Company, 1984.

[17] H. Ben-Abdallah and S. Leue. Expressing and analysing timing constraints in Message Sequence Chart specifications. Technical report 97-04, Electrical and Computer Engineering, University of Waterloo, 1997.

[18] H. Ben-Abdallah and S. Leue. Timing constraints in Message Sequence Chart specifications. In *Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE X) and Protocol Specification, Testing and Verification (PSTV XVII)*, number 107 in IFIP Conference Proceedings, pages 91–106. Chapman & Hall, 1998.

[19] S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable Petri net models. In *3rd International Workshop on Software and Performance (WOSP'02)*, pages 35–45. ACM Press, 2002.

[20] R. Bræk, J. Gorman, Ø. Haugen, B. Møller-Pedersen, G. Melby, R. Sanders, and T. Stålhane. *TIMe: The Integrated Method. Electronic Textbook v4.0*. SINTEF, 1999.

[21] L. Brandán Briones. *Theories for model-based testing: Real-time and coverage*. PhD thesis, Faculty of Electrical Engineering, Mathematics & Computer Science, University of Twente, 2007.

[22] L. Briand and Y. Labiche. A UML-based approach to system testing. *Software and Systems Modeling*, 1(1):10–42, 2002.

[23] E. Brinksma. A theory for the derivation of tests. In *Protocol Specification, Testing and Verification VIII*, pages 63–74. Elsevier, 1988.

[24] E. Brinksma, L. Heerink, and J. Tretmans. Developments in testing transition systems. In *International Workshop on Testing of Communicating Systems X*, pages 143–166. Chapman & Hall, 1997.

[25] E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In *4th Summer School on Modeling and Verification of Parallel Processes (MOVEP'00)*, number 2067 in Lecture Notes in Computer Science, pages 187–195. Springer, 2001.

[26] M. Broy. The 'grand challenge' in informatics: Engineering software-intensive systems. *IEEE Computer*, 39(10):72–80, 2006.

[27] M. Broy and K. Stølen. *Specification and development of interactive systems. FOCUS on streams, interface, and refinement*. Springer, 2001.

[28] R. Bruni and J. Meseguer. Generalized rewrite theories. In *30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, number 2719 in Lecture Notes in Computer Science, pages 252–266. Springer, 2003.

[29] A. Cavarra and J. Küster-Filipe. Formalizing liveness-enriched sequence diagrams using ASMs. In *11th International Workshop on Abstract State Machines: Advances in Theory and Practice (ASM'04)*, number 3052 in Lecture Notes in Computer Science, pages 67–77. Springer, 2004.

[30] M. V. Cengarle and A. Knapp. UML 2.0 interactions: Semantics and refinement. In *3rd International Workshop on Critical Systems Development with UML (CSD-UML'04)*, pages 85–99. Technische Universität München, 2004.

[31] M. V. Cengarle and A. Knapp. Operational semantics of UML 2.0 interactions. Technical report TUM-I0505, Technische Universität München, 2005.

[32] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[33] D. Clark, W. Lehr, and I. Liu. Provisioning for bursty Internet traffic: Implications for industry and Internet structure. In *MIT ITC Workshop on Internet Quality of Service*, 1999.

[34] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[35] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.2)*. SRI International, Menlo Park, 2005.

[36] B. J. Copeland. Computation. In L. Floridi, editor, *Philosophy of computing and information*, pages 3–17. Blackwell, 2004.

[37] T. Cornford and S. Smithson. *Project research in information systems. A student's guide*. MacMillan, 1996.

[38] Z. R. Dai, J. Grabowski, and H. Neukirchen. TimedTTCN-3 based graphical real-time test specification. In *15th IFIP International Conference on Testing of Communicating Systems (TestCom'03)*, number 2644 in Lecture Notes in Computer Science, pages 110–127. Springer, 2003.

[39] W. Damm and D. Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001.

[40] F. den Braber, M. S. Lund, K. Stølen, and F. Vraalsen. Integrating security in the development process with UML. In *Encyclopedia of Information Science and Technology*, pages 1560–1566. Idea Group, 2005.

[41] B. P. Douglass. *Real time UML*. Addison-Wesley, 3rd edition, 2004.

[42] D. F. D'Souza and A. C. Wills. *Objects, components and frameworks with UML. The Catalysis approach*. Addison-Wesley, 1999.

[43] Eclipse - an open development platform. http://www.eclipse.org/. Accessed 30. June 2007.

[44] A. G. Engels, S. Mauw, and M. A. Reniers. A hierarchy of communication models for Message Sequence Charts. *Science of Computer Programming*, 44(3):253–292, 2002.

[45] P. Enriquez, A. B. Brown, and D. A. Patterson. Lessons from the PSTN for dependable computing. In *Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN'02)*, 2002.

[46] T. Erl. *Service-oriented architecture. Concepts, technology, and design.* Prentice Hall, 2005.

[47] C. Facchi. Formal semantics of Time Sequence Diagrams. Technical report TUM-I9540, Technische Universität München, 1995.

[48] J. K. Feibleman. Pure, science, applied science and technology: An attempt at definitions. In C. Mitcham and R. Mackey, editors, *Philosophy and technology. Readings in the philosophical problems of technology*, pages 33–41. The Free Press, 1983.

[49] N. Fenton. How effective are software engineering methods? *Journal of Systems and Software*, 22(2):141–146, 1993.

[50] K. Finney. Mathematical notation in formal specifications: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, 1996.

[51] T. Firley, M. Huhn, K. Diethers, T. Gehrke, and U. Goltz. Timed sequence diagrams and tool-based analysis – A case study. In *2nd International Conference on The Unified Modeling Language: Beyond the Standard (UML'99)*, number 1723 in Lecture Notes in Computer Science, pages 645–660. Springer, 1999.

[52] F. Fraikin and T. Leonhardt. SeDiTeC – Testing based on sequence diagrams. In *17th IEEE International Conference on Automated Software Engineering (ASE'02)*, pages 261–266. IEEE Computer Society, 2002.

[53] S. Gérard and F. Terrier. UML for real-time. In L. Lavagano, G. Martin, and B. Selic, editors, *UML for real. Design of embedded real-time systems*, pages 17–51. Kluwer, 2003.

[54] R. L. Glass, V. Ramesh, and I. Vessey. An analysis of research in computing disciplines. *Communications of the ACM*, 47(6):89–94, 2004.

[55] R. L. Glass, I. Vessey, and V. Ramesh. Research in software engineering: an analysis of the literature. *Information and Software Technology*, 44:491–506, 2002.

[56] J. Grabowski, B. Koch, M. Schmitt, and D. Hogrefe. SDL and MSC based test generation for distributed test architectures. In *9th International SDL Forum: The Next Millennium (SDL'99)*, pages 389–404. Elsevier, 1999.

[57] P. Graubmann, E. Rudolph, and J. Grabowski. Towards a petri net based semantics for Message Sequence Charts. In *6th International SDL Forum: Using objects (SDL'93)*, pages 179–190. Elsevier, 1993.

[58] R. Grosu and S. A. Smolka. Safety-liveness semantics for UML 2.0 sequence diagrams. In *5th International Conference on Application of Concurrency to System Design (ACSD'05)*, pages 6–14. IEEE Computer Society, 2005.

[59] E. L. Gunter, A. Muscholl, and D. Peled. Compositional Message Sequence Charts. *International Journal on Software Tools for Technology Transfer*, 5(1):78–89, 2003.

[60] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. In *5th International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'06)*, pages 13–19. ACM Press, 2006.

[61] D. Harel and R. Marelly. *Come, let's play: Scenario-based programming using LSCs and the Play-Engine*. Springer, 2003.

[62] D. Harel and P. S. Thiagarajan. Message Sequence Charts. In L. Lavagano, G. Martin, and B. Selic, editors, *UML for real. Design of embedded real-time systems*, pages 77–105. Kluwer, 2003.

[63] J. Harrison. Verification: Industrial application. Working material of Marktoberdorf International Summer School on Proof Technology and Computation, 2003.

[64] J. Hartmanis. Some observations about the nature of computer science. In *13th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'93)*, number 761 in Lecture Notes in Computer Science, pages 1–12. Springer, 1993.

[65] J. Hartmanis. Turing award lecture: On computational complexity and the nature of computer science. *Communications of the ACM*, 37(10):37–43, 1994.

[66] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In *ACM SIGSOFT 2000 International Symposium on Software Testing and Analysis (ISSTA'00)*, number 5 in Software Engineering Notes, pages 60–70, 2000.

[67] Ø. Haugen. Comparing UML 2.0 Interactions and MSC-2000. In *4th International SDL and MSC Workshop: System Analysis and Modeling (SAM'04)*, number 3319 in Lecture Notes in Computer Science, pages 65–79. Springer, 2004.

[68] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. STAIRS towards formal design with sequence diagrams. *Software and Systems Modeling*, 4(4):355–367, 2005. Reprinted as chapter 9 in [153].

[69] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. Why timed sequence diagrams require three-event semantics. In *Scenarios: Models, transformations and tools. International Workshop, Dagstuhl Castle, Germany, September 2003. Revised selected papers*, number 3466 in Lecture Notes in Computer Science, pages 1–25. Springer, 2005.

[70] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. Why timed sequence diagrams require three-event semantics. Research report 309, Department of Informatics, University of Oslo, 2006. Extended and revised version of [69]. Reprinted as chapter 10 in [153].

[71] L. Hélouët. Distributed system requirement modeling with message sequence charts: the case of the PMTP2 protocol. *Information and Software Technology*, 45:701–714, 2003.

[72] A. R. Hevner. The information systems research cycle. *IEEE Computer*, 36(11):111–113, 2003.

[73] S. Heymer. A semantics for MSC based on petri net components. In *4th International SDL and MSC Workshop (SAM'00)*, pages 262–275, 2000.

[74] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[75] C. A. R. Hoare and He Jifeng. *Unifying theories of programming*. Prentice Hall, 1998.

[76] C. A. R. Hoare and J. Misra. Verified software: theories, tools, experiments. In *Verified Software: Theories, Tools, Experiments Conference (VSTTE)*, 2005.

[77] D. Hogrefe, B. Koch, and H. Neukirchen. Some implications of MSC, SDL and TTCN time extensions for computer-aided test generation. In *10th International SDL Forum: Meeting UML (SDL'01)*, number 2078 in Lecture Notes in Computer Science, pages 168–181. Springer, 2001.

[78] International Standards Organization. *ISO 7498-2, Information Processing Systems – Interconnection Reference Model – Part 2: Security Architecture*, 1989.

[79] International Standards Organization. *ISO/IEC 15408, Information technology – Security techniques – Evaluation criteria for IT security*, 1999.

[80] International Standards Organization. *ISO/IEC 17799, Information technology – Code of practice for information security management*, 2000.

[81] International Standards Organization. *ISO/IEC 13335, Information technology – Security techniques – Guidelines for the management of IT security*, 2001.

[82] International Telecommunication Union. *Information technology – Open Systems Interconnection – Basic reference model: Conventions for the definition of OSI services, ITU-T Recommendation X.210*, 1993.

[83] International Telecommunication Union. *Message Sequence Chart (MSC), ITU-T Recommendation Z.120*, 1996.

[84] International Telecommunication Union. *Message Sequence Chart (MSC), ITU-T Recommendation Z.120, Annex B: Formal semantics of Message Sequence Charts*, 1998.

[85] International Telecommunication Union. *Message Sequence Chart (MSC), ITU-T Recommendation Z.120*, 1999.

[86] International Telecommunication Union. *Specification and description language (SDL), ITU-T Recommendation Z.100*, 2000.

[87] J. Jacob. On the derivation of secure components. In *IEEE Symposium on Security and Privacy*, pages 242–247, 1989.

[88] R. Jeffries and G. Melnik. TDD: The art of fearless programming. *IEEE Software*, 24(3):24–30, 2007.

[89] B. Jonsson and G. Padilla. An execution semantics for MSC-2000. In *10th International SDL Forum: Meeting UML (SDL'01)*, number 2078 in Lecture Notes in Computer Science, pages 365–378. Springer, 2001.

[90] E. Jonsson. Towards an integrated conceptual model of security and dependability. In *1st International Conference on Availability, Reliability and Security (ARES'06)*, pages 646–653. IEEE Computer Society, 2006.

[91] N. Jusristo, A. M. Moreno, and S. Vegas. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, 9(1–2):7–44, 2004.

[92] J.-P. Katoen and L. Lambert. Pomsets for Message Sequence Charts. In *Formale Beschreibungstechniken für Verteilte Systeme*, pages 197–208. Shaker, 1998.

[93] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha. Test cases generation from UML state diagrams. *IEE Proceedings – Software*, 146(4):187–192, August 1999.

[94] E. Kindler and A. Martens. Cross-talk revisited: What's the problem? *Petri Net Newsletter*, 58:4–10, 2000.

[95] B. Koch, J. Grabowski, D. Hogrefe, and M. Schmitt. Autolink. A tool for automatic test generation from SDL specifications. In *IEEE International Workshop on Industrial Strength Formal Specification Techniques (WIFT'98)*, 1998.

[96] P. Kosiuczenko and M. Wirsing. Towards an integration of Message Sequence Charts and Timed Maude. *Journal of Integrated Design & Process Science*, 5(1):23–44, 2001.

[97] P. Kruchten. *The Rational Unified Process. An introduction.* Addison-Wesley, 2nd edition, 2000.

[98] J. Küster-Filipe. Modelling concurrent interactions. In *10th International Conference on Algebraic Methodology and Software Technology (AMAST'04)*, number 3116 in Lecture Notes in Computer Science, pages 304–318. Springer, 2004.

[99] P. B. Ladkin and S. Leue. What do Message Sequence Charts mean? In *6th International Conference on Formal Description Techniques (FORTE'93), Formal Description Techniques VI*, pages 301–316. Elsevier, 1994.

[100] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[101] L. Lamport. Real time is really simple. Technical report MSR-TR-2005-30, Microsoft Research, 2005.

[102] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines—a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[103] N. H. Lee and S. D. Cha. Generating test sequences from a set of MSCs. *Computer Networks*, 42:405–417, 2004.

[104] A. A. Letichevsky, J. V. Kapitonova, V. P. Kotlyarov, V. A. Volkov, A. A. Letichevsky, and T. Weigert. Semantics of Message Sequence Charts. In *12th International SDL Forum: Model Driven Systems Design (SDL'05)*, number 3530 in Lecture Notes in Computer Science, pages 117–132. Springer, 2005.

[105] H. R. Lewis and C. H. Papadimitriou. *Elements of the theory of computation*. Prentice Hall, 2nd edition, 1998.

[106] Xuandong Li and J. Lilius. Timing analysis of UML sequence diagrams. In *2nd International Conference on The Unified Modeling Language: Beyond the Standard (UML'99)*, number 1723 in Lecture Notes in Computer Science, pages 661–674. Springer, 1999.

[107] A. Limyr. Graphical editor for UML 2.0 sequence diagrams. Master's thesis, Department of Informatics, University of Oslo, 2005.

[108] P. Lucas. Timed semantics of Message Sequence Charts based on timed automata. *Electronic Notes in Theoretical Computer Science*, 65(6):160–179, 2002.

[109] M. S. Lund. Validation of contract decomposition by testing. Master's thesis, Department of Informatics, University of Oslo, 2002.

[110] M. S. Lund. Validation of contract decomposition by testing. In *Norsk Informatikkonferanse (NIK'02)*, pages 191–202. Tapir, 2002.

[111] M. S. Lund. Testing decomposition of component specifications based on a rule for formal verification. In *3rd International Conference on Quality Software (QSIC'03)*, pages 154–160. IEEE Computer Society, 2003.

[112] M. S. Lund, F. den Braber, and K. Stølen. Maintaining results from security assessments. In *7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 341–350. IEEE Computer Society, 2003.

[113] M. S. Lund and K. Stølen. Deriving tests from UML 2.0 sequence diagrams with neg and assert. In *1st International Workshop on Automation of Software Test (AST'06)*, pages 22–28. ACM Press, 2006.

[114] M. S. Lund and K. Stølen. A fully general operational semantics for UML 2.0 sequence diagrams with potential and mandatory choice. In *14th International Symposium on Formal Methods (FM'06)*, number 4085 in Lecture Notes in Computer Science, pages 380–395. Springer, 2006.

[115] M. S. Lund and K. Stølen. A fully general operational semantics for UML sequence diagrams with potential and mandatory choice. Research report 330, Department of Informatics, University of Oslo, 2007. Extended and revised version of [114].

[116] S. T. March and G. F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266, 1995.

[117] S. Mauw. The formalization of Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(1):1643–1657, 1996.

[118] S. Mauw and M. A. Reniers. An algebraic semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–278, 1994.

[119] S. Mauw and M. A. Reniers. High-level Message Sequence Charts. In *8th International SDL Forum: Time for Testing, SDL, MSC and Trends (SDL'97)*, pages 291–306. Elsevier, 1997.

[120] S. Mauw and M. A. Reniers. Operational semantics for MSC'96. *Computer Networks*, 31(17):1785–1799, 1999.

[121] T. McCombs. *Maude 2.0 Primer, Version 1.0*, 2003.

[122] J. E. McGrath. *Groups: interaction and performance*. Prentice-Hall, 1984.

[123] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[124] J. F. Meyer. Performability evaluations: Where it is and what lies ahead. In *International Computer Performance and Dependability Symposium*, pages 334–343. IEEE Computer Society, 1995.

[125] R. Milner. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer, 1980.

[126] C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. Jézéquel. Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering*, 32(3):140–155, 2006.

[127] A. Newell and H. A. Simon. Computer science as empirical inquiry: Symbols and search. 1975 ACM Turing award lecture. *Communications of the ACM*, 19(3):113–126, 1976.

[128] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.

[129] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[130] OASIS. *Reference model for service oriented architecture 1.0*, 2006. OASIS Standard, 12 October 2006.

[131] Object Management Group. *Unified Modeling Language Specification, version 1.4*, 2001. OMG Document: formal/01-09-67.

[132] Object Management Group. *UML Profile for Schedulability, Performance, and Time Specification, version 1.1*, 2005. OMG Document: formal/2005-01-02.

[133] Object Management Group. *UML Testing Profile, version 1.0*, 2005. OMG Document: formal/2005-07-07.

[134] Object Management Group. *Unified Modeling Language: Superstructure, version 2.1.1 (non-change bar)*, 2005. OMG Document: formal/2007-02-05.

[135] Object Management Group. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*, 2006. OMG Document: formal/2006-05-02.

[136] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *2nd International Conference on The Unified Modeling Language: Beyond the Standard (UML'99)*, number 1723 in Lecture Notes in Computer Science, pages 416–429. Springer, 1999.

[137] M. Okazaki, T. Aoki, and T. Katayama. Formalizing sequence diagrams and state machines using Concurrent Regular Expression. In *2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'03)*, 2003.

[138] P. C. Ølveczky. *Real-Time Maude 2.2 manual*. Department of Informatics, University of Oslo.

[139] P. C. Ølveczky. Formal modeling and analysis of distributed systems in Maude, Lecture Notes INF3230/INF4230. Department of Informatics, University of Oslo, 2005.

[140] P. C. Ølveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002.

[141] P. C. Ølveczky and J. Meseguer. Real-Time maude 2.1. *Electronic Notes in Theoretical Computer Science*, 117:285–314, 2005.

[142] I. Phillips. Refusal testing. *Theoretical Computer Science*, 50:241–284, 1987.

[143] S. Pickin, C. Jard, T. Jéron, J.-M. Jézéquel, and Y. L. Traon. Test synthesis from UML models of distributed software. *IEEE Transactions on Software Engineering*, 33(4):252–268, 2007.

[144] A. Refsdal, K. E. Husa, and K. Stølen. Specification and refinement of soft real-time requirements using sequence diagrams. In *3rd International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'05)*, number 3829 in Lecture Notes in Computer Science, pages 32–48. Springer, 2005.

[145] A. Refsdal, K. E. Husa, and K. Stølen. Specification and refinement of soft real-time requirements using sequence diagrams. Research report 323, Department of Informatics, University of Oslo, 2007. Extended and revised version of [144].

[146] A. Refsdal, R. K. Runde, and K. Stølen. Underspecification, inherent nondeterminism and probability in sequence diagrams. Research report 335, Department of Informatics, University of Oslo, 2006. Extended and revised version of [147]. Reprinted as chapter 14 in [153].

[147] A. Refsdal, R. K. Runde, and K. Stølen. Underspecification, inherent nondeterminism and probability in sequence diagrams. In *8th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'06)*, number 4037 in Lecture Notes in Computer Science, pages 138–155. Springer, 2006.

[148] S. M. Ross. *Introduction to probability models*. Academic Press, 6th edition, 1997.

[149] J. E. Y. Rossebø, M. S. Lund, A. Refsdal, and K. E. Husa. A conceptual model for service availability. In *Quality of Protection: Security Measurements and Metrics (QoP'05)*, number 23 in Advances in Information Security, pages 107–118. Springer, 2006.

[150] J. E. Y. Rossebø, M. S. Lund, A. Refsdal, and K. E. Husa. A conceptual model for service availability. Research report 337, Department of Informatics, University of Oslo, 2006. Extended and revised version of [149].

[151] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language reference manual*. Addison-Wesley, 2nd edition, 2004.

[152] B. Rumpe. Agile test-based modeling. In *International Conference on Software Engineering Research and Practice (SERP'06), volume 1*, pages 10–15. CSREA Press, 2006.

[153] R. K. Runde. *STAIRS - Understanding and developing specifications expressed as UML interaction diagrams*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2007.

[154] R. K. Runde. STAIRS case study: The BuddySync system. Research report 345, Department of Informatics, University of Oslo, 2007. Reprinted as chapter 16 in [153].

[155] R. K. Runde, Ø. Haugen, and K. Stølen. How to transform UML neg into a useful construct. In *Norsk Informatikkonferanse (NIK'05)*, pages 55–66. Tapir, 2005. Reprinted as chapter 11 in [153].

[156] R. K. Runde, Ø. Haugen, and K. Stølen. Refining UML interactions with underspecification and nondeterminism. *Nordic Journal of Computing*, 12(2):175–188, 2005.

[157] R. K. Runde, Ø. Haugen, and K. Stølen. The pragmatics of STAIRS. In *5th International Symposium on Formal Methods for Components and Objects (FMCO'06)*, number 4111 in Lecture Notes in Computer Science, pages 88–114. Springer, 2006. Reprinted as chapter 13 in [153].

[158] R. K. Runde, Ø. Haugen, and K. Stølen. Refining UML interactions with underspecification and nondeterminism. Research report 325, Department of Informatics, University of Oslo, 2007. Revised and extended version of [156]. Reprinted as chapter 12 in [153].

[159] R. K. Runde, A. Refsdal, and K. Stølen. Relating computer systems to sequence diagrams with underspecification, inherent nondeterminism and probabilistic choice. Part 1. Research report 346, Department of Informatics, University of Oslo, 2007. Reprinted as chapter 15 in [153].

[160] I. Schieferdecker, Z. R. Dai, J. Grabowski, and A. Rennoch. The UML Testing Profile and its relation to TTCN-3. In *15th IFIP International Conference on Testing of Communicating Systems (TestCom'03)*, number 2644 in Lecture Notes in Computer Science, pages 79–94. Springer, 2003.

[161] D. A. Schmidt. *Denotational Semantics. A Methodology for Language Development*. wcb, 1988.

[162] Service Availability Forum. *SAF Backgrounder*, 2004.

[163] M. Sgroi, A. Kondratyev, Y. Watanabe, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis of petri nets from Message Sequence Charts specifications for protocol design. In *Design, Analysis and Simulation of Distributed Systems Symposium (DASD'04)*, pages 193–199, 2004.

[164] H. A. Simon. *The sciences of the artificial*. MIT Press, 3rd edition, 1996.

[165] D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanović, N.-K. Liborg, and A. C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753, 2005.

[166] H. Skolimowski. The structure of thinking in technology. In C. Mitcham and R. Mackey, editors, *Philosophy and technology. Readings in the philosophical problems of technology*, pages 42–49. The Free Press, 1983.

[167] D. Sokenou. Ein UML-basierter Testansatz zum Klassen- und Integrationstest objektorientierter Systeme. In *Software Engineering 2005*, number 64 in Lecture Notes in Informatics, pages 91–102. GI-Edition, 2005.

[168] D. Sokenou. Generating test sequences from UML sequence diagrams and state diagrams. In *Informatik 2006: Informatik für Menschen, Band 2*, number 94 in Lecture Notes in Informatics, pages 236–240. GI-Edition, 2006.

[169] I. Solheim and K. Stølen. Technology research explained. Technical report A313, SINTEF Information and Communication Technology, 2007.

[170] I. Sommerville. *Software engineering*. Addison-Wesley, 7th edition, 2004.

[171] Standards Australia. *AS/NZS 4360:1999, Risk Management*, 1999.

[172] K. Stølen. Assumption/commitment rules for dataflow networks — with an emphasis on completeness. In *6th European Symposium on Programming (ESOP'96)*, number 1058 in Lecture Notes in Computer Science, pages 356–372. Springer, 1996.

[173] H. Störrle. Assert, negate and refinement in UML 2 interactions. In *2nd International Workshop on Critical Systems Development with UML (CSD-UML'03)*, pages 79–93. Technische Universität München, 2003.

[174] H. Störrle. Semantics of interaction in UML 2.0. In *IEEE Symposium on Human Centric Computing Languages and Environments (HCC'03)*, pages 129–136. IEEE Computer Society, 2003.

[175] W. F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, 1998.

[176] W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28:9–18, 1995.

[177] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software – Concepts and Tools*, 17(3):103–120, 1996.

[178] J. Tretmans. Testing concurrent systems: A formal approach. In *10th International Conference on Concurrency Theory (CONCUR'99)*, number 1664 in Lecture Notes in Computer Science, pages 46–65. Springer, 1999.

[179] J. Tretmans. Testing techniques. Reader, Universiteit Twente, 2002.

[180] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *7th European International Conference on Software Testing, Analysis & Review (EuroSTAR'99)*, 1999.

[181] J. Tretmans and L. Verhaard. A queue model relating synchronous and asynchronous communication. In *Protocol specification and verification XII*, pages 131–145. Elsevier, 1992.

[182] S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specification and behavior models using implied scenarios. *ACM Transactions on Software Engineering and Methodology*, 13(1):37–85, 2004.

[183] M. Utting and B. Legeard. *Practical model-based testing. A tools approach.* Morgan Kaufman, 2007.

[184] F. W. Vaandrager. Does it pay off? Model-based verification and validation of embedded systems! In *PROGRESS White Papers 2006*, pages 43–66. SWT, 2006.

[185] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. *Electronic Notes in Theoretical Computer Science*, 71:282–300, 2002.

[186] L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On asynchronous testing. In *Protocol Test Systems V*, pages 55–66. Elsevier, 1993.

[187] A. Vogel, B. Kerherve, G. von Bochmann, and J. Gecsei. Distributed multimedia and QoS: A survey. *IEEE Multimedia*, 2(1):10–18, 1995.

[188] J. A. Whittaker. What is software testing? And why is it so hard? *IEEE Software*, 17(1):70–79, 2000.

[189] J. Woodcock. First steps in the verified software grand challenge. *IEEE Computer*, 39(10):57–64, 2006.

[190] R. K. Yin. *Case study research. Design and methods.* Sage, 3rd edition, 2003.

[191] M. V. Zelkowitz and D. R. Wallace. Experimental models for validating technology. *IEEE Computer*, 31(5):23–31, 1998.

[192] T. Zeng, F. Khendek, and L. Hélouët. A semantics for timed MSC. *Electronic Notes in Theoretical Computer Science*, 65(7):85–99, 2002.

[193] T. Zeng, F. Khendek, and B. Parreaux. Refining timed MSCs. In *11th International SDL Forum: System Design (SDL'03)*, number 2707 in Lecture Notes in Computer Science, pages 234–250. Springer, 2003.

[194] T. Zheng and F. Khendek. An execution for MSC-2000 and its application. In *3th International SDL and MSC Workshop: Telecommunications and beyond: The broader applicability of SDL and MSC (SAM'02)*, number 2599 in Lecture Notes in Computer Science, pages 221–232. Springer, 2003.

[195] T. Zheng and F. Khendek. Time consistency of MSC-2000 specifications. *Computer Networks*, 42:303–322, 2003.

Index

Appendix A

Proofs of soundness, completeness and termination of the operational semantics

In this appendix we provide proofs of the soundness and completeness of the operational semantics presented in chapter 8 with respect to the denotational semantics presented in section 5.3. We also prove termination of execution of diagrams without infinite loop.

Informally, the *soundness* property means that that if the operational semantics produces a trace from a given diagram, this trace should be included in the denotational semantics of that diagram. By *completeness* we mean that all traces in the denotational semantics of a diagram should be producible applying the operational semantics on that diagram.

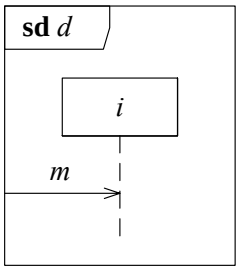
The proofs are based on interpreting the rules of the operational semantics as a rewrite theory. For a formal treatment of rewrite theories, see [28,123]. We assume the communication model with one FIFO buffer for each message as defined in section 8.2.1.

In section A.1 we define a notion of environment of diagrams. In section A.2 we prove soundness and completeness of simple diagrams (theorem 1 on page 279 and theorem 2 on page 293). In section A.3 we prove termination (theorem 3 on page 297), soundness (theorem 4 on page 304) and completeness (theorem 5 on page 306) of diagrams with high-level operators, but without infinite loop. In section A.4 we prove soundness (theorem 6 on page 311) and completeness (theorem 7 on 317) of diagrams with infinite loops.

A.1 Environment

The denotational semantics has an implicit notion of environment, which allows for, e.g., the diagram in figure A.1 which consists of only one input event as long as the lifeline of the corresponding output event does not occur in the diagram. In the operational semantics, the state of the communication medium β in an execution state $[\beta, d]$ may be seen as an explicit notion of the environment. Given the diagram d in figure A.1, the operational semantics produces the trace of $[\![d]\!]$ only if the message m is available in the state of the communication medium, i.e.

$$(m, j, i) \in \beta \Rightarrow [\beta, d] \xrightarrow{(?,(m,j,i))} [\beta \setminus \{(m, j, i)\}, \mathsf{skip}]$$



$$\begin{array}{rcl} d & = & (?,(m,j,i)) \\ [\![\, d \,]\!] & = & \{\langle (?,(m,j,i)) \rangle\} \end{array}$$

Figure A.1: Diagram with single receive event

$$(m,j,i) \notin \beta \Rightarrow [\beta, d] \nrightarrow$$

In order to align the two semantics we must make explicit the environment of the denotational semantics.

Definition 1 (Environment) *The function* $env.\_ \in \mathcal{D} \to \mathbb{P}(\mathcal{E})$ *returns the environment of a diagram* d

$$env.d \stackrel{\mathsf{def}}{=} \{(k,m) | k \in \{!,?\} \wedge m \in msg.d \wedge (k,m) \notin ev.d\}$$

Further we define

$$\begin{array}{rcl} env_{\mathcal{M}}.d & \stackrel{\mathsf{def}}{=} & \{m.e | e \in env.d\} \\ env^k.d & \stackrel{\mathsf{def}}{=} & \{(k,m) | (k,m) \in env.d\} \\ env^k_{\mathcal{M}}.d & \stackrel{\mathsf{def}}{=} & \{m | (k,m) \in env.d\} \end{array}$$

A.2 Simple sequence diagrams

In this section we prove soundness and completeness of what we refer to as *simple diagrams*; diagrams without high-level constructs such as loop and choice. The general case is addressed in section A.3. In section A.3 we also prove termination.

Definition 2 (Simple diagram) *A diagram is called simple iff it only contains*

- *events*
- skip
- *the operators* seq, par *and* strict

In this section we are only concerned with simple diagrams. We also restrict ourselves to diagrams where no event occurs more than once:

$$\begin{array}{rcl} d = d_1 \ \mathsf{seq} \ d_2 & \Rightarrow & ev.d_1 \cap ev.d_2 = \emptyset \\ d = d_1 \ \mathsf{par} \ d_2 & \Rightarrow & ev.d_1 \cap ev.d_2 = \emptyset \\ d = d_1 \ \mathsf{strict} \ d_2 & \Rightarrow & ev.d_1 \cap ev.d_2 = \emptyset \end{array}$$

It follows from this condition that no message can occur in more than one transmit and one receive event.

Lemma 1 *If* d *is a simple diagram, then*

$$[\![\, d \,]\!] = \{(T, \emptyset)\} \quad \text{with} \quad T \subseteq \mathcal{H}$$

Proof of lemma 1
ASSUME: d simple
PROVE: $[\![d]\!] = \{(T, \emptyset)\} \wedge T \subseteq \mathcal{H}$
PROOF: by induction on the structure of d
$\langle 1 \rangle 1$. CASE: $d = \mathsf{skip}$ (induction step)
  $\langle 2 \rangle 1$. $[\![d]\!] = \{(\{\langle\rangle\}, \emptyset)\}$
    PROOF: Def. 5.3.
  $\langle 2 \rangle 2$. $\{\langle\rangle\} \subseteq \mathcal{H}$
    PROOF: Def. of $\mathcal{H}$ and (5.1)
  $\langle 2 \rangle 3$. Q.E.D.
    PROOF: $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$
$\langle 1 \rangle 2$. CASE: $d = e$ (induction step)
  $\langle 2 \rangle 1$. $[\![e]\!] = \{(\{\langle e \rangle\}, \emptyset)\}$
    PROOF: Def. 5.4.
  $\langle 2 \rangle 2$. $\{\langle e \rangle\} \subseteq \mathcal{H}$
    PROOF: Def. of $\mathcal{H}$ and (5.1).
  $\langle 2 \rangle 3$. Q.E.D.
    PROOF: $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$.
$\langle 1 \rangle 3$. $d = d_1$ seq d_2 (induction step)
  ASSUME: 1. $[\![d_1]\!] = \{(T_1, \emptyset)\}$ (induction hypothesis)
    2. $[\![d_2]\!] = \{(T_2, \emptyset)\}$ (induction hypothesis)
    3. $T_1, T_2 \subseteq \mathcal{H}$ (induction hypothesis)
  PROVE: $[\![d_1 \text{ seq } d_2]\!] = \{(T, \emptyset)\} \wedge T \subseteq \mathcal{H}$
  $\langle 2 \rangle 1$. $[\![d_1 \text{ seq } d_2]\!] = \{o_1 \succsim o_2 | o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\}$
    PROOF: Def. (5.7).
  $\langle 2 \rangle 2$. $[\![d_1 \text{ seq } d_2]\!] = \{(T_1, \emptyset) \succsim (T_2, \emptyset)\}$
    PROOF: $\langle 2 \rangle 1$ and induction hypothesis.
  $\langle 2 \rangle 3$. $[\![d_1 \text{ seq } d_2]\!] = \{(T_1 \succsim T_2, (\emptyset \succsim T_2) \cup (\emptyset \succsim \emptyset) \cup (T_1 \succsim \emptyset))\}$
    PROOF: $\langle 2 \rangle 2$ and def. (5.6).
  $\langle 2 \rangle 4$. $[\![d_1 \text{ seq } d_2]\!] = \{(T, \emptyset)\}$, with $T = T_1 \succsim T_2 \subseteq \mathcal{H}$
    $\langle 3 \rangle 1$. $T_1 \succsim T_2$ is a set, and $T_1 \succsim T_2 \subseteq \mathcal{H}$
      PROOF: Def. (5.5).
    $\langle 3 \rangle 2$. $(\emptyset \succsim T_2) \cup (\emptyset \succsim \emptyset) \cup (T_1 \succsim \emptyset) = \emptyset$
      PROOF: Def. (5.5).
    $\langle 3 \rangle 3$. Q.E.D.
      PROOF: $\langle 2 \rangle 3$, $\langle 3 \rangle 1$ and $\langle 3 \rangle 2$.
  $\langle 2 \rangle 5$. Q.E.D.
    PROOF: $\langle 2 \rangle 4$
$\langle 1 \rangle 4$. $d = d_1$ par d_2 (induction step)
  ASSUME: 1. $[\![d_1]\!] = \{(T_1, \emptyset)\}$ (induction hypothesis)
    2. $[\![d_2]\!] = \{(T_2, \emptyset)\}$ (induction hypothesis)
    3. $T_1, T_2 \subseteq \mathcal{H}$ (induction hypothesis)
  PROVE: $[\![d_1 \text{ par } d_2]\!] = \{(T, \emptyset)\} \wedge T \subseteq \mathcal{H}$
  $\langle 2 \rangle 1$. $[\![d_1 \text{ par } d_2]\!] = \{o_1 \parallel o_2 | o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\}$
    PROOF: Def. (5.10).
  $\langle 2 \rangle 2$. $[\![d_1 \text{ par } d_2]\!] = \{(T_1, \emptyset) \parallel (T_2, \emptyset)\}$
    PROOF: $\langle 2 \rangle 1$ and induction hypothesis.
  $\langle 2 \rangle 3$. $[\![d_1 \text{ par } d_2]\!] = \{(T_1 \parallel T_2, (\emptyset \parallel T_2) \cup (\emptyset \parallel \emptyset) \cup (T_1 \parallel \emptyset))\}$

PROOF: ⟨2⟩2 and def. (5.9).
⟨2⟩4. $[\![d_1 \text{ par } d_2]\!] = \{(T, \emptyset)\}$, with $T = T_1 \parallel T_2 \subseteq \mathcal{H}$
⟨3⟩1. $T_1 \parallel T_2$ is a set, and $T_1 \parallel T_2 \subseteq \mathcal{H}$
PROOF: Def. (5.8).
⟨3⟩2. $(\emptyset \parallel T_2) \cup (\emptyset \parallel \emptyset) \cup (T_1 \parallel \emptyset) = \emptyset$
PROOF: Def. (5.8).
⟨3⟩3. Q.E.D.
PROOF: ⟨2⟩3, ⟨3⟩1 and ⟨3⟩2.
⟨2⟩5. Q.E.D.
PROOF: ⟨2⟩4
⟨1⟩5. $d = d_1 \text{ strict } d_2$ (induction step)
ASSUME: 1. $[\![d_1]\!] = \{(T_1, \emptyset)\}$ (induction hypothesis)
2. $[\![d_2]\!] = \{(T_2, \emptyset)\}$ (induction hypothesis)
3. $T_1, T_2 \subseteq \mathcal{H}$ (induction hypothesis)
PROVE: $[\![d_1 \text{ strict } d_2]\!] = \{(T, \emptyset)\} \wedge T \subseteq \mathcal{H}$
⟨2⟩1. $[\![d_1 \text{ strict } d_2]\!] = \{o_1 \succ o_2 | o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\}$
PROOF: Def. (5.13).
⟨2⟩2. $[\![d_1 \text{ strict } d_2]\!] = \{(T_1, \emptyset) \succ (T_2, \emptyset)\}$
PROOF: ⟨2⟩1 and induction hypothesis.
⟨2⟩3. $[\![d_1 \text{ strict } d_2]\!] = \{(T_1 \succ T_2, (\emptyset \succ T_2) \cup (\emptyset \succ \emptyset) \cup (T_1 \succ \emptyset))\}$
PROOF: ⟨2⟩2 and def. (5.12).
⟨2⟩4. $[\![d_1 \text{ strict } d_2]\!] = \{(T, \emptyset)\}$, with $T = T_1 \succ T_2 \subseteq \mathcal{H}$
⟨3⟩1. $T_1 \succ T_2$ is a set, and $T_1 \succ T_2 \subseteq \mathcal{H}$
PROOF: Def. (5.11).
⟨3⟩2. $(\emptyset \succ T_2) \cup (\emptyset \succ \emptyset) \cup (T_1 \succ \emptyset) = \emptyset$
PROOF: Def. (5.11).
⟨3⟩3. Q.E.D.
PROOF: ⟨2⟩3, ⟨3⟩1 and ⟨3⟩2.
⟨2⟩5. Q.E.D.
PROOF: ⟨2⟩4
⟨1⟩6. Q.E.D.
PROOF: ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4 and ⟨1⟩5.

□

Given lemma 1 we will in the following associate $[\![d]\!]$ with T, and write $t \in [\![d]\!]$ when $t \in T$ as shorthand notation.

Lemma 2 *Given two simple diagrams $d_1, d_2 \in \mathcal{D}$, then*

$$env.(d_1 \text{ seq } d_2) = env.(d_1 \text{ par } d_2) = env.(d_1 \text{ strict } d_2) = (env.d_1 \backslash ev.d_2) \cup (env.d_2 \backslash ev.d_1)$$

Proof of lemma 2
⟨1⟩1. $env.(d_1 \text{ seq } d_2)$

$$\begin{aligned} &= \{(k,m) \mid k \in \{!,?\} \wedge m \in msg.(d_1 \text{ seq } d_2) \\ &\qquad \wedge (k,m) \notin ev.(d_1 \text{ seq } d_2)\} && \text{(Def. 1 of } env.\_) \\ &= \{(k,m) \mid k \in \{!,?\} \wedge m \in (msg.d_1 \cup msg.d_2) \\ &\qquad \wedge (k,m) \notin (ev.d_1 \cup ev.d_2)\} && \text{(Defs. of } ev.\_ \text{ and } msg.\_) \end{aligned}$$

$\langle 1\rangle 2.\ env.(d_1 \text{ par } d_2)$
$= \{(k,m) \mid k \in \{!,?\} \wedge m \in msg.(d_1 \text{ par } d_2)$
$\wedge (k,m) \notin ev.(d_1 \text{ par } d_2)\}$ (Def. 1 of $env.\_$)
$= \{(k,m) \mid k \in \{!,?\} \wedge m \in (msg.d_1 \cup msg.d_2)$
$\wedge (k,m) \notin (ev.d_1 \cup ev.d_2)\}$ (Defs. of $ev.\_$ and $msg.\_$)
$\langle 1\rangle 3.\ env.(d_1 \text{ strict } d_2)$
$= \{(k,m) \mid k \in \{!,?\} \wedge m \in msg.(d_1 \text{ strict } d_2)$
$\wedge (k,m) \notin ev.(d_1 \text{ strict } d_2)\}$ (Def. 1 of $env.\_$)
$= \{(k,m) \mid k \in \{!,?\} \wedge m \in (msg.d_1 \cup msg.d_2)$
$\wedge (k,m) \notin (ev.d_1 \cup ev.d_2)\}$ (Defs. of $ev.\_$ and $msg.\_$)
$\langle 1\rangle 4.\ \{(k,m) \mid k \in \{!,?\} \wedge m \in (msg.d_1 \cup msg.d_2) \ \wedge (k,m) \notin (ev.d_1 \cup ev.d_2)\}$
$= \{(k,m) \mid k \in \{!,?\} \wedge (m \in msg.d_1 \vee m \in msg.d_2)$
$\wedge (k,m) \notin ev.d_1 \wedge (k,m) \notin ev.d_2\}$
$= \{(k,m) \mid (k \in \{!,?\} \wedge m \in msg.d_1 \wedge (k,m) \notin ev.d_1 \wedge (k,m) \notin ev.d_2)$
$\vee (k \in \{!,?\} \wedge m \in msg.d_2 \wedge (k,m) \notin ev.d_1 \wedge (k,m) \notin ev.d_2)$
$= \{(k,m) \mid k \in \{!,?\} \wedge m \in msg.d_1 \wedge (k,m) \notin ev.d_1 \wedge (k,m) \notin ev.d_2\}$
$\cup \{(k,m) \mid k \in \{!,?\} \wedge m \in msg.d_2 \wedge (k,m) \notin ev.d_1 \wedge (k,m) \notin ev.d_2\}$
$= (\{(k,m) \mid k \in \{!,?\} \wedge m \in msg.d_1 \wedge (k,m) \notin ev.d_1\} \cap$
$\{(k,m) \mid (k,m) \notin ev.d_2\}) \cup$
$(\{(k,m) \mid k \in \{!,?\} \wedge m \in msg.d_2 \wedge (k,m) \notin ev.d_2\} \cap$
$\{(k,m) \mid (k,m) \notin ev.d_1\})$
$= (\{(k,m) \mid k \in \{!,?\} \wedge m \in msg.d_1 \wedge (k,m) \notin ev.d_1\} \setminus$
$\overline{\{(k,m) \mid (k,m) \notin ev.d_2\}}) \cup$
$(\{(k,m) \mid k \in \{!,?\} \wedge m \in msg.d_2 \wedge (k,m) \notin ev.d_2\} \setminus$
$\overline{\{(k,m) \mid (k,m) \notin ev.d_1\}})$
$= (env.d_1 \setminus \{(k,m) \mid (k,m) \in ev.d_2\})$
$\cup (env.d_2 \setminus \{(k,m) \mid (k,m) \in ev.d_1\})$
$= (env.d_1 \setminus ev.d_2) \cup (env.d_2 \setminus ev.d_1)$
PROOF: Def. of $env.\_$, and set theory.
$\langle 1\rangle 5.$ Q.E.D.
PROOF: $\langle 1\rangle 1$, $\langle 1\rangle 2$, $\langle 1\rangle 3$ and $\langle 1\rangle 4$.

□

Definition 3 *The function $ev.\_ \in \mathcal{E}^* \to \mathbb{P}(\mathcal{E})$ returns the events of a trace t. The function is defined recursively:*

$$ev.\langle\rangle \stackrel{\text{def}}{=} \emptyset$$
$$ev.(\langle e\rangle^\frown t) \stackrel{\text{def}}{=} \{e\} \cup ev.t$$

Lemma 3 *Given $t_1, t_2 \in \mathcal{E}^*$, then $ev.(t_1{}^\frown t_2) = ev.t_1 \cup ev.t_2$*

Proof of lemma 3
ASSUME: $t_1, t_2 \in \mathcal{E}^*$
PROVE: $ev.(t_1{}^\frown t_2) = ev.t_1 \cup ev.t_2$
PROOF: by induction on t_1.
$\langle 1\rangle 1.$ Induction start. $t_1 = \langle\rangle$
$\langle 2\rangle 1.\ ev.(t_1{}^\frown t_2) = ev.(\langle\rangle^\frown t_2) = ev.t_2$

PROOF: ⟨1⟩1 and identity of $\_^\frown\_$.

⟨2⟩2. $ev.t_1 \cup ev.t_2 = ev.\langle\rangle \cup ev.t_2 = \emptyset \cup ev.t_2 = ev.t_2$

PROOF: ⟨1⟩1 and def. of $ev.\_$.

⟨2⟩3. Q.E.D.

PROOF: ⟨2⟩1 and ⟨2⟩2.

⟨1⟩2. Induction step

ASSUME: $ev.(t_1{}^\frown t_2) = ev.t_1 \cup ev.t_2$ (induction hypothesis)

PROVE: $ev.((\langle e\rangle{}^\frown t_1){}^\frown t_2) = ev.(\langle e\rangle{}^\frown t_1) \cup ev.t_2$

⟨2⟩1. $ev.((\langle e\rangle{}^\frown t_1){}^\frown t_2) = ev.(\langle e\rangle{}^\frown (t_1{}^\frown t_2)) = \{e\} \cup ev.(t_1{}^\frown t_2)$
$= \{e\} \cup ev.t_1 \cup ev.t_2$

PROOF: Associativity of $\_^\frown\_$, def. of $ev.\_$ and induction hypothesis.

⟨2⟩2. $ev.(\langle e\rangle{}^\frown t_1) \cup ev.t_2 = \{e\} \cup ev.t_1 \cup ev.t_2$

PROOF: Def. of $ev.\_$.

⟨2⟩3. Q.E.D.

PROOF: ⟨2⟩1 and ⟨2⟩2.

⟨1⟩3. Q.E.D.

⟨1⟩1 and ⟨1⟩2.

□

Lemma 4 *Given simple diagrams $d, d' \in \mathcal{D}$ without repetition of events. Then, for all traces t, if there exist $\beta, \beta' \in \mathcal{B}$ such that*

$$[\beta, d] \xrightarrow{t} [\beta', d']$$

then

$$ev.t = ev.d \setminus ev.d'$$

Proof of lemma 4

ASSUME: $[\beta, d] \xrightarrow{t} [\beta', d']$

PROVE: $e \in ev.t \iff e \in (ev.d \setminus ev.d')$

⟨1⟩1. $e \in (ev.d \setminus ev.d') \iff (e \in ev.d \wedge e \notin ev.d')$

PROOF: Set theory.

⟨1⟩2. $e \in ev.t \Rightarrow e \in ev.d \wedge e \notin ev.d'$

PROOF: Assume $e \in ev.t$. Then there must exist t_1, t_2 such that $t = t_1{}^\frown\langle e\rangle{}^\frown t_2$ and $\beta_1, \beta_2, d_1, d_2$ such that

$$[\beta, d] \xrightarrow{t_1} [\beta_1, d_1] \xrightarrow{e} [\beta_2, d_2] \xrightarrow{t_2} [\beta', d']$$

By the assumption that d, and therefore also d_1 and d_2, are simple and have no repetition of events, and by rules (8.3)-(8.14) we must have that $e \in ev.d_1$ and $e \notin ev.d_2$. Because events cannot reappear during rewrite of a diagram, this also means that $e \in ev.d$ and $e \notin ev.d'$

⟨1⟩3. $e \in ev.d \wedge e \notin ev.d' \Rightarrow e \in ev.t$

PROOF: Assume $e \in ev.d$ and $e \notin ev.d'$. Because application of rules (8.3)-(8.14) is the only way of removing an event there must exist a sequence of rewrites such that

$$[\beta, d] \xrightarrow{t_1{}^\frown\langle e\rangle{}^\frown t_2} [\beta', d']$$

which means there must exist t_1 and t_2 such that $t = t_1{}^\frown\langle e\rangle{}^\frown t_2$. By the definition of $ev.t$ we then have that $e \in ev.t$.

⟨1⟩4. Q.E.D.
PROOF: ⟨1⟩1, ⟨1⟩2 and ⟨1⟩3.

□

Lemma 5 *Given simple diagram $d \in \mathcal{D}$. For all traces t, if there exist $\beta, \beta' \in \mathcal{B}$ such that*

$$[\beta, d] \xrightarrow{t} [\beta', \mathsf{skip}]$$

then

$$ev.t = ev.d$$

Proof of lemma 5 Assume $[\beta, d] \xrightarrow{t} [\beta', \mathsf{skip}]$. By lemma 4 and the definition of $ev.\mathsf{skip}$, we have

$$ev.t = ev.d \setminus ev.\mathsf{skip} = ev.d \setminus \emptyset = ev.d$$

□

Definition 4 *We formally define the filtering operator $\_ \circledS \_$ for finite sequences over A:*

$$\begin{array}{rcl} V \circledS \langle\rangle & \stackrel{\mathsf{def}}{=} & \langle\rangle \\ v \in V \Rightarrow V \circledS (\langle v \rangle ^\frown t) & \stackrel{\mathsf{def}}{=} & \langle v \rangle ^\frown (V \circledS t) \\ v \notin V \Rightarrow V \circledS (\langle v \rangle ^\frown t) & \stackrel{\mathsf{def}}{=} & V \circledS t \end{array}$$

(with $V \subseteq A$ and $t \in A^*$).

Lemma 6 *Given simple diagram $d \in \mathcal{D}$. For all $t \in \mathcal{E}^*$, if there exists β such that*

$$[env^!_{\mathcal{M}}.d, d] \xrightarrow{t} [\beta, \mathsf{skip}]$$

then

$$t \in \mathcal{H}$$

Proof of lemma 6
ASSUME: $[env^!_{\mathcal{M}}.d, d] \xrightarrow{t} [\beta, \mathsf{skip}]$
PROVE: $t \in \mathcal{H}$
⟨1⟩1. $t \in \mathcal{H} \iff$
$\forall i \in [1..\#t] : k.t[i] = ! \Rightarrow \#(\{(!, m.t[i])\} \circledS t|_i) > \#(\{(?, m.t[i])\} \circledS t|_i)$
PROOF: Semantic constraint on traces (5.1).
⟨1⟩2. $\forall i \in [1..\#t] : k.t[i] = ! \Rightarrow \#(\{(!, m.t[i])\} \circledS t|_i) > \#(\{(?, m.t[i])\} \circledS t|_i)$
LET: j be arbitrary $j \in [1..\#t]$
PROVE: $k.t[j] = ! \Rightarrow \#(\{(!, m.t[j])\} \circledS t|_j) > \#(\{(?, m.t[j])\} \circledS t|_j)$
⟨2⟩1. CASE: $t[j]$ transmit event
LET: $t[j] = (!, m)$. Then $k.t[j] = !$ and $m.t[j] = m$
PROVE: $\#(\{(!, m)\} \circledS t|_j) > \#(\{(?, m)\} \circledS t|_j)$
⟨3⟩1. $\#(\{(!, m)\} \circledS t|_j) = 1$
PROOF: The assumptions that d is simple and has no repetition of events, and lemma 5.
⟨3⟩2. $\#(\{(?, m)\} \circledS t|_j) = 0$
PROOF: by contradiction

ASSUME: $\#(\{(?,m)\}\circledS t|_j) > 0$
$\langle 4\rangle 1$. There exist s, s' such that
$t|_j = s^\frown\langle(?,m)\rangle^\frown s'$
PROOF: Defs. of $\#\_$ and $\_\circledS\_$.
$\langle 4\rangle 2$. There exist $\beta', \beta'', \beta''' \in \mathcal{B}, d', d'', d''' \in \mathcal{D}$ such that
$[env^!_{\mathcal{M}}.d, d] \xrightarrow{s} [\beta', d'] \xrightarrow{(?,m)} [\beta'', d''] \xrightarrow{s'} [\beta''', d''']$
PROOF: Assumption and $\langle 4\rangle 1$.
$\langle 4\rangle 3$. $m \in \beta'$
PROOF: $\langle 4\rangle 2$ and rules (8.3)-(8.14).
$\langle 4\rangle 4$. $m \notin env^!_{\mathcal{M}}.d$
PROOF: $(!,m) \in ev.t = ev.d$ (by assumption, and lemma 5) and def. 1 of $env^!_{\mathcal{M}}.\_$.
$\langle 4\rangle 5$. $(!,m) \in ev.s$
PROOF: $\langle 4\rangle 3$, $\langle 4\rangle 4$ and rules (8.3)-(8.14).
$\langle 4\rangle 6$. $\#(\{(!,m)\}\circledS t|_{j-1}) = 0$
PROOF: $\langle 3\rangle 1$ and defs. of $\#\_$ and $\_\circledS\_$ and assumption that $t[j] = (!,m)$.
$\langle 4\rangle 7$. $(!,m) \notin ev.t|_{j-1}$
PROOF: $\langle 4\rangle 6$ and def. of $\_\circledS\_$.
$\langle 4\rangle 8$. $(!,m) \notin ev.s$
PROOF: $\langle 4\rangle 7$, $\langle 4\rangle 1$ and def. of $ev.\_$.
$\langle 4\rangle 9$. Q.E.D.
PROOF: $\langle 4\rangle 5$ and $\langle 4\rangle 8$ yield a contradiction, so we must have that $\#(\{(?,m)\}\circledS t|_j) = 0$
$\langle 3\rangle 3$. Q.E.D.
PROOF: $\langle 3\rangle 1$ and $\langle 3\rangle 2$
$\langle 2\rangle 2$. CASE: $t[j]$ receive event
LET: $t[j] = (?,m)$
$\langle 3\rangle 1$. Q.E.D.
PROOF: Trivial since $k.t[j] = ?$ and therefore $(k.t[j] = !) = \mathbf{false}$.
$\langle 2\rangle 3$. Q.E.D.
PROOF: $\langle 2\rangle 1$ and $\langle 2\rangle 2$.
$\langle 1\rangle 3$. Q.E.D.
PROOF:$\langle 1\rangle 1$ and $\langle 1\rangle 2$.

□

Lemma 7 *Given simple diagrams $d_1, d_2 \in \mathcal{D}$. For all traces t, if there exists $\beta \in \mathcal{B}$ such that*

$$[env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1 \text{ seq } d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$$

then there exist traces t_1, t_2 and $\beta_1, \beta_2 \in \mathcal{B}$ such that

$$\begin{aligned}
&[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}] \wedge \\
&[end^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}] \wedge \\
&\forall l \in \mathcal{L} : e.l\circledS t = e.l\circledS t_1{}^\frown e.l\circledS t_2
\end{aligned}$$

Proof of lemma 7

ASSUME: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1 \text{ seq } d_2] \xrightarrow{t} [\beta, \text{skip}]$

PROVE: There exist traces t_1, t_2, and $\beta_1, \beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \text{skip}] \wedge$
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \text{skip}] \wedge$
$\forall l \in \mathcal{L} : e.l \circledS t = e.l \circledS t_1 \frown e.l \circledS t_2$

⟨1⟩1. There exist trace t_1, t_2, and $\beta_1, \beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \text{skip}] \wedge$
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \text{skip}]$

⟨2⟩1. LET: $t = \langle e \rangle \frown t'$

⟨2⟩2. There exist $\beta'_1, \beta'_2 \in \mathcal{B}, d'_1, d'_2 \in \mathcal{D}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{e} [\beta'_1, d'_1] \veebar$
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{e} [\beta'_2, d'_2]$
(where $\veebar$ denotes exclusive or)

⟨3⟩1. There exist $\beta' \in \mathcal{B}, d' \in \mathcal{D}$ such that
$[env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1 \text{ seq } d_2] \xrightarrow{e} [\beta', d'] \xrightarrow{t'} [\beta, \text{skip}]$
PROOF: Assumption and ⟨2⟩1

⟨3⟩2. $[env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1 \text{ seq } d_2] \xrightarrow{e} [\beta', d'_1 \text{ seq } d_2] \veebar$
$[env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1 \text{ seq } d_2] \xrightarrow{e} [\beta', d_1 \text{ seq } d'_2]$
PROOF: ⟨3⟩1, rules (8.3), (8.11) and (8.12), the assumption $ev.d_1 \cap ev.d_2 = \emptyset$ and lemma 4.

⟨3⟩3. CASE: $[env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1 \text{ seq } d_2] \xrightarrow{e} [\beta', d'_1 \text{ seq } d_2]$
PROVE: $[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{e} [\beta'_1, d'_1] \wedge [env^!_{\mathcal{M}}.d_2, d_2] \stackrel{e}{\not\rightarrow}$

⟨4⟩1. $[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{e} [\beta'_1, d'_1]$

⟨5⟩1. $\Pi(ll.(d_1 \text{ seq } d_2), env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1 \text{ seq } d_2) \xrightarrow{e}$
$\Pi(ll.(d_1 \text{ seq } d_2), env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d'_1 \text{ seq } d_2)$
PROOF: By rule (8.3) this is a necessary condition for ⟨3⟩3 to be satisfied.

⟨5⟩2. $\Pi(ll.d_1 \cap ll.(d_1 \text{ seq } d_2), env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1) \xrightarrow{e}$
$\Pi(ll.d_1 \cap ll.(d_1 \text{ seq } d_2), env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d'_1)$
PROOF: By rule (8.11) this is a necessary condition for ⟨5⟩1 to be satisfied.

⟨5⟩3. $\Pi(ll.d_1, env^!_{\mathcal{M}}.d_1, d_1) \xrightarrow{e} \Pi(ll.d_1, env^!_{\mathcal{M}}.d_1, d'_1)$

⟨6⟩1. $ll.d_1 \cap ll.(d_1 \text{ seq } d_2) = ll.d_1 \cap (ll.d_1 \cup ll.d_2) = ll.d_1$
PROOF: Def. of $ll.\_$

⟨6⟩2. CASE: $e = (!, m)$
It follows from the rules that the state of the communication medium is irrelevant.

⟨6⟩3. CASE: $e = (?, m)$
In this case it must be shown that $m \in env^!_{\mathcal{M}}.d_1$.

⟨7⟩1. $m \in msg.d_1$
PROOF: Defs. of $ev.\_$ and $msg.\_$, rules (8.10)-(8.14), ⟨6⟩3 and ⟨5⟩2.

⟨7⟩2. $m \in env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2)$
PROOF: It follows from the recursive use of rules (8.3)-(8.14), and def. of *ready* that this is a necessary condition for ⟨3⟩3 to be satisfied.

⟨7⟩3. $m \in env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2) \iff m \in env^!_{\mathcal{M}}.d_1 \wedge (!, m) \notin ev.d_2$
PROOF: $m \in env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2)$

$\Leftrightarrow$ $(!, m) \in env.(d_1 \text{ seq } d_2)$ (Def. 1)
$\Leftrightarrow$ $(!, m) \in (env.d_1 \setminus ev.d_2) \cup (env.d_2 \setminus ev.d_1)$ (Lemma 2)
$\Leftrightarrow$ $((!, m) \in env.d_1 \wedge (!, m) \notin ev.d_2) \vee$
$((!, m) \in env.d_2 \wedge (!, m) \notin ev.d_1)$
$\Leftrightarrow$ $((!, m) \in env.d_1 \wedge (!, m) \notin ev.d_2) \vee$
$((!, m) \in env.d_2 \wedge (!, m) \in env.d_1)$ ($\langle 7 \rangle 1$)
$\Leftrightarrow$ $(!, m) \in env.d_1 \wedge$
$((!, m) \notin ev.d_2 \vee (!, m) \in env.d_2)$
$\Leftrightarrow$ $(!, m) \in env.d_1 \wedge ((!, m) \notin ev.d_2 \vee$
$((!, m) \notin ev.d_2 \wedge m \in msg.d_2))$ (Def. 1)
$\Leftrightarrow$ $(!, m) \in env.d_1 \wedge (!, m) \notin ev.d_2$
$\Leftrightarrow$ $m \in env^!_{\mathcal{M}}.d_1 \ \wedge (!, m) \notin ev.d_2$ (Def. 1)

$\langle 7 \rangle 4$. Q.E.D.
PROOF: $\langle 7 \rangle 2$ and $\langle 7 \rangle 3$.

$\langle 6 \rangle 4$. Q.E.D.
PROOF: $\langle 5 \rangle 2$, $\langle 6 \rangle 1$, $\langle 6 \rangle 2$ and $\langle 6 \rangle 3$.

$\langle 5 \rangle 4$. Q.E.D.
PROOF: $\langle 5 \rangle 3$ and rule (8.3)

$\langle 4 \rangle 2$. $[env^!_{\mathcal{M}}.d_2, d_2] \stackrel{e}{\nrightarrow}$
$\langle 5 \rangle 1$. $e \notin ev.d_2$
PROOF: $\langle 3 \rangle 3$ and application of rules imply $e \in ev.d_1$, and by assumption, $ev.d_1 \cap ev.d_2 = \emptyset$.
$\langle 5 \rangle 2$. $e \notin ev.d_2 \setminus ev.d'_2$
PROOF: $\langle 5 \rangle 1$.
$\langle 5 \rangle 3$. $\neg([env^!_{\mathcal{M}}.d_2, d_2] \stackrel{e}{\longrightarrow} [\beta'_2, d'_2])$
PROOF: $\langle 5 \rangle 2$ and lemma 4.
$\langle 5 \rangle 4$. Q.E.D.
PROOF: $\langle 5 \rangle 3$

$\langle 4 \rangle 3$. Q.E.D.
PROOF: $\langle 4 \rangle 1$ and $\langle 4 \rangle 2$

$\langle 3 \rangle 4$. CASE: $[env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1 \text{ seq } d_2] \stackrel{e}{\longrightarrow} [\beta', d_1 \text{ seq } d'_2]$
PROVE: $[env^!_{\mathcal{M}}.d_2, d_2] \stackrel{e}{\longrightarrow} [\beta'_2, d'_2] \wedge [env^!_{\mathcal{M}}.d_1, d_1] \stackrel{e}{\nrightarrow}$

$\langle 4 \rangle 1$. $[env^!_{\mathcal{M}}.d_2, d_2] \stackrel{e}{\longrightarrow} [\beta'_2, d'_2]$
$\langle 5 \rangle 1$. $\Pi(ll.(d_1 \text{ seq } d_2), env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1 \text{ seq } d_2) \stackrel{e}{\longrightarrow}$
$\Pi(ll.(d_1 \text{ seq } d_2), env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1 \text{ seq } d'_2)$
PROOF: By rule (8.3) this is a necessary condition for $\langle 3 \rangle 4$ to be satisfied.
$\langle 5 \rangle 2$. $\Pi(ll.(d_1 \text{ seq } d_2) \setminus ll.d_1, env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_2) \stackrel{e}{\longrightarrow}$
$\Pi(ll.(d_1 \text{ seq } d_2) \setminus ll.d_2, env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d'_2)$
PROOF: By rule (8.12) this is a necessary condition for $\langle 5 \rangle 1$ to be satisfied.
$\langle 5 \rangle 3$. $\Pi(ll.d_2, env^!_{\mathcal{M}}.d_2, d_2) \stackrel{e}{\longrightarrow} \Pi(ll.d_2, env^!_{\mathcal{M}}.d_2, d'_2)$
$\langle 6 \rangle 1$. $l.e \in ll.d_2$
$\langle 7 \rangle 1$. $l.e \in ll.(d_1 \text{ seq } d_2) \setminus ll.d_1$
PROOF: This follows from $\langle 5 \rangle 2$ and the rules.
$\langle 7 \rangle 2$. $l.e \in ll.(d_1 \text{ seq } d_2) \setminus ll.d_1 \Rightarrow l.e \in (ll.d_1 \cup ll.d_2) \wedge l.e \notin ll.d_1 \Rightarrow (l.e \in ll.d_1 \vee l.e \in ll.d_2) \wedge l.e \notin ll.d_1 \Rightarrow l.e \in ll.d_2$
PROOF: Def. of $ll.\_$ and $\langle 7 \rangle 1$.
$\langle 7 \rangle 3$. Q.E.D.
PROOF: $\langle 7 \rangle 2$.

⟨6⟩2. CASE: $e = (!, m)$
It follows from the rules that the state of the communication medium is irrelevant.
⟨6⟩3. CASE: $e = (?, m)$
In this case it must be shown that $m \in env^!_{\mathcal{M}}.d_2$.
⟨7⟩1. $m \in msg.d_2$
PROOF: Defs. of $ev.\_$ and $msg.\_$, rules (8.10)-(8.14), ⟨6⟩3 and ⟨5⟩2.
⟨7⟩2. $m \in env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2)$
PROOF: It follows from the recursive use of rules (8.3)-(8.14), and def. of *ready* that this is a necessary condition for ⟨3⟩4 to be satisfied.
⟨7⟩3. $m \in env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2) \iff m \in env^!_{\mathcal{M}}.d_2 \wedge (!, m) \notin ev.d_1$
PROOF: $m \in env^!_{\mathcal{M}}.(d_2 \ \mathsf{seq}\ d_1)$

$$
\begin{array}{lll}
\Leftrightarrow & (!, m) \in env.(d_1 \ \mathsf{seq}\ d_2) & \text{(Def. 1)} \\
\Leftrightarrow & (!, m) \in (env.d_1 \setminus ev.d_2) \cup (env.d_2 \setminus ev.d_1) & \text{(Lemma 2)} \\
\Leftrightarrow & ((!, m) \in env.d_1 \wedge (!, m) \notin ev.d_2) \vee & \\
 & ((!, m) \in env.d_2 \wedge (!, m) \notin ev.d_1) & \\
\Leftrightarrow & ((!, m) \in env.d_2 \wedge (!, m) \notin ev.d_1) \vee & \\
 & ((!, m) \in env.d_1 \wedge (!, m) \in env.d_2) & (\langle 7 \rangle 1) \\
\Leftrightarrow & (!, m) \in env.d_2 \wedge & \\
 & ((!, m) \notin ev.d_1 \vee (!, m) \in env.d_1) & \\
\Leftrightarrow & (!, m) \in env.d_2 \wedge ((!, m) \notin ev.d_1 \vee & \\
 & ((!, m) \notin ev.d_1 \wedge m \in msg.d_1)) & \text{(Def. 1)} \\
\Leftrightarrow & (!, m) \in env.d_2 \wedge (!, m) \notin ev.d_1 & \\
\Leftrightarrow & m \in env^!_{\mathcal{M}}.d_2 \ \wedge (!, m) \notin ev.d_1 & \text{(Def. 1)}
\end{array}
$$

⟨7⟩4. Q.E.D.
PROOF: ⟨7⟩2 and ⟨7⟩3.
⟨6⟩4. Q.E.D.
PROOF: ⟨5⟩2, ⟨6⟩1, ⟨6⟩2 and ⟨6⟩3.
⟨5⟩4. Q.E.D.
PROOF: ⟨5⟩3 and rule (8.3)
⟨4⟩2. $[env^!_{\mathcal{M}}.d_1, d_1] \stackrel{e}{\nrightarrow}$
⟨5⟩1. $e \notin ev.d_1$
PROOF: ⟨3⟩4 and application of rules imply $e \in ev.d_1$, and by assumption, $ev.d_1 \cap ev.d_2 = \emptyset$.
⟨5⟩2. $e \notin ev.d_1 \setminus ev.d'_1$
PROOF: ⟨5⟩1.
⟨5⟩3. $\neg([env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{e} [\beta'_1, d'_1])$
PROOF: ⟨5⟩2 and lemma 4.
⟨5⟩4. Q.E.D.
PROOF: ⟨5⟩3
⟨4⟩3. Q.E.D.
PROOF: ⟨4⟩1 and ⟨4⟩2
⟨3⟩5. Q.E.D.
PROOF: ⟨3⟩2, ⟨3⟩3 and ⟨3⟩4.
⟨2⟩3. Q.E.D.
PROOF: We now have that $[\beta', d'] \xrightarrow{t'} [\beta, \mathsf{skip}]$ where $d' = d'_1 \ \mathsf{seq}\ d_2$ or $d' = d_1 \ \mathsf{seq}\ d'_2$. By substituting $[env^!_{\mathcal{M}}.d', d'] \xrightarrow{t'} [\beta, \mathsf{skip}]$ for the assumption we may

repeat the argument until $t' = \langle\rangle$, and we get $\langle 1\rangle 1$. The justification of the substitution is that if $e = (?, m)$ then $\beta' = env^!_{\mathcal{M}}.d \setminus \{m\}$, but we know that $e \notin ev.d'$. If $e = (!, m)$, then $\beta' = env^!_{\mathcal{M}}.d \cup \{m\}$, and $env^!_{\mathcal{M}}.d' = env^!_{\mathcal{M}}.d \cup \{m\} \Leftrightarrow (?, m) \in ev.d'$ and $env^!_{\mathcal{M}}.d' = env^!_{\mathcal{M}}.d \Leftrightarrow (?, m) \notin ev.d$.

$\langle 1\rangle 2$. $\forall l \in \mathcal{L} : e.l \circledS t = e.l \circledS t_1 \frown e.l \circledS t_2$

PROOF: by contradiction

ASSUME: 1. The traces t_1 and t_2 from $\langle 1\rangle 1$
2. l arbitrary chosen lifeline in $\mathcal{L}$
3. $e.l \circledS t \neq e.l \circledS t_1 \frown e.l \circledS t_2$

$\langle 2\rangle 1$. $\#t = \#t_1 + \#t_2$

PROOF: This follows from $ev.t = ev.d = ev.d_1 \cup ev.d_2 = ev.t_1 \cup ev.t_2$ (lemma 5) and the assumption that there is no repetition of events in d.

$\langle 2\rangle 2$. There exist $e_1, e_2 \in e.l$ such that $\{e_1, e_2\} \circledS t \neq \{e_1, e_2\} \circledS t_1 \frown \{e_1, e_2\} \circledS t_2$

PROOF: By assumption 3, $\langle 2\rangle 1$ and the assumption that there is no repetition of events, such events must exist.

$\langle 2\rangle 3$. CASE: $e_1, e_2 \notin ev.d_1 \cup ev.d_2$

$\langle 3\rangle 1$. $\{e_1, e_2\} \circledS t = \langle\rangle$

PROOF: Lemma 5

$\langle 3\rangle 2$. $\{e_1, e_2\} \circledS t_1 \frown \{e_1, e_2\} \circledS t_2 = \langle\rangle \frown \langle\rangle = \langle\rangle$

PROOF: Lemma 5

$\langle 2\rangle 4$. CASE: $e_1 \notin ev.d_1 \cup ev.d_2 \wedge e_2 \in ev.d_1$

$\langle 3\rangle 1$. $\{e_1, e_2\} \circledS t = \langle e_2\rangle$

PROOF: Lemma 5

$\langle 3\rangle 2$. $\{e_1, e_2\} \circledS t_1 \frown \{e_1, e_2\} \circledS t_2 = \langle e_2\rangle \frown \langle\rangle = \langle e_2\rangle$

PROOF: Lemma 5

$\langle 2\rangle 5$. CASE: $e_1 \notin ev.d_1 \cup ev.d_2 \wedge e_2 \in ev.d_2$

$\langle 3\rangle 1$. $\{e_1, e_2\} \circledS t = \langle e_2\rangle$

PROOF: Lemma 5

$\langle 3\rangle 2$. $\{e_1, e_2\} \circledS t_1 \frown \{e_1, e_2\} \circledS t_2 = \langle\rangle \frown \langle e_2\rangle = \langle e_2\rangle$

PROOF: Lemma 5

$\langle 2\rangle 6$. CASE: $e_1 \in ev.d_1 \wedge e_2 \notin ev.d_1 \cup ev.d_2$

$\langle 3\rangle 1$. $\{e_1, e_2\} \circledS t = \langle e_1\rangle$

PROOF: Lemma 5

$\langle 3\rangle 2$. $\{e_1, e_2\} \circledS t_1 \frown \{e_1, e_2\} \circledS t_2 = \langle e_1\rangle \frown \langle\rangle = \langle e_1\rangle$

PROOF: Lemma 5

$\langle 2\rangle 7$. CASE: $e_1 \in ev.d_2 \wedge e_2 \notin ev.d_1 \cup ev.d_2$

$\langle 3\rangle 1$. $\{e_1, e_2\} \circledS t = \langle e_1\rangle$

PROOF: Lemma 5

$\langle 3\rangle 2$. $\{e_1, e_2\} \circledS t_1 \frown \{e_1, e_2\} \circledS t_2 = \langle\rangle \frown \langle e_1\rangle = \langle e_1\rangle$

PROOF: Lemma 5

$\langle 2\rangle 8$. CASE: $e_1, e_2 \in ev.d_1$

$\langle 3\rangle 1$. $e_1, e_2 \notin ev.d_2$

PROOF: Assumption $ev.d_1 \cap ev.d_2 = \emptyset$.

$\langle 3\rangle 2$. $t_1 = ev.d_1 \circledS t \wedge t_2 = ev.d_2 \circledS t$

PROOF: Lemma 4 and assumption $ev.d_1 \cap ev.d_2 = \emptyset$

$\langle 3\rangle 3$. $\{e_1, e_2\} \circledS t_1 \frown \{e_1, e_2\} \circledS t_2$
$= \{e_1, e_2\} \circledS (ev.d_1 \circledS t) \frown \{e_1, e_2\} \circledS (ev.d_2 \circledS t)$

$= (\{e_1, e_2\} \cap ev.d_1)\circledS t^\frown(\{e_1, e_2\} \cap ev.d_2)\circledS t$
$= \{e_1, e_2\}\circledS t^\frown\emptyset\circledS t$
$= \{e_1, e_2\}\circledS t^\frown\langle\rangle$
$= \{e_1, e_2\}\circledS t$

PROOF: $\langle 3\rangle 2$, $\{e_1, e_2\} \subseteq ev.d_1$ ($\langle 2\rangle 8$) and $\{e_1, e_2\} \cap ev.d_2 = \emptyset$ ($\langle 3\rangle 1$).

$\langle 2\rangle 9$. CASE: $e_1, e_2 \in ev.d_2$

$\langle 3\rangle 1$. $e_1, e_2 \notin ev.d_1$

PROOF: Assumption $ev.d_1 \cap ev.d_2 = \emptyset$.

$\langle 3\rangle 2$. $t_1 = ev.d_1\circledS t \wedge t_2 = ev.d_2\circledS t$

PROOF: Lemma 4 and assumption $ev.d_1 \cap ev.d_2 = \emptyset$

$\langle 3\rangle 3$. $\{e_1, e_2\}\circledS t_1{}^\frown\{e_1, e_2\}\circledS t_2$
$= \{e_1, e_2\}\circledS(ev.d_1\circledS t)^\frown\{e_1, e_2\}\circledS(ev.d_2\circledS t)$
$= \{e_1, e_2\} \cap ev.d_1\circledS t^\frown\{e_1, e_2\} \cap ev.d_2\circledS t$
$= \emptyset\circledS t^\frown\{e_1, e_2\}\circledS t$
$= \langle\rangle^\frown\{e_1, e_2\}\circledS t$
$= \{e_1, e_2\}\circledS t$

PROOF: $\langle 3\rangle 2$, $\{e_1, e_2\} \subseteq ev.d_2$ ($\langle 2\rangle 9$) and $\{e_1, e_2\} \cap ev.d_1 = \emptyset$ ($\langle 3\rangle 1$).

$\langle 2\rangle 10$. CASE: $e_1 \in ev.d_1 \wedge e_2 \in ev.d_2$

$\langle 3\rangle 1$. $e_1 \notin ev.d_2 \wedge e_2 \notin ev.d_1$

PROOF: Assumption $ev.d_1 \cap ev.d_2 = \emptyset$.

$\langle 3\rangle 2$. $t_1 = ev.d_1\circledS t \wedge t_2 = ev.d_2\circledS t$

PROOF: Lemma 4 and assumption $ev.d_1 \cap ev.d_2 = \emptyset$

$\langle 3\rangle 3$. $\{e_1, e_2\}\circledS t_1 = \{e_1, e_2\}\circledS(ev.d_1\circledS t) = \{e_1, e_2\} \cap ev.d_1\circledS t = \{e_1\}\circledS t = \langle e_1\rangle$

PROOF: Lemma 4, $\langle 2\rangle 10$, $\langle 3\rangle 1$ and $\langle 3\rangle 2$.

$\langle 3\rangle 4$. $\{e_1, e_2\}\circledS t_2 = \{e_1, e_2\}\circledS(ev.d_2\circledS t) = \{e_1, e_2\} \cap ev.d_2\circledS t = \{e_2\}\circledS t = \langle e_2\rangle$

PROOF: Lemma 4, $\langle 2\rangle 10$, $\langle 3\rangle 1$ and $\langle 3\rangle 2$.

$\langle 3\rangle 5$. $\{e_1, e_2\}\circledS t = \langle e_2, e_1\rangle$

PROOF: By assumption 3, $\langle 3\rangle 3$ and $\langle 3\rangle 4$ we have that $\{e_1, e_2\}\circledS t \neq \langle e_1, e_2\rangle$. By $\langle 2\rangle 10$ and lemma 5 we have that $\{e_,e_2\} \subseteq ev.t$.

$\langle 3\rangle 6$. There exist traces s_1, s_2, s_3 such that $t = s_1{}^\frown\langle e_2\rangle^\frown s_2{}^\frown\langle e_1\rangle^\frown s_3$

PROOF: $\langle 3\rangle 5$.

$\langle 3\rangle 7$. There exist $\beta, \beta' \in \mathcal{B}, d_1', d_2', d_2'' \in \mathcal{D}$ such that
$[env^!_{\mathcal{M}}.(d_1 \mathsf{\ seq\ } d_2), d_1 \mathsf{\ seq\ } d_2] \xrightarrow{s_1} [\beta, d_1' \mathsf{\ seq\ } d_2'] \xrightarrow{e_2} [\beta', d_1' \mathsf{\ seq\ } d_2'']$

PROOF: $e_2 \in ev.d_2$ and $\langle 3\rangle 6$.

$\langle 3\rangle 8$. $l.e_1 = l.e_2 = l$

PROOF: $\langle 2\rangle 2$, defs. of $e.\_$ and $l.\_$.

$\langle 3\rangle 9$. $l \notin ll.d_1'$

$\langle 4\rangle 1$. $l \in ll.(d_1' \mathsf{\ seq\ } d_2') \setminus ll.d_1'$

PROOF: By $\langle 3\rangle 7$ and application of the rules this is a necessary condition for $\langle 3\rangle 8$.

$\langle 4\rangle 2$. $l \in ll.(d_1' \mathsf{\ seq\ } d_2') \wedge l \notin ll.d_1'$

PROOF: $\langle 4\rangle 1$.

$\langle 4\rangle 3$. Q.E.D.

PROOF: $\langle 4\rangle 2$

$\langle 3\rangle 10$. $l \in ll.d_1'$

$\langle 4\rangle 1$. $e_1 \in ev.d_1'$

PROOF: $\langle 2\rangle 10$, $e_1 \notin ev.s_1$ and lemma 4.

$\langle 4\rangle 2$. Q.E.D.

PROOF: $\langle 3\rangle 8$ and def. of $ll.\_$.

$\langle 2\rangle 11$. CASE: $e_1 \in ev.d_2 \wedge e_2 \in ev.d_1$

$\langle 3\rangle 1$. $e_1 \notin ev.d_1 \wedge e_2 \notin ev.d_2$

PROOF: Assumption $ev.d_1 \cap ev.d_2 = \emptyset$.

$\langle 3\rangle 2$. $t_1 = ev.d_1 \circledS t \wedge t_2 = ev.d_2 \circledS t$

PROOF: Lemma 4 and assumption $ev.d_1 \cap ev.d_2 = \emptyset$

$\langle 3\rangle 3$. $\{e_1, e_2\} \circledS t_1 = \{e_1, e_2\} \circledS (ev.d_1 \circledS t) = \{e_1, e_2\} \cap ev.d_1 \circledS t = \{e_2\} \circledS t = \langle e_2 \rangle$

PROOF: Lemma 4, $\langle 2\rangle 11$, $\langle 3\rangle 1$ and $\langle 3\rangle 2$.

$\langle 3\rangle 4$. $\{e_1, e_2\} \circledS t_2 = \{e_1, e_2\} \circledS (ev.d_2 \circledS t) = \{e_1, e_2\} \cap ev.d_2 \circledS t = \{e_1\} \circledS t = \langle e_1 \rangle$

PROOF: Lemma 4, $\langle 2\rangle 11$, $\langle 3\rangle 1$ and $\langle 3\rangle 2$.

$\langle 3\rangle 5$. $\{e_1, e_2\} \circledS t = \langle e_1, e_2 \rangle$

PROOF: By assumption 3, $\langle 3\rangle 3$ and $\langle 3\rangle 4$ we have that $\{e_1, e_2\} \circledS t \neq \langle e_2, e_1 \rangle$. By $\langle 2\rangle 10$ and lemma 5 we have that $\{e_1, e_2\} \subseteq ev.t$.

$\langle 3\rangle 6$. There exist traces s_1, s_2, s_3 such that $t = s_1 \frown \langle e_1 \rangle \frown s_2 \frown \langle e_2 \rangle \frown s_3$

PROOF: $\langle 3\rangle 5$.

$\langle 3\rangle 7$. There exist $\beta, \beta' \in \mathcal{B}, d_1', d_2', d_2'' \in \mathcal{D}$ such that
$[env^!_{\mathcal{M}}.(d_1, d_2), d_1 \text{ seq } d_2] \xrightarrow{s_1} [\beta, d_1' \text{ seq } d_2'] \xrightarrow{e_1} [\beta', d_1' \text{ seq } d_2'']$

PROOF: $e_1 \in ev.d_2$ and $\langle 3\rangle 6$.

$\langle 3\rangle 8$. $l.e_1 = l.e_2 = l$

PROOF: $\langle 2\rangle 2$, defs. of $e.\_$ and $l.\_$.

$\langle 3\rangle 9$. $l \notin ll.d_1'$

$\langle 4\rangle 1$. $l \in ll.(d_1' \text{ seq } d_2') \setminus ll.d_1'$

PROOF: By $\langle 3\rangle 7$ and application of the rules this is a necessary condition for $\langle 3\rangle 8$.

$\langle 4\rangle 2$. $l \in ll.(d_1' \text{ seq } d_2') \wedge l \notin ll.d_1'$

PROOF: $\langle 4\rangle 1$.

$\langle 4\rangle 3$. Q.E.D.

PROOF: $\langle 4\rangle 2$

$\langle 3\rangle 10$. $l \in ll.d_1'$

$\langle 4\rangle 1$. $e_2 \in ev.d_1'$

PROOF: $\langle 2\rangle 11$, $e_2 \notin ev.s_1$ and lemma 4.

$\langle 4\rangle 2$. Q.E.D.

PROOF: $\langle 3\rangle 8$ and def. of $ll.\_$.

$\langle 2\rangle 12$. Q.E.D.

PROOF: All possible cases yield a contradiction, so we must have that $e.l \circledS t = e.l \circledS t_1 \frown e.l \circledS t_2$. Since l was arbitrary chosen, this must be true for all lifelines in $\mathcal{L}$.

$\langle 1\rangle 3$. Q.E.D.

PROOF: $\langle 1\rangle 1$ and $\langle 1\rangle 2$.

□

Definition 5 *We define a function* $\Omega \in \mathbb{P}(\mathcal{E}) \times \mathbb{P}(\mathcal{E}) \times \mathcal{E}^* \to \{1, 2\}^*$ *in the following*

way

$$\begin{aligned}\Omega(E_1, E_2, \langle\rangle) &\stackrel{\mathsf{def}}{=} \langle\rangle \\ \Omega(E_1, E_2, \langle e\rangle^\frown t) &\stackrel{\mathsf{def}}{=} \begin{cases}\langle 1\rangle^\frown\Omega(E_1, E_2, t) & \textbf{if } e \in E_1 \\ \langle 2\rangle^\frown\Omega(E_1, E_2, t) & \textbf{if } e \in E_2 \\ \Omega(E_1, E_2, t) & \textbf{otherwise}\end{cases}\end{aligned}$$

where we assume $E_1 \cap E_2 = \emptyset$.

Definition 6 *We generalize the concatenation operator* $\_^\frown\_$ *to range over pairs of traces:*

$$(t_1, t_2)^\frown(s_1, s_2) \stackrel{\mathsf{def}}{=} (t_1{}^\frown s_1, t_2{}^\frown s_2)$$

Lemma 8 $\pi_i(t_1{}^\frown s_1, t_2{}^\frown s_2) = \pi_i(t_1, t_2)^\frown\pi_i(s_1, s_2),\ i \in \{1, 2\}$

Proof of lemma 8
⟨1⟩1. $\pi_i(t_1{}^\frown s_1, t_2{}^\frown s_2) = t_i{}^\frown s_i$
PROOF: Def. of π_i.
⟨1⟩2. $\pi_i(t_1, t_2)^\frown\pi_i(s_1, s_2) = t_i{}^\frown s_i$
PROOF: Def. of π_i.
⟨1⟩3. Q.E.D.
PROOF: ⟨1⟩1 and ⟨1⟩2.

□

Definition 7 *We formally define the pair filtering operator* $\_ⓣ\_$ *(for finite sequences):*

$$\begin{aligned}P ⓣ (t_1, t_2) &\stackrel{\mathsf{def}}{=} P ⓣ (t_1|_{min(\#t_1, \#t_2)}, t_2|_{min(\#t_1, \#t_2)}) \\ P ⓣ (\langle\rangle, \langle\rangle) &\stackrel{\mathsf{def}}{=} (\langle\rangle, \langle\rangle) \\ (v_1, v_2) \in P \Rightarrow P ⓣ (\langle v_1\rangle^\frown t_1, \langle v_2\rangle^\frown t_2) &\stackrel{\mathsf{def}}{=} (\langle v_1\rangle, \langle v_2\rangle)^\frown P ⓣ (t_1, t_2) \\ (v_1, v_2) \notin P \Rightarrow P ⓣ (\langle v_1\rangle^\frown t_1, \langle v_2\rangle^\frown t_2) &\stackrel{\mathsf{def}}{=} P ⓣ (t_1, t_2)\end{aligned}$$

where P *is a set of pairs* ($P \subseteq A \times B$, $t_1 \in A^*$, $t_2 \in B^*$).

Lemma 9 *Given simple diagrams* $d_1, d_2 \in \mathcal{D}$. *For all traces* t, *if there exists* $\beta \in \mathcal{B}$ *such that*

$$[env^!_{\mathcal{M}}.(d_1 \ \mathsf{par}\ d_2), d_1 \ \mathsf{par}\ d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$$

then there exist traces t_1, t_2 *and* $\beta_1, \beta_2 \in \mathcal{B}$ *such that*

$$\begin{aligned}&[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}] \wedge \pi_2((\{1\} \times \mathcal{E}) ⓣ (\Omega(ev.d_1, ev.d_2, t), t)) = t_1 \ \wedge \\ &[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}] \wedge \pi_2((\{2\} \times \mathcal{E}) ⓣ (\Omega(ev.d_1, ev.d_2, t), t)) = t_2\end{aligned}$$

Proof of lemma 9
ASSUME: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \ \mathsf{par}\ d_2), d_1 \ \mathsf{par}\ d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$

PROVE: There exist traces t_1, t_2, and $\beta_1, \beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}] \wedge$
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}] \wedge$
$\pi_2((\{1\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, t), t)) = t_1 \wedge$
$\pi_2((\{2\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, t), t)) = t_2 \wedge$

$\langle 1 \rangle 1$. LET: $t = \langle e \rangle^\frown t'$

$\langle 1 \rangle 2$. There exist $\beta'_1, \beta'_2 \in \mathcal{B}, d'_1, d'_2 \in \mathcal{D}$ such that
$([env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{e} [\beta'_1, d'_1] \wedge$
$\pi_2((\{1\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, \langle e \rangle^\frown t'), \langle e \rangle^\frown t')) =$
$\langle e \rangle^\frown \pi_2((\{1\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, t'), t'))) \veebar$
$([env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{e} [\beta'_2, d'_2] \wedge$
$\pi_2((\{2\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, \langle e \rangle^\frown t'), \langle e \rangle^\frown t')) =$
$\langle e \rangle^\frown \pi_2((\{2\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, t'), t')))$

$\langle 2 \rangle 1$. There exist $\beta' \in \mathcal{B}, d' \in \mathcal{D}$ such that
$[env^!_{\mathcal{M}}.(d_1 \mathsf{\ par\ } d_2), d_1 \mathsf{\ par\ } d_2] \xrightarrow{e} [\beta', d'] \xrightarrow{t'} [\beta, \mathsf{skip}]$
PROOF: Assumption and $\langle 1 \rangle 1$

$\langle 2 \rangle 2$. $[env^!_{\mathcal{M}}.(d_1 \mathsf{\ par\ } d_2), d_1 \mathsf{\ par\ } d_2] \xrightarrow{e} [\beta', d'_1 \mathsf{\ par\ } d_2] \veebar$
$[env^!_{\mathcal{M}}.(d_1 \mathsf{\ par\ } d_2), d_1 \mathsf{\ par\ } d_2] \xrightarrow{e} [\beta', d_1 \mathsf{\ par\ } d'_2]$
PROOF: $\langle 2 \rangle 1$, rules (8.3), (8.13), (8.14), assumption that $ev.d_1 \cap ev.d_2 = \emptyset$, and lemma 4.

$\langle 2 \rangle 3$. CASE: $[env^!_{\mathcal{M}}.(d_1 \mathsf{\ par\ } d_2), d_1 \mathsf{\ par\ } d_2] \xrightarrow{e} [\beta', d'_1 \mathsf{\ par\ } d_2]$

$\langle 3 \rangle 1$. There exists $\beta'_1 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{e} [\beta'_1, d'_1]$
PROOF: Identical to proof of lemma 7.

$\langle 3 \rangle 2$. $[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{e}\!\!\!\!\!\!\rightarrow$
PROOF: Identical to proof of lemma 7.

$\langle 3 \rangle 3$. $\pi_2((\{1\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, \langle e \rangle^\frown t'), \langle e \rangle^\frown t'))$
$= \langle e \rangle^\frown \pi_2((\{1\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, t'), t'))$

$\langle 4 \rangle 1$. $e \in ev.d_1$
PROOF: $\langle 3 \rangle 1$ and lemma 4.

$\langle 4 \rangle 2$. $\Omega(ev.d_1, ev.d_2, \langle e \rangle^\frown t') = \langle 1 \rangle^\frown \Omega(ev.d_1, ev.d_2, t')$
PROOF: $\langle 4 \rangle 1$, def. 5 of Ω and the assumption that $ev.d_1 \cap ev.d_2 = \emptyset$.

$\langle 4 \rangle 3$. $\pi_2((\{1\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, \langle e \rangle^\frown t'), \langle e \rangle^\frown t'))$
$= \pi_2((\{1\} \times \mathcal{E})$
$\circledcirc (\langle 1 \rangle^\frown \Omega(ev.d_1, ev.d_2, t'), \langle e \rangle^\frown t'))$ ($\langle 4 \rangle 2$)
$= \pi_2((\langle 1 \rangle, \langle e \rangle)$
$\frown (\{1\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, t'), t')))$ (Def. 7 of $\circledcirc$)
$= \pi_2((\langle 1 \rangle, \langle e \rangle))$
$\frown \pi_2((\{1\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, t'), t'))$ (Lemma 8)
$= \langle e \rangle^\frown \pi_2((\{1\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, t'), t'))$ (Def. of π_2)

$\langle 4 \rangle 4$. Q.E.D.
PROOF: $\langle 4 \rangle 2$ and $\langle 4 \rangle 3$.

$\langle 3 \rangle 4$. Q.E.D.
PROOF: $\langle 3 \rangle 1$, $\langle 3 \rangle 2$ and $\langle 3 \rangle 3$.

$\langle 2 \rangle 4$. CASE: $[env^!_{\mathcal{M}}.(d_1 \mathsf{\ par\ } d_2), d_1 \mathsf{\ par\ } d_2] \xrightarrow{e} [\beta'', d_1 \mathsf{\ par\ } d'_2]$

PROOF: Identical to proof of ⟨2⟩3.

⟨2⟩5. Q.E.D.

PROOF: ⟨2⟩2, ⟨2⟩3 and ⟨2⟩4.

⟨1⟩3. Q.E.D.

PROOF: We now have that $[\beta', d'] \xrightarrow{t'} [\beta, \mathsf{skip}]$ where $d' = d'_1$ par d_2 or $d' = d_1$ par d'_2. By substituting $[env^!_{\mathcal{M}}.d', d'] \xrightarrow{t'} [\beta, \mathsf{skip}]$ for the assumption we may repeat the argument until $t' = \langle\rangle$, and we get what we wanted to prove. The justification of the substitution is that if $e = (?, m)$ then $\beta' = env^!_{\mathcal{M}}.d \setminus \{m\}$, but we know that $e \notin ev.d'$. If $e = (!, m)$, then $\beta' = env^!_{\mathcal{M}}.d \cup \{m\}$, and $env^!_{\mathcal{M}}.d' = env^!_{\mathcal{M}}.d \cup \{m\} \Leftrightarrow (?, m) \in ev.d'$ and $env^!_{\mathcal{M}}.d' = env^!_{\mathcal{M}}.d \Leftrightarrow (?, m) \notin ev.d$.

□

Lemma 10 *Given simple diagrams $d_1, d_2 \in \mathcal{D}$. For all traces t, if there exists $\beta \in \mathcal{B}$ such that*

$$[env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2), d_1 \text{ strict } d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$$

then there exist traces t_1, t_2 and $\beta_1, \beta_2 \in \mathcal{B}$ such that

$$\begin{aligned} &[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}] \wedge \\ &[end^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}] \wedge \\ &t = t_1 \frown t_2 \end{aligned}$$

Proof of lemma 10

ASSUME: There exists β such that
$[env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2), d_1 \text{ strict } d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$

PROVE: There exists traces t_1, t_2, and $\beta_1, \beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}] \wedge$
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}] \wedge$
$t = t_1 \frown t_2$

⟨1⟩1. There exist traces t', t'', and $\beta' \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2), d_1 \text{ strict } d_2] \xrightarrow{t'} [\beta', \mathsf{skip} \text{ strict } d_2] \wedge$
$[\beta', d_2] \xrightarrow{t''} [\beta, \mathsf{skip}]$

PROOF: Assumption and rules (8.3) and (8.15).

⟨1⟩2. $[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}]$

⟨2⟩1. $[env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2), d_1] \xrightarrow{t'} [\beta', \mathsf{skip}]$

PROOF: ⟨1⟩1.

⟨2⟩2. There exists $\beta'' \in \mathcal{B}$ such that $[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t'} [\beta'', \mathsf{skip}]$

PROOF: by contradiction

ASSUME: ⟨2⟩2 is not the case. By ⟨2⟩1 and rules (8.3), (8.10), (8.11), (8.12), (8.13), (8.14) and (8.15), two cases must be considered.

⟨3⟩1. CASE: There exists $e \in ev.t'$ such that $l.e \in ll.(d_1 \text{ strict } d_2)$ but $l.e \notin ll.d_1$.

PROOF: By ⟨2⟩1 and lemma 5, $e \in ev.d_1$ so by def. of $ev.\_$ and $ll.\_$ we must have that $l.e \in ll.d_1$. Thus this is impossible.

⟨3⟩2. CASE: There exists message m such that $(?, m) \in ev.t'$ and $m \in env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2)$, but $m \notin env^!_{\mathcal{M}}.d_1$.

⟨4⟩1. $m \in msg.d_1$
PROOF: $(?, m) \in ev.t'$ (⟨3⟩2)
$\Rightarrow$ $(?, m) \in ev.d_1$ (Lemma 5)
$\Rightarrow$ $m \in msg.d_1$ (Defs. of msg and ev)
⟨4⟩2. $(!, m) \in ev.d_1$
PROOF: $m \notin env^!_{\mathcal{M}}.d_1$ (⟨3⟩2)
$\Rightarrow$ $(!, m) \notin env.d_1$ (Def. 1)
$\Rightarrow$ $m \notin msg.d_1 \vee (!, m) \in ev.d_1$ (Def. 1)
$\Rightarrow$ $(!, m) \in ev.d_1$ (⟨4⟩1)
⟨4⟩3. $(!, m) \notin ev.d_1$
PROOF: $m \in env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2)$ (⟨3⟩2)
$\Rightarrow$ $(!, m) \in env.(d_1 \text{ strict } d_2)$ (Def. 1)
$\Rightarrow$ $m \in msg.(d_1 \text{ strict } d_2) \wedge (!, m) \notin ev.(d_1 \text{ strict } d_2)$ (Def. 1)
$\Rightarrow$ $m \in msg.d_1 \cup msg.d_2 \wedge (!, m) \notin ev.d_1 \cup ev.d_2$ (Def. of $ev.\_$)
$\Rightarrow$ $(!, m) \notin ev.d_1 \wedge (!, m) \notin ev.d_2$ (Set theory)
$\Rightarrow$ $(!, m) \notin ev.d_1$
⟨4⟩4. Q.E.D.
PROOF: ⟨4⟩2 and ⟨4⟩3 yield a contradiction, so ⟨3⟩2 must be impossible.
⟨3⟩3. Q.E.D.
PROOF: Since ⟨3⟩1 and ⟨3⟩2 are impossible, we must have ⟨2⟩2
⟨2⟩3. Q.E.D.
PROOF: ⟨2⟩2 by letting $t_1 = t'$ and $\beta_1 = \beta''$.
⟨1⟩3. $[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}]$
⟨2⟩1. $[\beta', d_2] \xrightarrow{t''} [\beta, \mathsf{skip}]$
PROOF: ⟨1⟩1.
⟨2⟩2. There exist $\beta''' \in \mathcal{B}$ such that $[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t''} [\beta''', \mathsf{skip}]$
PROOF: by contradiction
ASSUME: ⟨2⟩2 is not the case
⟨3⟩1. There exists message m such that $(?, m) \in ev.t''$ and $m \in \beta'$, but $m \notin env^!_{\mathcal{M}}.d_2$
PROOF: ⟨2⟩1, assumption ⟨2⟩2 is not the case, and rules (8.3), (8.10), (8.11), (8.12), (8.13), (8.14) and (8.15).
⟨3⟩2. $m \in msg.d_2$
PROOF: $(?, m) \in ev.t''$ (⟨3⟩1)
$\Rightarrow$ $(?, m) \in ev.d_2$ (Lemma 5)
$\Rightarrow$ $m \in msg.d_2$ (Defs. of msg and ev)
⟨3⟩3. $(!, m) \in ev.d_2$
PROOF: $m \notin env^!_{\mathcal{M}}.d_2$ (⟨3⟩1)
$\Rightarrow$ $(!, m) \notin env.d_2$ (Def. 1)
$\Rightarrow$ $m \notin msg.d_2 \vee (!, m) \in ev.d_2$ (Def. 1)
$\Rightarrow$ $(!, m) \in ev.d_2$ (⟨3⟩2)
⟨3⟩4. $(!, m) \notin ev.d_2$

PROOF: $m \in \beta'$ ($\langle 3 \rangle 1$)
$\Rightarrow$ $m \in env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2) \vee (!, m) \in ev.t'$ ($\langle 1 \rangle 1$ and rules)
$\Rightarrow$ $m \in env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2) \vee (!, m) \in ev.d_1$ (Lemma 5)
$\Rightarrow$ $m \in env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2)$ ($ev.d_1 \cap ev.d_2 = \emptyset$)
$\Rightarrow$ $(!, m) \in env.(d_1 \text{ strict } d_2)$ (Def. 1)
$\Rightarrow$ $m \in msg.(d_1 \text{ strict } d_2) \wedge$
$(!, m) \notin ev.(d_1 \text{ strict } d_2)$
$\Rightarrow$ $(m \in msg.d_1 \vee m \in msg.d_2) \wedge$
$(!, m) \notin ev.d_1 \wedge (!, m) \notin ev.d_2$ (Def. 1)
$\Rightarrow$ $(!, m) \notin ev.d_2$

$\langle 3 \rangle 5$. Q.E.D.
PROOF: $\langle 3 \rangle 3$ and $\langle 3 \rangle 4$ yield a contradiction, so we must have $\langle 2 \rangle 2$.

$\langle 2 \rangle 3$. Q.E.D.
PROOF: $\langle 2 \rangle 2$ by letting $t_2 = t''$ and $\beta_2 = \beta'''$.

$\langle 1 \rangle 4$. $t = t_1 \frown t_2$
PROOF: By assumption and $\langle 1 \rangle 1$, $t = t' \frown t''$. By proofs of $\langle 1 \rangle 2$ and $\langle 1 \rangle 3$, $t' = t_1$ and $t'' = t_2$.

$\langle 1 \rangle 5$. Q.E.D.
PROOF: $\langle 1 \rangle 2$, $\langle 1 \rangle 3$ and $\langle 1 \rangle 4$.

□

Theorem 1 (Soundness) *Given a simple diagram $d \in \mathcal{D}$. Then, for all traces t, if there exists $\beta \in \mathcal{B}$ such that*

$$[env^!_{\mathcal{M}}.d, d] \xrightarrow{t} [\beta, \mathsf{skip}]$$

then

$$t \in [\![d]\!]$$

Proof of theorem 1
ASSUME: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d, d] \xrightarrow{t} [\beta, \mathsf{skip}]$
PROVE: $t \in [\![d]\!]$
PROOF: by induction on the structure of d
$\langle 1 \rangle 1$. CASE: $d = \mathsf{skip}$ (induction start: empty diagram)
ASSUME: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.\mathsf{skip}, \mathsf{skip}] \xrightarrow{t} [\beta, \mathsf{skip}]$
PROVE: $t \in [\![\mathsf{skip}]\!]$
$\langle 2 \rangle 1$. $t = \langle\rangle$
PROOF: Rule (8.7)
$\langle 2 \rangle 2$. $\langle\rangle \in [\![\mathsf{skip}]\!]$
$\langle 3 \rangle 1$. $[\![\mathsf{skip}]\!] = \{\langle\rangle\}$
PROOF: Definition (5.3).
$\langle 3 \rangle 2$. Q.E.D.
PROOF: $\langle\rangle \in \{\langle\rangle\}$
$\langle 2 \rangle 3$. Q.E.D.
PROOF: $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$.

⟨1⟩2. CASE: $d = (!, m)$ (induction start: single transmit event)
ASSUME: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(!, m), (!, m)] \xrightarrow{t} [\beta, \mathsf{skip}]$
PROVE: $t \in [\![(!, m)]\!]$
⟨2⟩1. $t = \langle (!, m) \rangle$
⟨3⟩1. $[env^!_{\mathcal{M}}.(!, m), (!, m)] \xrightarrow{(!,m)} [update(env^!_{\mathcal{M}}.(!, m), m.(!, m)), \mathsf{skip}]$
⟨4⟩1. $\Pi(ll.(!, m), env^!_{\mathcal{M}}.(!, m), (!, m))$
$\xrightarrow{(!,m)} \Pi(ll.(!, m), env^!_{\mathcal{M}}.(!, m), \mathsf{skip})$
⟨5⟩1. $ll.(!, m) = \{l.(!, m)\}$
PROOF: Def. of ll._
⟨5⟩2. $l.(!, m) \in ll.(!, m)$
PROOF: ⟨5⟩1
⟨5⟩3. $k.(!, m) =!$
PROOF: Def. of k._
⟨5⟩4. Q.E.D.
PROOF: ⟨5⟩2, ⟨5⟩3 and rule (8.10)
⟨4⟩2. Q.E.D.
PROOF: ⟨4⟩1 and rule (8.3)
⟨3⟩2. Q.E.D.
PROOF: ⟨3⟩1 and lemma 5
⟨2⟩2. $\langle (!, m) \rangle \in [\![(!, m)]\!]$
⟨3⟩1. $[\![(!, m)]\!] = \{\langle (!, m) \rangle\}$
PROOF: Def. (5.4).
⟨3⟩2. Q.E.D.
PROOF: $\langle (!, m) \rangle \in \{\langle (!, m) \rangle\}$.
⟨2⟩3. Q.E.D.
PROOF: ⟨2⟩1 and ⟨2⟩2.
⟨1⟩3. CASE: $d = (?, m)$ (induction start: single receive event)
ASSUME: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(?, m), (?, m)] \xrightarrow{t} [\beta, \mathsf{skip}]$
PROVE: $t \in [\![(?, m)]\!]$
⟨2⟩1. $t = \langle (?, m) \rangle$
⟨3⟩1. $env^!_{\mathcal{M}}.(?, m) = \{m\}$
⟨4⟩1. $env.(?, m) = \{(!, m)\}$
PROOF: Def. 1
⟨4⟩2. Q.E.D.
PROOF: ⟨4⟩1 and def. 1
⟨3⟩2. $[\{m\}, (?, m)] \xrightarrow{(?,m)} [update(\{m\}, m.(?, m)), \mathsf{skip}]$
⟨4⟩1. $\Pi(ll.(?, m), \{m\}, (?, m)) \xrightarrow{(?,m)} \Pi(ll.(?, m), \{m\}, \mathsf{skip})$
⟨5⟩1. $ll.(?, m) = \{l.(?, m)\}$
PROOF: Def. of ll._
⟨5⟩2. $l.(?, m) \in ll.(?, m)$
PROOF: ⟨5⟩1
⟨5⟩3. $ready(\{m\}, m.(?, m))$
PROOF: $ready(\{m\}, m.(?, m)) = m \in \{m\} = true$ (def. (8.8) and def. of m._)

⟨5⟩4. Q.E.D.
PROOF: ⟨5⟩2, ⟨5⟩3 and rule (8.10)
⟨4⟩2. Q.E.D.
PROOF: ⟨4⟩1 and rule (8.3)
⟨3⟩3. Q.E.D.
PROOF: ⟨3⟩1, ⟨3⟩2 and lemma 5
⟨2⟩2. $\langle(?, m)\rangle \in [\![(?, m)]\!]$
⟨3⟩1. $[\![(?, m)]\!] = \{\langle(?, m)\rangle\}$
PROOF: Def. (5.4).
⟨3⟩2. Q.E.D.
PROOF: $\langle(?, m)\rangle \in \{\langle(?, m)\rangle\}$.
⟨2⟩3. Q.E.D.
PROOF: ⟨2⟩1 and ⟨2⟩2.
⟨1⟩4. CASE: $d = d_1$ seq d_2 (induction step)
ASSUME: 1. There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1 \text{ seq } d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$
2. For all traces t_1, if there exists $\beta_1 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}]$,
then $t_1 \in [\![d_1]\!]$ (induction hypothesis)
3. For all traces t_2, if there exists $\beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}]$,
then $t_2 \in [\![d_2]\!]$ (induction hypothesis)
PROVE: $t \in [\![d_1 \text{ seq } d_2]\!]$
⟨2⟩1. $[\![d_1 \text{ seq } d_2]\!] = \{h \in \mathcal{H} | \exists h_1 \in [\![d_1]\!], h_2 \in [\![d_2]\!] : \forall l \in \mathcal{L} : e.l \circledS h = e.l \circledS h_1 \frown e.l \circledS h_2\}$
PROOF: Assumption that d is simple, and defs. (5.5), (5.7) and (5.6).
⟨2⟩2. $t \in \mathcal{H}$
PROOF: Assumption 1 and lemma 6.
⟨2⟩3. $\exists h_1 \in [\![d_1]\!], h_2 \in [\![d_2]\!] : \forall l \in \mathcal{L} : e.l \circledS t = e.l \circledS h_1 \frown e.l \circledS h_2$
⟨3⟩1. There exist traces s_1, s_2, and $\beta_1, \beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{s_1} [\beta_1, \mathsf{skip}] \wedge$
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{s_2} [\beta_2, \mathsf{skip}] \wedge$
$\forall l \in \mathcal{L} : e.l \circledS t = e.l \circledS s_1 \frown e.l \circledS s_2$
PROOF: Assumption 1 and lemma 7.
⟨3⟩2. $s_1 \in [\![d_1]\!], s_2 \in [\![d_2]\!]$
PROOF: ⟨3⟩1and assumptions 2 and 3 (induction hypothesis).
⟨3⟩3. Q.E.D.
PROOF: ⟨3⟩1 and ⟨3⟩2
⟨2⟩4. Q.E.D.
PROOF: ⟨2⟩1, ⟨2⟩2 and ⟨2⟩3.
⟨1⟩5. CASE: $d = d_1$ par d_2 (induction step)
ASSUME: 1. There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \text{ par } d_2), d_1 \text{ par } d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$
2. For all traces t_1, if there exists $\beta_1 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}]$,
then $t_1 \in [\![d_1]\!]$ (induction hypothesis)

3. For all traces t_2, if there exists $\beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}]$,
then $t_2 \in [\![d_2]\!]$ (induction hypothesis)

PROVE: $t \in [\![d_1 \ \mathsf{par}\ d_2]\!]$

$\langle 2 \rangle 1$. $[\![d_1 \ \mathsf{par}\ d_2]\!] = \{h \in \mathcal{H} | \exists p \in \{1,2\}^\infty :$
$\pi_2((\{1\} \times \mathcal{E}) \circledcirc (p,h)) \in [\![d_1]\!] \wedge$
$\pi_2((\{2\} \times \mathcal{E}) \circledcirc (p,h)) \in [\![d_2]\!] \}$

PROOF: Assumption that d is simple, and defs. (5.8), (5.10) and (5.9).

$\langle 2 \rangle 2$. $t \in \mathcal{H}$

PROOF: Assumption 1 and lemma 6.

$\langle 2 \rangle 3$. $\exists p \in \{1,2\}^\infty : \pi_2((\{1\} \times \mathcal{E}) \circledcirc (p,t)) \in [\![d_1]\!] \wedge$
$\pi_2((\{2\} \times \mathcal{E}) \circledcirc (p,t)) \in [\![d_2]\!]$

$\langle 3 \rangle 1$. There exist traces s_1, s_2, and $\beta_1, \beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{s_1} [\beta_1, \mathsf{skip}] \wedge$
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{s_2} [\beta_2, \mathsf{skip}] \wedge$
$\pi_2((\{1\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, t), t)) = s_1 \wedge$
$\pi_2((\{2\} \times \mathcal{E}) \circledcirc (\Omega(ev.d_1, ev.d_2, t), t)) = s_2$

PROOF: Assumption 1 and lemma 9.

$\langle 3 \rangle 2$. $s_1 \in [\![d_1]\!], s_2 \in [\![d_2]\!]$

PROOF: $\langle 3 \rangle 1$ and assumptions 2 and 3 (induction hypothesis).

$\langle 3 \rangle 3$. LET: $p = \Omega(ev.d_1, ev.d_2, t)^\frown \{1\}^\infty$

$\langle 3 \rangle 4$. Q.E.D.

PROOF: $\langle 3 \rangle 1$, $\langle 3 \rangle 2$, $\langle 3 \rangle 3$ and def. of $\_ \circledcirc \_$.

$\langle 2 \rangle 4$. Q.E.D.

PROOF: $\langle 2 \rangle 1$, $\langle 2 \rangle 2$ and $\langle 2 \rangle 3$.

$\langle 1 \rangle 6$. CASE: $d = d_1 \ \mathsf{strict}\ d_2$ (induction step)

ASSUME: 1. There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \ \mathsf{strict}\ d_2), d_1 \ \mathsf{strict}\ d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$

2. For all traces t_1, if there exists $\beta_1 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}]$,
then $t_1 \in [\![d_1]\!]$ (induction hypothesis)

3. For all traces t_2, if there exists $\beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}]$,
then $t_2 \in [\![d_2]\!]$ (induction hypothesis)

PROVE: $t \in [\![d_1 \ \mathsf{strict}\ d_2]\!]$

$\langle 2 \rangle 1$. $[\![d_1 \ \mathsf{strict}\ d_2]\!] = \{h \in \mathcal{H} \mid \exists h_1 \in [\![d_1]\!], h_2 \in [\![d_2]\!] : h = h_1{}^\frown h_2\}$

PROOF: Assumption that d is simple, and defs. (5.11), (5.13) and (5.12).

$\langle 2 \rangle 2$. $t \in \mathcal{H}$

PROOF: Assumption 1 and lemma 6.

$\langle 2 \rangle 3$. $\exists h_1 \in [\![d_1]\!], h_2 \in [\![d_2]\!] : t = h_1{}^\frown h_2$

$\langle 3 \rangle 1$. There exist traces s_1, s_2, and $\beta_1, \beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{s_1} [\beta_1, \mathsf{skip}] \wedge$
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{s_2} [\beta_2, \mathsf{skip}] \wedge$
$t = s_1{}^\frown s_2$

PROOF: Assumption 1 and lemma 10.

$\langle 3 \rangle 2$. $s_1 \in [\![d_1]\!], s_2 \in [\![d_2]\!]$

PROOF: ⟨3⟩1and assumptions 2 and 3 (induction hypothesis).
⟨3⟩3. Q.E.D.
PROOF: ⟨3⟩1 and ⟨3⟩2
⟨2⟩4. Q.E.D.
PROOF: ⟨2⟩1, ⟨2⟩2 and ⟨2⟩3.
⟨1⟩7. Q.E.D.
PROOF: ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4, ⟨1⟩5 and ⟨1⟩6.

□

Lemma 11 *Given* $t \in \mathcal{E}^*$. *Then* $ev.t = \bigcup_{l \in \mathcal{L}} ev.(e.l \circledS t)$.

Proof of lemma 11
ASSUME: $t \in \mathcal{E}^*$
PROVE: $ev.t = \bigcup_{l \in \mathcal{L}} ev.(e.l \circledS t)$
PROOF: by induction on t
⟨1⟩1. Induction start: $t = \langle\rangle$
⟨2⟩1. $ev.t = ev.\langle\rangle = \emptyset$
PROOF: ⟨1⟩1 and def. of $ev.\_$.
⟨2⟩2. $\bigcup_{l \in \mathcal{L}} ev.(e.l \circledS t)$

$=$	$\bigcup_{l \in \mathcal{L}} ev.(e.l \circledS \langle\rangle)$	(⟨1⟩1)
$=$	$\bigcup_{l \in \mathcal{L}} ev.\langle\rangle$	(def. of $\_\circledS\_$)
$=$	$\bigcup_{l \in \mathcal{L}} \emptyset$	(def. of $ev.\_$)
$=$	$\emptyset$	

⟨2⟩3. Q.E.D.
PROOF: ⟨2⟩1 and ⟨2⟩2.
⟨1⟩2. Induction step
ASSUME: $ev.t = \bigcup_{l \in \mathcal{L}} ev.(e.l \circledS t)$ (induction hypothesis)
PROVE: $ev.(\langle e \rangle^\frown t) = \bigcup_{l \in \mathcal{L}} ev.(e.l \circledS (\langle e \rangle^\frown t))$
⟨2⟩1. ASSUME: 1. $\mathcal{L} = \{l_1, l_2, \ldots, l_i, \ldots, l_k, \ldots\}$
2. $l.e = l_i$
⟨2⟩2. $e \in e.l_i$
PROOF: Assumption 2 and defs. of $l.\_$ and $e.\_$.
⟨2⟩3. $i \neq j \Rightarrow e \notin e.l_j$
PROOF: Assumption 2 and defs. of $l.\_$ and $e.\_$.
⟨2⟩4. $ev.(\langle e \rangle^\frown t) = \{e\} \cup ev.t$
PROOF: Def. of $ev.\_$.
⟨2⟩5. $\bigcup_{l \in \mathcal{L}} ev.(e.l \circledS (\langle e \rangle^\frown t))$

$$
\begin{array}{lll}
= & ev.(e.l_1 \circledS (\langle e \rangle^\frown t)) \cup ev.(e.l_2 \circledS (\langle e \rangle^\frown t)) \cup \cdots & \\
& \cup\, ev.(e.l_i \circledS (\langle e \rangle^\frown t)) \cup \cdots & \\
& \cup\, ev.(e.l_k \circledS (\langle e \rangle^\frown t)) \cup \cdots & \text{(Assumption 1)} \\
= & ev.(e.l_1 \circledS t) \cup ev.(e.l_2 \circledS t) \cup \cdots & (\langle 2 \rangle 2,\ \langle 2 \rangle 3 \text{ and} \\
& \cup\, ev.(\langle e \rangle^\frown e.l_i \circledS t) \cup \cdots \cup ev.(e.l_k \circledS t) \cup \cdots & \text{def. of } \_\circledS\_) \\
= & ev.(e.l_1 \circledS t) \cup ev.(e.l_2 \circledS t) \cup \cdots & \\
& \cup\, \{e\} \cup ev.(e.l_i \circledS t) \cup \cdots \cup ev.(e.l_k \circledS t) \cup \cdots & \text{(Def. of } ev.\_) \\
= & \{e\} \cup ev.(e.l_1 \circledS t) \cup ev.(e.l_2 \circledS t) \cup \cdots & \\
& \cup\, ev.(e.l_i \circledS t) \cup \cdots \cup ev.(e.l_k \circledS t) \cup \cdots & \text{(Prop. of } \cup) \\
= & \{e\} \cup \bigcup_{l \in \mathcal{L}} ev.(e.l \circledS t) & \text{(Assumption 1)} \\
= & \{e\} \cup ev.t & \text{(Ind. hyp.)}
\end{array}
$$

$\langle 2 \rangle 6$. Q.E.D.
PROOF: $\langle 2 \rangle 4$ and $\langle 2 \rangle 5$.

$\langle 1 \rangle 3$. Q.E.D.
PROOF: $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$.

□

Lemma 12 *Given diagram $d \in \mathcal{D}$. If there exist $\beta, \beta' \in \mathcal{B},\ d' \in \mathcal{D},\ e_1, e_2 \in \mathcal{E}$ such that*

$$[\beta, d] \xrightarrow{\langle e_1, e_2 \rangle} [\beta', d']$$

and

$$l.e_1 \neq l.e_2, \quad m.e_1 \neq m.e_2$$

then

$$[\beta, d] \xrightarrow{\langle e_2, e_1 \rangle} [\beta', d']$$

Proof of lemma 12
ASSUME: There exist $d' \in \mathcal{D},\ \beta, \beta' \in \mathcal{B},\ e_1, e_2 \in \mathcal{E}$ such that
1. $[\beta, d] \xrightarrow{\langle e_1, e_2 \rangle} [\beta', d']$
2. $l.e_1 \neq l.e_2$
3. $m.e_1 \neq m.e_2$

PROVE: $[\beta, d] \xrightarrow{\langle e_2, e_1 \rangle} [\beta', d']$
PROOF: by contradiction

$\langle 1 \rangle 1$. Assume not $[\beta, d] \xrightarrow{\langle e_2, e_1 \rangle} [\beta', d']$. Then there must exist $\beta'' \in \mathcal{B},\ d'' \in \mathcal{D}$ such that $[\beta, d] \xrightarrow{e_1} [\beta'', d''] \xrightarrow{e_2} [\beta', d']$ and something in $[\beta, d]$, that is not present in $[\beta'', d'']$, prevents e_2 from being enabled (i.e. $[\beta, d] \overset{e_2}{\not\to}$). There are two ways in which this can be the case.
  $\langle 2 \rangle 1$. CASE: There exist $d_1, d_1', d_2 \in \mathcal{D}$ such that $d = d_1 \ \mathsf{seq}\ d_2$, $d'' = d_1' \ \mathsf{seq}\ d_2$, $e_2 \in ev.d_2$, $l.e_2 \in ll.d_1$ and $l.e_2 \notin ll.d_1'$.
  This implies that $e_1 \in ev.d_1$ and $l.e_1 = l.e_2$ (because only e has been removed from the diagram), which is impossible because of assumption 2.
  $\langle 2 \rangle 2$. CASE: $k.e_2 = ?$, $ready(\beta, m.e_2) = \mathbf{false}$ and $ready(\beta'', m.e_2) = \mathbf{true}$.
  This implies that $\beta'' = add(\beta, m.e_2)$ which again implies that $k.e_1 = !$ and $m.e_1 = m.e_2$. But this is impossible because of assumption 3

$\langle 1 \rangle 2$. Q.E.D.

PROOF: Because the assumption of $\langle 1\rangle 1$ leads to contradiction we must have that $[\beta, d] \xrightarrow{\langle e_2, e_1\rangle} [\beta', d']$ is possible.

□

Lemma 13 *For all traces t, if there exit traces t_1, t_2, and $\beta_1, \beta_2 \in \mathcal{B}, d_1, d_2 \in \mathcal{D}$ such that*

$$\begin{array}{l} [env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}] \wedge \\ [env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}] \wedge \\ \forall l \in \mathcal{L} : e.l \circledS t = e.l \circledS t_1 \frown e.l \circledS t_2 \wedge \\ t \in \mathcal{H} \end{array}$$

then there exists $\beta \in \mathcal{B}$ such that

$$[env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2), d_1 \ \mathsf{seq}\ d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$$

Proof of lemma 13

ASSUME: t is given and there exit traces t_1, t_2, and $\beta_1, \beta_2 \in \mathcal{B}, d_1, d_2 \in \mathcal{D}$ such that

1. $[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}]$
2. $[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}]$
3. $\forall l \in \mathcal{L} : e.l \circledS t = e.l \circledS t_1 \frown e.l \circledS t_2$
4. $t \in \mathcal{H}$

PROVE: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2), d_1 \ \mathsf{seq}\ d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$

$\langle 1\rangle 1$. $ev.t = ev.t_1 \cup ev.t_2$

PROOF:

$$\begin{array}{lll} ev.t & = \bigcup_{l\in\mathcal{L}} ev.(e.l \circledS t) & \text{(Lemma 11)} \\ & = \bigcup_{l\in\mathcal{L}} ev.(e.l \circledS t_1 \frown e.l \circledS t_2) & \text{(Assumption 3)} \\ & = \bigcup_{l\in\mathcal{L}} (ev.(e.l \circledS t_1) \cup ev.(e.l \circledS t_2)) & \text{(Lemma 3)} \\ & = \bigcup_{l\in\mathcal{L}} ev.(e.l \circledS t_1) \cup \bigcup_{l\in\mathcal{L}} ev.(e.l \circledS t_2) & \text{(Properties of } \cup) \\ & = ev.t_1 \cup ev.t_2 & \text{(Lemma 11)} \end{array}$$

$\langle 1\rangle 2$. $t_1 = ev.d_1 \circledS t$

$\langle 2\rangle 1$. $ev.t = ev.d_1 \cup ev.d_2$, $ev.d_1 = ev.t_1$

PROOF: $\langle 1\rangle 1$, assumptions 1 and 2, and lemma 5.

$\langle 2\rangle 2$. LET: $ev.d_1 \circledS t = t'_1$

ASSUME: $t_1 \neq t'_1$

$\langle 2\rangle 3$. $ev.d_1 = ev.t_1 = ev.t'_1$

PROOF: $\langle 2\rangle 1$ and $\langle 2\rangle 2$.

$\langle 2\rangle 4$. There exist $e_1, e_2 \in ev.d_1$, and traces $s_1, s_2, s'_2, s_3, s'_3$ such that

$t_1 = s_1 \frown \langle e_1\rangle \frown s_2 \frown \langle e_2\rangle \frown s_3$

$t'_1 = s_1 \frown s'_2 \frown \langle e_2, e_1\rangle \frown s'_3$

PROOF: By $\langle 2\rangle 2$, t_1 and t'_1 are unequal. By $\langle 2\rangle 3$ and the assumption that d has no repetition of events, we do not have the case that t_1 is a proper prefix of t'_1 or that t'_1 is a proper prefix of t_1. Let s_1 be the longest trace such that $s_1 \sqsubseteq t_1$ and $s_1 \sqsubseteq t'_1$ and let e_1 be the first element of t_1 after s_1 ($e_1 = t_1[\#s_1 + 1]$). Then we must have that $e_1 \neq t'_1[\#s_1 + 1]$, and, by $\langle 2\rangle 3$, that $e_1 = t'_1[j]$ for some $j > \#s_1 + 1$. Let $e_2 = t'_1[j-1]$. By $\langle 2\rangle 3$, the assumption that there are no repetition of events and the fact that t_1 and t'_1 is equal up to s_1, we must have that $e_2 = t_1[k]$ for some $k > \#s_1 + 1$.

⟨2⟩5. There exist $e_1, e_2 \in ev.d_1$ such that
$\{e_1, e_2\}\circledS t_1 = \langle e_1, e_2\rangle$ and
$\{e_1, e_2\}\circledS t_1' = \langle e_2, e_1\rangle$.
PROOF: ⟨2⟩4.

⟨2⟩6. $\{e_1, e_2\}\circledS t = \langle e_2, e_1\rangle$
PROOF: ⟨2⟩2 and ⟨2⟩5.

⟨2⟩7. $l.e_1 \neq l.e_2$
PROOF: If there exists l such that $l.e_1 = l.e_2 = l$, then $e_1, e_2 \in e.l$ which, by ⟨2⟩5 and ⟨2⟩6, imply that $(\{e_1, e_2\} \cap e.l)\circledS t \neq (\{e_1, e_2\} \cap e.l)\circledS t_1$. But this contradicts assumption 3.

⟨2⟩8. $m.e_1 \neq m.e_2$
PROOF: If we assume that $m.e_1 = m.e_2$, we must have, by the assumption that there are no repetition of events, that there exists m such that $e_1 = (!, m)$ and $e_2 = (?, m)$ or $e_1 = (?, m)$ and $e_2 = (!, m)$. The first case contradicts, by ⟨2⟩6, the assumption that $t \in \mathcal{H}$ (assumption 4), and the second case contradicts, by ⟨2⟩5, the fact that $t_1 \in \mathcal{H}$ (assumption 1 and lemma 6).

⟨2⟩9. Q.E.D.
PROOF: If we assume that $t_1 \neq ev.d_1\circledS t$ (⟨2⟩2) we get the result that $l.e_1 \neq l.e_2$ (⟨2⟩7) and $m.e_1 \neq m.e_2$ (⟨2⟩8) for any pair of events $e_1, e_2 \in ev.d_1 = ev.t_1$ such that $\{e_1, e_2\}\circledS t_1 \neq (\{e_1, e_2\} \cap ev.d_1)\circledS t$. By lemma 12, this means that the order of e_1 and e_2 in t_1' is arbitrary and we may swap their position without this affecting assumption 1. We may repeat this argument until $t_1'[\#s_1 + 1] = e_1$. We may then substitute $s_1{}^\frown\langle e_1\rangle$ for s_1 in ⟨2⟩4 and repeat the argument until $t_1 = t_1' = s_1$. For this reason we can assume ⟨1⟩2 without this affecting the rest of the proof.

⟨1⟩3. $t_2 = ev.d_2\circledS t$
PROOF: Identical to proof of ⟨1⟩2

⟨1⟩4. We may now describe a sequence of transitions such that
$[env^!_{\mathcal{M}}.(d_1 \mathsf{\ seq\ } d_2), d_1 \mathsf{\ seq\ } d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$

⟨2⟩1. LET: $t = \langle e\rangle^\frown t'$

⟨2⟩2. We have that either
there exists t_1' such that $t_1 = \langle e\rangle^\frown t_1'$ and $e \notin ev.t_2$, or
there exists t_2' such that $t_2 = \langle e\rangle^\frown t_2'$ and $e \notin ev.t_1$.
PROOF: ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, the overall assumption that $ev.d_1 \cap ev.d_2 = \emptyset$, and the facts that $ev.d_1 = ev.t_1$ and $ev.d_2 = ev.t_2$ (assumptions 1 and 2, and lemma 5.

⟨2⟩3. CASE: $t_1 = \langle e\rangle^\frown t_1'$ and $e \notin ev.t_2$
PROVE: There exists $d_1' \in \mathcal{D}$ and $\beta' \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \mathsf{\ seq\ } d_2), d_1 \mathsf{\ seq\ } d_2] \xrightarrow{e} [\beta', d_1' \mathsf{\ seq\ } d_2]$

⟨3⟩1. There exists $d_1' \in \mathcal{D}$ such that
$\Pi(ll.d_1, env^!_{\mathcal{M}}.d_1, d_1) \xrightarrow{e} \Pi(ll.d_1, env^!_{\mathcal{M}}.d_1, d_1')$
PROOF: Case assumption ⟨2⟩3, assumption 1 and rule (8.3).

⟨3⟩2. There exists $d_1' \in \mathcal{D}$ such that
$\Pi(ll.d_1 \cap ll.(d_1 \mathsf{\ seq\ } d_2), env^!_{\mathcal{M}}.(d_1 \mathsf{\ seq\ } d_2), d_1) \xrightarrow{e}$
$\Pi(ll.d_1 \cap ll.(d_1 \mathsf{\ seq\ } d_2), env^!_{\mathcal{M}}.(d_1 \mathsf{\ seq\ } d_2), d_1')$

⟨4⟩1. $ll.d_1 \cap ll.(d_1 \mathsf{\ seq\ } d_2) = ll.d_1 \cap (ll.d_1 \cup ll.d_2) = ll.d_1$
PROOF: Def. of $ll.\_$.

⟨4⟩2. $k.e = !$ implies that the state of the communication medium is irrelevant.
$k.e = ?$ implies that $m.e \in env^!_{\mathcal{M}}.(d_1 \mathsf{\ seq\ } d_2)$

PROOF: Rule (8.10). $k.e = ?$ implies that $m.e \in env^!_{\mathcal{M}}.d_1$ and also that $(!, m.e) \notin ev.t'$ because $t \in \mathcal{H}$ (assumption 4), which means $(!, m.e) \notin ev.d'_1 \cup ev.d_2$ ($\langle 1\rangle 1$ and lemma 5). This implies $m.e \in env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2)$ because $m.e \in env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2) \Leftrightarrow m.e \in env^!_{\mathcal{M}}.d_1 \wedge (!, m.e) \notin ev.d_2$ (see proof of lemma 7).

$\langle 4\rangle 3$. Q.E.D.

PROOF: $\langle 3\rangle 1$, $\langle 4\rangle 1$, $\langle 4\rangle 2$.

$\langle 3\rangle 3$. There exists $d'_1 \in \mathcal{D}$ such that
$\Pi(ll.(d_1 \ \mathsf{seq}\ d_2), env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2), d_1 \ \mathsf{seq}\ d_2) \xrightarrow{e}$
$\Pi(ll.(d_1 \ \mathsf{seq}\ d_2), env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2), d'_1 \ \mathsf{seq}\ d_2)$

$\langle 4\rangle 1$. $ll.d_1 \cap ll.(d_1 \ \mathsf{seq}\ d_2) \neq \emptyset$

PROOF: $e \in ev.d_1$ and def. of $ll.\_$.

$\langle 4\rangle 2$. Q.E.D.

PROOF: $\langle 3\rangle 2$, $\langle 4\rangle 1$ and rule (8.11)

$\langle 3\rangle 4$. Q.E.D.

PROOF: $\langle 3\rangle 3$ and rule (8.3).

$\langle 2\rangle 4$. CASE: $t_2 = \langle e\rangle^\frown t'_2$ and $e \notin ev.t_1$

PROVE: There exists $d'_2 \in \mathcal{D}$ and $\beta' \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2), d_1 \ \mathsf{seq}\ d_2] \xrightarrow{e} [\beta', d_1 \ \mathsf{seq}\ d'_2]$

$\langle 3\rangle 1$. There exists $d'_2 \in \mathcal{D}$ such that
$\Pi(ll.d_2, env^!_{\mathcal{M}}.d_2, d_2) \xrightarrow{e} \Pi(ll.d_2, env^!_{\mathcal{M}}.d_2, d'_2)$

PROOF: Case assumption $\langle 2\rangle 4$, assumption 2 and rule (8.3).

$\langle 3\rangle 2$. There exists $d'_2 \in \mathcal{D}$ such that
$\Pi(ll.(d_1 \ \mathsf{seq}\ d_2) \setminus ll.d_1, env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2), d_2) \xrightarrow{e}$
$\Pi(ll.(d_1 \ \mathsf{seq}\ d_2) \setminus ll.d_1, env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2), d'_2)$

$\langle 4\rangle 1$. $l.e \in ll.(d_1 \ \mathsf{seq}\ d_2) \setminus ll.d_1$

$\langle 5\rangle 1$. $l.e \in ll.(d_1 \ \mathsf{seq}\ d_2) \setminus ll.d_1$
$\Leftrightarrow (l.e \in ll.e_1 \vee l.e \in ll.d_2) \wedge l.e \notin ll.d_1$
$\Leftrightarrow l.e \in ll.d_2 \wedge l.e \notin ll.d_1$

PROOF: Def. of $ll.\_$ and basic set theory.

$\langle 5\rangle 2$. $l.e \in ll.d_2$

PROOF: Assumption 2, case assumption, lemma 5 and def. of $ll.\_$.

$\langle 5\rangle 3$. $l.e \notin ll.d_1$

PROOF: From assumption 3, $\langle 2\rangle 1$ and the case assumption we know that $e.(l.e) \circledS t_1 = \langle\rangle$. This implies that $e.(l.e) \cap ev.t_1 = e.(l.e) \cap ev.d_2 = \emptyset$, which again imply that $l.e \notin ll.d_1$ (lemma 5, defs. of $e.\_$, $l.\_$, $ev.\_$ and $ll.\_$).

$\langle 5\rangle 4$. Q.E.D.

PROOF: $\langle 5\rangle 1$, $\langle 5\rangle 2$ and $\langle 5\rangle 3$.

$\langle 4\rangle 2$. $k.e = !$ implies that the state of the communication medium is irrelevant.
$k.e = ?$ implies that $m.e \in env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2)$

PROOF: Rule (8.10). $k.e = ?$ implies that $m.e \in env^!_{\mathcal{M}}.d_1$ and also that $(!, m.e) \notin ev.t'$ because $t \in \mathcal{H}$ (assumption 4), which means $(!, m.e) \notin ev.d'_1 \cup ev.d_2$ ($\langle 1\rangle 1$ and lemma 5). This implies $m.e \in env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2)$ because $m.e \in env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2) \Leftrightarrow m.e \in env^!_{\mathcal{M}}.d_1 \wedge (!, m.e) \notin ev.d_2$ (see proof of lemma 7).

$\langle 4\rangle 3$. Q.E.D.

PROOF: $\langle 3\rangle 2$, $\langle 4\rangle 1$, $\langle 4\rangle 2$.

$\langle 3\rangle 3$. There exists $d'_2 \in \mathcal{D}$ such that

$\Pi(ll.(d_1 \text{ seq } d_2), env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1 \text{ seq } d_2) \xrightarrow{e}$
$\Pi(ll.(d_1 \text{ seq } d_2), env^!_{\mathcal{M}}.(d_1 \text{ seq } d_2), d_1 \text{ seq } d_2')$

⟨4⟩1. $ll.(d_1 \text{ seq } d_2) \setminus ll.d_1 \neq \emptyset$
PROOF: $l.e \in ll.(d_1 \text{ seq } d_2) \setminus ll.d_1$ (see ⟨3⟩2).
⟨4⟩2. Q.E.D.
PROOF: ⟨3⟩2, ⟨4⟩1 and rule (8.12)
⟨3⟩4. Q.E.D.
PROOF: ⟨3⟩3 and rule (8.3).
⟨2⟩5. We may now substitute t for t' and repeat the argument until $t = \langle\rangle$. (For justification of this see proof of lemma 7.)
⟨2⟩6. Q.E.D.
PROOF: ⟨2⟩1-⟨2⟩4
⟨1⟩5. Q.E.D.
PROOF: ⟨1⟩1-⟨1⟩4

□

Lemma 14 *Given traces t, t_1, t_2 and oracle $p \in \{1,2\}^\infty$. If*

$$\pi_2((\{1\} \times \mathcal{E})@(p,t)) = t_1 \wedge$$
$$\pi_2((\{2\} \times \mathcal{E})@(p,t)) = t_2$$

then

$$ev.t = ev.t_1 \cup ev.t_2$$

Proof of lemma 14
ASSUME: t, t_1, t_2 and $p \in \{1,2\}^\infty$ given, and

1. $\pi_2((\{1\} \times \mathcal{E})@(p,t)) = t_1$
2. $\pi_2((\{2\} \times \mathcal{E})@(p,t)) = t_2$

PROVE: $ev.t = ev.t_1 \cup ev.t_2$
PROOF: by induction on t

⟨1⟩1. Induction start: $t = \langle\rangle$
PROVE: $ev.t = ev.t_1 \cup ev.t_2$
⟨2⟩1. $ev.t = ev.\langle\rangle = \emptyset$
PROOF: ⟨1⟩1 and def. of $ev.\_$
⟨2⟩2. $ev.t_1 \cup ev.t_2$
$= ev.\pi_2((\{1\} \times \mathcal{E})@(p,\langle\rangle))$
$\cup\ ev.\pi_2((\{2\} \times \mathcal{E})@(p,\langle\rangle))$ (Assumptions 1 and 2, and ⟨1⟩1)
$= ev.\pi_2((\{1\} \times \mathcal{E})@(\langle\rangle,\langle\rangle))$
$\cup\ ev.\pi_2((\{2\} \times \mathcal{E})@(\langle\rangle,\langle\rangle))$ (Def. 7 of $\_@\_$)
$= ev.\pi_2(\langle\rangle,\langle\rangle) \cup ev.\pi_2(\langle\rangle,\langle\rangle)$ (Def. 7 of $\_@\_$)
$= ev.\langle\rangle \cup ev.\langle\rangle$ (Def. of π_2)
$= \emptyset \cup \emptyset$ (Def. of. $ev.\_$)
$= \emptyset$
⟨2⟩3. Q.E.D.
PROOF: ⟨2⟩1 and ⟨2⟩2.
⟨1⟩2. Induction step

ASSUME: $ev.t = ev.t_1 \cup ev.t_2$ (induction hypothesis)
PROVE: $ev.(t^\frown\langle e\rangle) = ev.\pi_2((\{1\} \times \mathcal{E})\circledcirc(p, t^\frown\langle e\rangle))$
$\cup\ ev.\pi_2((\{2\} \times \mathcal{E})\circledcirc(p, t^\frown\langle e\rangle))$

$\langle 2\rangle 1$. $ev.(t^\frown\langle e\rangle)$
$= ev.t \cup ev.\langle e\rangle$ (lemma 3)
$= ev.t \cup \{e\}$ (def. of $ev.\_$)

$\langle 2\rangle 2$. LET: $k \in \{1, 2\}$

$\langle 2\rangle 3$. $ev.\pi_2((\{k\} \times \mathcal{E})\circledcirc(p, t^\frown\langle e\rangle))$
$= ev.\pi_2((\{k\} \times \mathcal{E})\circledcirc(p|_{\#t+1}, t^\frown\langle e\rangle))$ (Def. 7 of $\_\circledcirc\_$)
$= ev.\pi_2((\{k\} \times \mathcal{E})\circledcirc(p|_{\#t}{}^\frown\langle j\rangle, t^\frown\langle e\rangle))$ ($j \in \{1, 2\}$)
$= ev.\pi_2(((\{k\} \times \mathcal{E})\circledcirc(p|_{\#t}, t))$
$^\frown((\{k\} \times \mathcal{E})\circledcirc(\langle j\rangle, \langle e\rangle)))$ (Defs. of $\_^\frown\_$ and $\_\circledcirc\_$)
$= ev.(\pi_2((\{k\} \times \mathcal{E})\circledcirc(p|_{\#t}, t))$
$^\frown\pi_2((\{k\} \times \mathcal{E})\circledcirc(\langle j\rangle, \langle e\rangle)))$ (Lemma 8)
$= ev.\pi_2((\{k\} \times \mathcal{E})\circledcirc(p|_{\#t}, t))$
$\cup\ ev.\pi_2((\{k\} \times \mathcal{E})\circledcirc(\langle j\rangle, \langle e\rangle))$ (Lemma 3)
$= ev.\pi_2((\{k\} \times \mathcal{E})\circledcirc(p, t))$
$\cup\ ev.\pi_2((\{k\} \times \mathcal{E})\circledcirc(\langle j\rangle, \langle e\rangle))$ (Def. 7 of $\_\circledcirc\_$)
$= t_k \cup ev.\pi_2((\{k\} \times \mathcal{E})\circledcirc(\langle j\rangle, \langle e\rangle))$ (Assumptions 1 and 2)

$\langle 2\rangle 4$. $ev.\pi_2((\{1\} \times \mathcal{E})\circledcirc(p, t^\frown\langle e\rangle))$
$\cup\ ev.\pi_2((\{2\} \times \mathcal{E})\circledcirc(p, t^\frown\langle e\rangle))$
$= t_1 \cup ev.\pi_2((\{1\} \times \mathcal{E})\circledcirc(\langle j\rangle, \langle e\rangle))$
$\cup\ t_2 \cup ev.\pi_2((\{2\} \times \mathcal{E})\circledcirc(\langle j\rangle, \langle e\rangle))$ ($\langle 2\rangle 3$, $j \in \{1, 2\}$)
$= t \cup ev.\pi_2((\{1\} \times \mathcal{E})\circledcirc(\langle j\rangle, \langle e\rangle))$
$\cup\ ev.\pi_2((\{2\} \times \mathcal{E})\circledcirc(\langle j\rangle, \langle e\rangle))$ (induction hypothesis)

$\langle 2\rangle 5$. CASE: $j = 1$
$t \cup ev.\pi_2((\{1\} \times \mathcal{E})\circledcirc(\langle 1\rangle, \langle e\rangle))$
$\cup\ ev.\pi_2((\{2\} \times \mathcal{E})\circledcirc(\langle 1\rangle, \langle e\rangle))$
$= t \cup ev.\pi_2(\langle 1\rangle, \langle e\rangle) \cup ev.\pi_2(\langle\rangle, \langle\rangle)$ (Def. 7 of $\_\circledcirc\_$)
$= ev.t \cup ev.\langle e\rangle \cup ev.\langle\rangle$ (Def. of π_2)
$= ev.t \cup \{e\} \cup \emptyset$ (Def. of $ev.\_$)
$= ev.t \cup \{e\}$

$\langle 2\rangle 6$. CASE: $j = 2$
$t \cup ev.\pi_2((\{1\} \times \mathcal{E})\circledcirc(\langle 2\rangle, \langle e\rangle))$
$\cup\ ev.\pi_2((\{2\} \times \mathcal{E})\circledcirc(\langle 2\rangle, \langle e\rangle))$
$= t \cup ev.\pi_2(\langle\rangle, \langle\rangle) \cup ev.\pi_2(\langle 2\rangle, \langle e\rangle)$ (Def. 7 of $\_\circledcirc\_$)
$= ev.t \cup ev.\langle\rangle \cup ev.\langle e\rangle$ (Def. of π_2)
$= ev.t \cup \emptyset \cup \{e\}$ (Def. of $ev.\_$)
$= ev.t \cup \{e\}$

$\langle 2\rangle 7$. Q.E.D.
PROOF: $\langle 2\rangle 1$-$\langle 2\rangle 6$

$\langle 1\rangle 3$. Q.E.D.
PROOF: $\langle 1\rangle 1$ and $\langle 1\rangle 2$

□

Lemma 15 *For all t, if there exist traces t_1, t_2, and $\beta_1, \beta_2 \in \mathcal{B}, d_1, d_2 \in \mathcal{D}$, and $p \in \{1,2\}^\infty$ such that*

$$\begin{array}{l}[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}] \wedge \\ [env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}] \wedge \\ \pi_2((\{1\} \times \mathcal{E}) \circledcirc (p,t)) = t_1 \wedge \\ \pi_2((\{2\} \times \mathcal{E}) \circledcirc (p,t)) = t_2 \wedge \\ t \in \mathcal{H}\end{array}$$

then there exists $\beta \in \mathcal{B}$ such that

$$[env^!_{\mathcal{M}}.(d_1 \ \mathsf{par}\ d_2), d_1 \ \mathsf{par}\ d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$$

Proof of lemma 15

ASSUME: t is given and there exist traces t_1, t_2, and $\beta_1, \beta_2 \in \mathcal{B}, d_1, d_2 \in \mathcal{D}$, and $p \in \{1,2\}^\infty$ such that
1. $[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}]$
2. $[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}]$
3. $\pi_2((\{1\} \times \mathcal{E}) \circledcirc (p,t)) = t_1$
4. $\pi_2((\{2\} \times \mathcal{E}) \circledcirc (p,t)) = t_2$
5. $t \in \mathcal{H}$

PROVE: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \ \mathsf{par}\ d_2), d_1 \ \mathsf{par}\ d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$

$\langle 1 \rangle 1$. $ev.t = ev.t_1 \cup ev.t_2$
PROOF: Assumptions 3 and 4, and lemma 14.

$\langle 1 \rangle 2$. We describe a sequence of transitions such that
$[env^!_{\mathcal{M}}.(d_1 \ \mathsf{par}\ d_2), d_1 \ \mathsf{par}\ d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$
We do this with guidance of the oracle p

$\langle 2 \rangle 1$. LET: $t = \langle e \rangle^\frown t'$

$\langle 2 \rangle 2$. CASE: $p = \langle 1 \rangle^\frown p'$
PROVE: There exists $d_1' \in \mathcal{D}$ and $\beta' \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \ \mathsf{par}\ d_2), d_1 \ \mathsf{par}\ d_2] \xrightarrow{e} [\beta', d_1' \ \mathsf{par}\ d_2]$

$\langle 3 \rangle 1$. $t_1 = \langle e \rangle^\frown \pi_2((\{1\} \times \mathcal{E}) \circledcirc (p', t'))$
PROOF:

$$\begin{array}{lll} t_1 & = \pi_2((\{1\} \times \mathcal{E}) \circledcirc (p,t)) & \text{(Assumption 3)} \\ & = \pi_2((\{1\} \times \mathcal{E}) \circledcirc (\langle 1 \rangle^\frown p', \langle e \rangle^\frown t')) & (\langle 2 \rangle 1 \text{ and } \langle 2 \rangle 2) \\ & = \pi_2((\langle 1 \rangle, \langle e \rangle)^\frown ((\{1\} \times \mathcal{E}) \circledcirc (p', t'))) & \text{(Def. 7 of } \_\circledcirc\_) \\ & = \pi_2(\langle 1 \rangle, \langle e \rangle)^\frown \pi_2((\{1\} \times \mathcal{E}) \circledcirc (p', t')) & \text{(Lemma 8)} \\ & = \langle e \rangle^\frown \pi_2((\{1\} \times \mathcal{E}) \circledcirc (p', t')) & \text{(Def. of } \pi_2) \end{array}$$

$\langle 3 \rangle 2$. There exists $d_1' \in \mathcal{D}$ such that
$\Pi(ll.d_1, env^!_{\mathcal{M}}.d_1, d_1) \xrightarrow{e} \Pi(ll.d_1, env^!_{\mathcal{M}}.d_1, d_1')$
PROOF: $\langle 3 \rangle 1$, assumption 1 and rule (8.3).

$\langle 3 \rangle 3$. There exists $d_1' \in \mathcal{D}$ such that
$\Pi(ll.d_1 \cap ll.(d_1 \ \mathsf{par}\ d_2), env^!_{\mathcal{M}}.(d_1 \ \mathsf{par}\ d_2), d_1) \xrightarrow{e}$
$\Pi(ll.d_1 \cap ll.(d_1 \ \mathsf{par}\ d_2), env^!_{\mathcal{M}}.(d_1 \ \mathsf{par}\ d_2), d_1')$

$\langle 4 \rangle 1$. $ll.d_1 \cap ll.(d_1 \ \mathsf{par}\ d_2) = ll.d_1 \cap (ll.d_1 \cup ll.d_2) = ll.d_1$
PROOF: Def. of $ll.\_$.

$\langle 4 \rangle 2$. $k.e = !$ implies that the state of the communication medium is irrelevant.
$k.e = ?$ implies that $m.e \in env^!_{\mathcal{M}}.(d_1 \ \mathsf{par}\ d_2)$

PROOF: Rule (8.10). $k.e = ?$ implies that $m.e \in env^!_{\mathcal{M}}.d_1$ and also that $(!, m.e) \notin ev.t'$ because $t \in \mathcal{H}$ (assumption 5), which means $(!, m.e) \notin ev.d'_1 \cup ev.d_2$ ($\langle 1 \rangle 1$ and lemma 5). This implies $m.e \in env^!_{\mathcal{M}}.(d_1 \text{ par } d_2)$ because $m.e \in env^!_{\mathcal{M}}.(d_1 \text{ par } d_2) \Leftrightarrow m.e \in env^!_{\mathcal{M}}.d_1 \wedge (!, m.e) \notin ev.d_2$ (see proof of lemma 7).

$\langle 4 \rangle 3$. Q.E.D.
PROOF: $\langle 3 \rangle 2$, $\langle 4 \rangle 1$, $\langle 4 \rangle 2$.

$\langle 3 \rangle 4$. There exists $d'_1 \in \mathcal{D}$ such that
$\Pi(ll.(d_1 \text{ par } d_2), env^!_{\mathcal{M}}.(d_1 \text{ par } d_2), d_1 \text{ par } d_2) \xrightarrow{e}$
$\Pi(ll.(d_1 \text{ par } d_2), env^!_{\mathcal{M}}.(d_1 \text{ par } d_2), d'_1 \text{ par } d_2)$
PROOF: $\langle 3 \rangle 3$, and rule (8.13)

$\langle 3 \rangle 5$. Q.E.D.
PROOF: $\langle 3 \rangle 4$ and rule (8.3).

$\langle 2 \rangle 3$. CASE: $p = \langle 2 \rangle^\frown p'$
PROVE: There exists $d'_2 \in \mathcal{D}$ and $\beta' \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \text{ par } d_2), d_1 \text{ par } d_2] \xrightarrow{e} [\beta', d_1 \text{ par } d'_2]$
PROOF: Identical to proof of $\langle 2 \rangle 2$ with t_2 and rule (8.14) substituted for t_1 and rule (8.13).

$\langle 2 \rangle 4$. We may now substitute t for t' and repeat the argument until $t = \langle\rangle$. (For justification of this see proof of lemma 9.)

$\langle 2 \rangle 5$. Q.E.D.
PROOF: $\langle 2 \rangle 1$-$\langle 2 \rangle 3$

$\langle 1 \rangle 3$. Q.E.D.
PROOF: $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$

□

Lemma 16 *For all traces t, if there exit traces t_1, t_2, and $\beta_1, \beta_2 \in \mathcal{B}, d_1, d_2 \in \mathcal{D}$ such that*

$$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}] \wedge$$
$$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}] \wedge$$
$$t = t_1{}^\frown t_2 \wedge$$
$$t \in \mathcal{H}$$

then there exists $\beta \in \mathcal{B}$ such that

$$[env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2), d_1 \text{ strict } d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$$

Proof of lemma 16

ASSUME: t is given and there exist traces t_1, t_2 and $\beta_1, \beta_2 \in \mathcal{B}$ such that
1. $[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}]$
2. $[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}]$
3. $t = t_1{}^\frown t_2$
4. $t \in \mathcal{H}$

PROVE: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2), d_1 \text{ strict } d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$

$\langle 1 \rangle 1$. There exists $\beta' \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2), d_1] \xrightarrow{t_1} [\beta', \mathsf{skip}]$

PROOF: by contradiction
ASSUME: ⟨1⟩1 is not the case. By assumption 1 and rules (8.3), (8.10), (8.11), (8.12), (8.13), (8.14) and (8.15), two cases must be considered.
⟨2⟩1. CASE: There exists $e \in ev.t_1$ such that $l.e \in ll.d_1$, but $l.e \notin ll.(d_1 \text{ strict } d_2)$
PROOF: By def. of $ll.\_$, $ll.d_1 \subseteq ll.(d_1 \text{ strict } d_2) = ll.d_1 \cup ll.d_2$ so this is impossible.
⟨2⟩2. CASE: There exists message m such that $(?, m) \in ev.t_1$ and $m \in env^!_{\mathcal{M}}.d_1$, but $m \notin env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2)$
⟨3⟩1. $m \in msg.d_1$
PROOF:

$(?, m) \in ev.t_1$		(⟨2⟩2)
$\Rightarrow$	$(?, m) \in ev.d_1$	(Lemma 5)
$\Rightarrow$	$m \in msg.d_1$	(Defs. of msg and ev)

⟨3⟩2. $(!, m) \notin ev.d_1$
PROOF:

$m \in env^!_{\mathcal{M}}.d_1$		(⟨2⟩2)
$\Rightarrow$	$(!, m) \in env.d_1$	(Def. 1)
$\Rightarrow$	$m \in msg.d_1 \wedge (!, m) \notin ev.d_1$	(Def. 1)
$\Rightarrow$	$(!, m) \notin ev.d_1$	

⟨3⟩3. $(!, m) \in ev.t_2$
PROOF:

$m \notin env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2)$		(⟨2⟩2)
$\Rightarrow$	$(!, m) \notin env.(d_1 \text{ strict } d_2)$	(Def. 1)
$\Rightarrow$	$m \notin msg.(d_1 \text{ strict } d_2) \vee$ $(!, m) \in ev.(d_1 \text{ strict } d_2)$	(Def. 1)
$\Rightarrow$	$(m \notin msg.d_1 \wedge m \notin msg.d_2) \vee$ $(!, m) \in ev.d_1 \vee (!, m) \in ev.d_2$	(Defs. of msg and ev)
$\Rightarrow$	$(!, m) \in ev.d_1 \vee (!, m) \in ev.d_2$	(⟨3⟩1)
$\Rightarrow$	$(!, m) \in ev.d_2$	(⟨3⟩2)
$\Rightarrow$	$(!, m) \in ev.t_2$	(Assumption 2 and Lemma 5)

⟨3⟩4. There exist traces s_1, s_2, s_3, s_4 such that
$t_1 = s_1 \frown \langle (?, m) \rangle \frown s_2$ and
$t_2 = s_3 \frown \langle (!, m) \rangle \frown s_4$
PROOF: ⟨2⟩2, ⟨3⟩3 and def. 3 of $ev.\_$.
⟨3⟩5. Q.E.D.
PROOF: By ⟨3⟩4 and assumption 3, $t = s_1 \frown \langle (?, m) \rangle \frown s_2 \frown s_3 \frown \langle (!, m) \rangle \frown s_4$. But by assumption 4 this is impossible, so we get a contradiction and ⟨2⟩2 is impossible.
⟨2⟩3. Q.E.D.
PROOF: Since both ⟨2⟩1 and ⟨2⟩2 are impossible we must have ⟨1⟩1.
⟨1⟩2. There exists $\beta' \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2), d_1 \text{ strict } d_2] \xrightarrow{t_1} [\beta', d_2]$
PROOF: ⟨1⟩1, rules (8.3) and (8.15), and def. of skip as the identity element of strict.
⟨1⟩3. There exists $\beta'' \in \mathcal{B}$ such that
$[\beta', d_2] \xrightarrow{t_2} [\beta'', \text{skip}]$
PROOF: by contradiction
ASSUME: ⟨1⟩3 is not the case
⟨2⟩1. There exists message m such that $(?, m) \in ev.t_2$ and $m \in env^!_{\mathcal{M}}.d_2$, but $m \notin \beta'$
PROOF: Assumption 2, assumption ⟨1⟩3 is not the case, and rules (8.3), (8.10), (8.11), (8.12), (8.13), (8.14) and (8.15).
⟨2⟩2. $m \in msg.d_2$

PROOF: $(?, m) \in ev.t_2$ ($\langle 2 \rangle 1$)
$\Rightarrow$ $(?, m) \in ev.d_2$ (Lemma 5)
$\Rightarrow$ $m \in msg.d_2$ (Defs. of msg and ev)

$\langle 2 \rangle 3$. $(!, m) \notin ev.d_2$
PROOF: $m \in env^!_{\mathcal{M}}.d_2$ ($\langle 2 \rangle 1$)
$\Rightarrow$ $(!, m) \in env.d_2$ (Def. 1)
$\Rightarrow$ $m \in msg.d_2 \wedge (!, m) \notin ev.d_2$ (Def. 1)
$\Rightarrow$ $(!, m) \notin ev.d_2$

$\langle 2 \rangle 4$. $(!, m) \in ev.d_2$
PROOF: $m \notin \beta'$ ($\langle 2 \rangle 1$)
$\Rightarrow$ $m \notin env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2) \wedge (!, m) \notin ev.t_1$ ($\langle 1 \rangle 1$ and rules)
$\Rightarrow$ $(!, m) \notin env.(d_1 \text{ strict } d_2) \wedge (!, m) \notin ev.d_1$ (Def. 1 and lemma 5)
$\Rightarrow$ $(m \notin msg.(d_1 \text{ strict } d_2) \vee$
$(!, m) \in ev.(d_1 \text{ strict } d_2)) \wedge (!, m) \notin ev.d_1$ (Def. 1)
$\Rightarrow$ $((m \notin msg.d_1 \wedge m \notin msg.d_2) \vee$
$(!, m) \in ev.d_1 \vee (!, m) \in ev.d_2) \wedge$
$(!, m) \notin ev.d_1$ (Defs. of msg and ev)
$\Rightarrow$ $((!, m) \in ev.d_1 \vee (!, m) \in ev.d_2) \wedge$
$(!, m) \notin ev.d_1$ ($\langle 2 \rangle 2$)
$\Rightarrow$ $(!, m) \in ev.d_2$

$\langle 2 \rangle 5$. Q.E.D.
PROOF: $\langle 2 \rangle 3$ and $\langle 2 \rangle 4$ yield a contradiction, so we must have that $\langle 1 \rangle 3$ holds.

$\langle 1 \rangle 4$. Q.E.D.
PROOF: $\langle 1 \rangle 2$, $\langle 1 \rangle 3$ and assumption 3 by letting $\beta = \beta''$.

□

Theorem 2 (Completeness) *Given a simple diagram d. Then, for all traces t, if*

$$t \in [\![d]\!]$$

then there exists $\beta \in \mathcal{B}$ such that

$$[env^!_{\mathcal{M}}.d, d] \xrightarrow{t} [\beta, \mathsf{skip}]$$

Proof of theorem 2
ASSUME: $t \in [\![d]\!]$
PROVE: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d, d] \xrightarrow{t} [\beta, \mathsf{skip}]$
PROOF: by induction on the structure of d.

$\langle 1 \rangle 1$. CASE: $d = \mathsf{skip}$ (induction start: empty diagram)
ASSUME: $t \in [\![\mathsf{skip}]\!]$
PROVE: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.\mathsf{skip}, \mathsf{skip}] \xrightarrow{t} [\beta, \mathsf{skip}]$
$\langle 2 \rangle 1$. $t = \langle\rangle$
$\langle 3 \rangle 1$. $[\![\mathsf{skip}]\!] = \{\langle\rangle\}$
PROOF: Assumption that d is simple, and definition (5.3).
$\langle 3 \rangle 2$. Q.E.D.

PROOF: ⟨3⟩1.

⟨2⟩2. $[env^!_{\mathcal{M}}.\mathsf{skip}, \mathsf{skip}] \xrightarrow{\langle\rangle} [\emptyset, \mathsf{skip}]$
PROOF: Rule (8.7) and def. of $env^!_{\mathcal{M}}$.

⟨2⟩3. Q.E.D.
PROOF: ⟨2⟩1 and ⟨2⟩2.

⟨1⟩2. CASE: $d = (!, m)$ (induction start: single output)
ASSUME: $t \in [\![(!, m)]\!]$
PROVE: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(!, m), (!, m)] \xrightarrow{t} [\beta, \mathsf{skip}]$

⟨2⟩1. $t = \langle (!, m) \rangle$

⟨3⟩1. $[\![(!, m)]\!] = \{\langle (!, m) \rangle\}$
PROOF: Assumption that d is simple and def. (5.4).

⟨3⟩2. Q.E.D.
PROOF: ⟨3⟩1

⟨2⟩2. $[env^!_{\mathcal{M}}.(!, m), (!, m)] \xrightarrow{(!,m)} [\{m\}, \mathsf{skip}]$

⟨3⟩1. $\Pi(ll.(!, m), env^!_{\mathcal{M}}.(!, m), (!, m))$
$\xrightarrow{(!,m)} \Pi(ll.(!, m), env^!_{\mathcal{M}}.(!, m), \mathsf{skip})$
PROOF: $l.(!, m) \in ll.(!, m) = \{l.(!, m)\}$ (def. of $ll.\_$), $k.(!, m) = !$ (def. of $k.\_$), and rule (8.10).

⟨3⟩2. Q.E.D.
PROOF: ⟨3⟩1, rule (8.3) and def. of $env^!_{\mathcal{M}}$.

⟨2⟩3. Q.E.D.
PROOF: ⟨2⟩1 and ⟨2⟩2.

⟨1⟩3. CASE: $d = (?, m)$ (induction start: single input)
ASSUME: $t \in [\![(?, m)]\!]$
PROVE: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(?, m), (?, m)] \xrightarrow{t} [\beta, \mathsf{skip}]$

⟨2⟩1. $t = \langle (?, m) \rangle$

⟨3⟩1. $[\![(?, m)]\!] = \{\langle (?, m) \rangle\}$
PROOF: Assumption that d is simple and def. (5.4).

⟨3⟩2. Q.E.D.
PROOF: ⟨3⟩1

⟨2⟩2. $[env^!_{\mathcal{M}}.(?, m), (?, m)] \xrightarrow{(?,m)} [\emptyset, \mathsf{skip}]$

⟨3⟩1. $\Pi(ll.(?, m), env^!_{\mathcal{M}}.(?, m), (?, m))$
$\xrightarrow{(?,m)} \Pi(ll.(?, m), env^!_{\mathcal{M}}.(?, m), \mathsf{skip})$
PROOF: $l.(?, m) \in ll.(?, m) = \{l.(?, m)\}$ (def. of $ll.\_$), $ready(env^!_{\mathcal{M}}.(?, m))$, $m.(?, m) = ready(\{m\}, m) = m \in \{m\}$ (defs. of $ready$, $env^!_{\mathcal{M}}.\_$ and $m.\_$), and rule (8.10).

⟨3⟩2. Q.E.D.
PROOF: ⟨3⟩1, rule (8.3) and def. of $env^!_{\mathcal{M}}$.

⟨2⟩3. Q.E.D.
PROOF: ⟨2⟩1 and ⟨2⟩2.

⟨1⟩4. CASE: $d = d_1 \ \mathsf{seq} \ d_2$ (induction step)
ASSUME: 1. $t \in [\![d_1 \ \mathsf{seq} \ d_2]\!]$
2. For all t_1, if $t_1 \in [\![d_1]\!]$, then there exists $\beta_1 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}]$ (induction hypothesis)

3. For all t_2, if $t_2 \in [\![d_2]\!]$, then there exists $\beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}]$ (induction hypothesis)

PROVE: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2), d_1 \ \mathsf{seq}\ d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$

$\langle 2 \rangle 1$. $t \in \{h \in \mathcal{H} \mid \exists h_1 \in [\![d_1]\!], h_2 \in [\![d_2]\!] :$
$\forall l \in \mathcal{L} : e.l \circledS h = e.l \circledS h_1 \frown e.l \circledS h_2\}$

PROOF: Assumption d is simple, assumption 1, and defs. (5.5), (5.7) and (5.6).

$\langle 2 \rangle 2$. $t \in \mathcal{H} \wedge \exists h_1 \in [\![d_1]\!], h_2 \in [\![d_2]\!] : \forall l \in \mathcal{L} : e.l \circledS t = e.l \circledS h_1 \frown e.l \circledS h_2$

PROOF: $\langle 2 \rangle 1$

$\langle 2 \rangle 3$. There exist traces h_1, h_2, and $\beta_1, \beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{h_1} [\beta_1, \mathsf{skip}] \wedge$
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{h_2} [\beta_2, \mathsf{skip}] \wedge$
$\forall l \in \mathcal{L} : e.l \circledS t = e.l \circledS h_1 \frown e.l \circledS h_2 \wedge$
$t \in \mathcal{H}$

PROOF: Assumptions 2 and 3 (induction hypothesis), and $\langle 2 \rangle 2$.

$\langle 2 \rangle 4$. There exist $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \ \mathsf{seq}\ d_2), d_1 \ \mathsf{seq}\ d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$

PROOF: $\langle 2 \rangle 3$ and lemma 13.

$\langle 2 \rangle 5$. Q.E.D.

PROOF: $\langle 2 \rangle 4$.

$\langle 1 \rangle 5$. CASE: $d = d_1 \ \mathsf{par}\ d_2$ (induction step)

ASSUME: 1. $t \in [\![d_1 \ \mathsf{par}\ d_2]\!]$
2. For all t_1, if $t_1 \in [\![d_1]\!]$, then there exists $\beta_1 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}]$ (induction hypothesis)
3. For all t_2, if $t_2 \in [\![d_2]\!]$, then there exists $\beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}]$ (induction hypothesis)

PROVE: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \ \mathsf{par}\ d_2), d_1 \ \mathsf{par}\ d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$

$\langle 2 \rangle 1$. $t \in \{h \in \mathcal{H} \mid \exists p \in \{1,2\}^\infty :$
$\pi_2((\{1\} \times \mathcal{E}) \circledT (p, h)) \in [\![d_1]\!] \wedge$
$\pi_2((\{2\} \times \mathcal{E}) \circledT (p, h)) \in [\![d_2]\!]$

PROOF: Assumption d is simple, assumption 1, and defs. (5.8), (5.10) and (5.9).

$\langle 2 \rangle 2$. $t \in \mathcal{H} \wedge \exists p \in \{1,2\}^\infty :$
$\pi_2((\{1\} \times \mathcal{E}) \circledT (p, t)) \in [\![d_1]\!] \wedge$
$\pi_2((\{2\} \times \mathcal{E}) \circledT (p, t)) \in [\![d_2]\!]$

PROOF: $\langle 2 \rangle 1$

$\langle 2 \rangle 3$. There exist $\beta_1, \beta_2 \in \mathcal{B}, p \in \{1,2\}^\infty$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{\pi_2((\{1\} \times \mathcal{E}) \circledT (p,t))} [\beta_1, \mathsf{skip}] \wedge$
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{\pi_2((\{2\} \times \mathcal{E}) \circledT (p,t))} [\beta_2, \mathsf{skip}]$

PROOF: $\langle 2 \rangle 2$ and assumptions 2 and 3 (induction hypothesis).

$\langle 2 \rangle 4$. There exist traces t_1, t_2, and $\beta_1, \beta_2 \in \mathcal{B}, p \in \{1,2\}^\infty$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}] \wedge$
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}] \wedge$

$\pi_2((\{1\} \times \mathcal{E})⊛(p,t)) = t_1 \wedge$
$\pi_2((\{2\} \times \mathcal{E})⊛(p,t)) = t_2 \wedge$
$t \in \mathcal{H}$
PROOF: $\langle 2\rangle 3$

$\langle 2\rangle 5$. There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \text{ par } d_2), d_1 \text{ par } d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$
PROOF: $\langle 2\rangle 4$ and lemma 15.

$\langle 2\rangle 6$. Q.E.D.
PROOF: $\langle 2\rangle 5$.

$\langle 1\rangle 6$. CASE: $d = d_1$ strict d_2 (induction step)
ASSUME: 1. $t \in [\![d_1 \text{ strict } d_2]\!]$
2. For all t_1, if $t_1 \in [\![d_1]\!]$, then there exists $\beta_1 \in \mathcal{B}$ such that $[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1} [\beta_1, \mathsf{skip}]$ (induction hypothesis)
3. For all t_2, if $t_2 \in [\![d_2]\!]$, then there exists $\beta_2 \in \mathcal{B}$ such that $[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{t_2} [\beta_2, \mathsf{skip}]$ (induction hypothesis)
PROVE: There exists $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2), d_1 \text{ strict } d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$

$\langle 2\rangle 1$. $t \in \{h \in \mathcal{H} \mid \exists h_1 \in [\![d_1]\!], h_2 \in [\![d_2]\!] : h = h_1 \frown h_2\}$
PROOF: Assumption d is simple, assumption 1, and defs. (5.11), (5.13) and (5.12).

$\langle 2\rangle 2$. $t \in \mathcal{H} \wedge \exists h_1 \in [\![d_1]\!], h_2 \in [\![d_2]\!] : t = h_1 \frown h_2$
PROOF: $\langle 2\rangle 1$

$\langle 2\rangle 3$. There exist traces h_1, h_2, and $\beta_1, \beta_2 \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{h_1} [\beta_1, \mathsf{skip}] \wedge$
$[env^!_{\mathcal{M}}.d_2, d_2] \xrightarrow{h_2} [\beta_2, \mathsf{skip}] \wedge$
$t = h_1 \frown h_2 \wedge$
$t \in \mathcal{H}$
PROOF: Assumptions 2 and 3 (induction hypothesis), and $\langle 2\rangle 2$.

$\langle 2\rangle 4$. There exist $\beta \in \mathcal{B}$ such that
$[env^!_{\mathcal{M}}.(d_1 \text{ strict } d_2), d_1 \text{ strict } d_2] \xrightarrow{t} [\beta, \mathsf{skip}]$
PROOF: $\langle 2\rangle 3$ and lemma 16.

$\langle 2\rangle 5$. Q.E.D.
PROOF: $\langle 2\rangle 4$.

$\langle 1\rangle 7$. Q.E.D.
PROOF: $\langle 1\rangle 1$, $\langle 1\rangle 2$, $\langle 1\rangle 3$, $\langle 1\rangle 4$, $\langle 1\rangle 5$ and $\langle 1\rangle 6$.

□

A.3 Sequence diagrams with high-level operators and finite behavior

In this section we prove termination, soundness and completeness of diagrams with high-level operators with finite behavior. This means the operators refuse, assert, alt, xalt, and loop$\langle n\rangle$ with $n \neq \infty$. Diagrams with infinite loops are addresed in section A.4.

In the above section we assumed $[\![d]\!] = \{(T, \emptyset)\}$ for any diagram d. In this section we assume the general semantic model $[\![d]\!] = \{(p_1, n_1), (p_2, n_2), \ldots, (p_m, p_m)\}$, but write $t \in [\![d]\!]$ as a shorthand for $t \in \bigcup_{i=1,\ldots,m}(p_i \cup n_i)$ when we find this suitable. Further we treat the denotation of assert to be $[\![\mathsf{assert}\ d]\!] = [\![d]\!]$ for the purpose of the following proofs.

By termination we mean a state $[\beta, d]$ where no further execution steps are possible. Usually this means that $d = \mathsf{skip}$, but it is possible to make other diagrams where no execution is possible, e.g. the diagram $d = (?, (m, l, l))\ \mathsf{seq}\ (!, (m, l, l))$ which specify that lifeline l receives message m from itself before it sends it.

Theorem 3 (Termination) *Given a diagram $d \in \mathcal{D}$ without infinite loops. If we assume progression of execution, then execution of $[env^{!}_{\mathcal{M}}.d, d]$ will terminate.*

Proof of theorem 3

ASSUME: Diagram $d \in \mathcal{D}$ without infinite loops
PROVE: Execution of $[env^{!}_{\mathcal{M}}.d_1, d_1]$ terminates
$\langle 1 \rangle 1$. LET: $w \in \mathcal{D} \to \mathbb{N}$ be a weight function defined as

$$\begin{aligned}
w(\mathsf{skip}) &\stackrel{\mathsf{def}}{=} 0 \\
w(e) &\stackrel{\mathsf{def}}{=} 1\ (\textbf{for}\ e \in \mathcal{E}) \\
w(d_1\ \mathsf{seq}\ d_2) &\stackrel{\mathsf{def}}{=} w(d_1\ \mathsf{par}\ d_2) \\
&\stackrel{\mathsf{def}}{=} w(d_1\ \mathsf{strict}\ d_2) &&\stackrel{\mathsf{def}}{=} w(d_1) + w(d_2) + 1 \\
w(d_1\ \mathsf{alt}\ d_2) &\stackrel{\mathsf{def}}{=} w(d_1\ \mathsf{xalt}\ d_2) &&\stackrel{\mathsf{def}}{=} \mathbf{max}(w(d_1), w(d_2)) + 1 \\
w(\mathsf{refuse}\ d_1) &\stackrel{\mathsf{def}}{=} w(\mathsf{assert}\ d_1) &&\stackrel{\mathsf{def}}{=} w(d_1) + 1 \\
w(\mathsf{loop}\langle n \rangle\ d_1) &&&\stackrel{\mathsf{def}}{=} n \cdot (w(d_1) + 2)
\end{aligned}$$

$\langle 1 \rangle 2$. $w(d) \geq 0$
PROOF: by induction on the structure of d.
Induction start:
$\langle 2 \rangle 1$. CASE: $d = \mathsf{skip}$
$\langle 3 \rangle 1$. $w(d) = 0$
PROOF: Def. of w.
$\langle 3 \rangle 2$. Q.E.D.
PROOF: $\langle 3 \rangle 1$.
$\langle 2 \rangle 2$. CASE: $d = e$ (single event)
$\langle 3 \rangle 1$. $w(d) = 1$
PROOF: Def. of w.
$\langle 3 \rangle 2$. Q.E.D.
PROOF: $\langle 3 \rangle 1$.
Induction step:
ASSUME: $w(d_1) \geq 0 \wedge w(d_2) \geq 0$ (Induction hypothesis)
$\langle 2 \rangle 3$. CASE: $d = d_1\ \mathsf{seq}\ d_2$ or $d = d_1\ \mathsf{par}\ d_2$ or $d = d_1\ \mathsf{strict}\ d_2$
$\langle 3 \rangle 1$. $w(d) = w(d_1) + w(d_2) + 1 > 0$
PROOF: Def. of w and induction hypothesis.
$\langle 3 \rangle 2$. Q.E.D.
PROOF: $\langle 3 \rangle 1$
$\langle 2 \rangle 4$. CASE: $d = d_1\ \mathsf{alt}\ d_2$ or $d = d_1\ \mathsf{xalt}\ d_2$
$\langle 3 \rangle 1$. $w(d) = \mathbf{max}(w(d_1), w(d_2)) + 1 > 0$

PROOF: Def. of w and $\mathbf{max}(w(d_1), w(d_2)) \geq 0$ (induction hypothesis and properties of **max**).

$\langle 3 \rangle 2$. Q.E.D.

PROOF: $\langle 3 \rangle 1$

$\langle 2 \rangle 5$. CASE: $d = \mathsf{refuse}\ d_1$ or $d = \mathsf{assert}\ d_1$

$\langle 3 \rangle 1$. $w(d) = w(d_1) + 1 > 0$

PROOF: Def. of w and induction hypothesis.

$\langle 3 \rangle 2$. Q.E.D.

PROOF: $\langle 3 \rangle 1$.

$\langle 2 \rangle 6$. CASE: $d = \mathsf{loop}\langle n \rangle\ d_1$

$\langle 3 \rangle 1$. $w(d) = n \cdot (w(d_1) + 2)$

PROOF: Def. of w.

$\langle 3 \rangle 2$. $w(d_1) + 2 > 0$

PROOF: Induction hypothesis.

$\langle 3 \rangle 3$. $n \geq 0$

PROOF: Def. of $\mathcal{D}$.

$\langle 3 \rangle 4$. $w(d) \geq 0$

PROOF: $\langle 3 \rangle 1$, $\langle 3 \rangle 2$ and $\langle 3 \rangle 3$.

$\langle 3 \rangle 5$. Q.E.D.

PROOF: $\langle 3 \rangle 4$.

$\langle 2 \rangle 7$. Q.E.D.

PROOF: $\langle 2 \rangle 1$-$\langle 2 \rangle 6$.

$\langle 1 \rangle 3$. $[\beta, d] \xrightarrow{e} [\beta', d'] \wedge e \in \mathcal{E} \cup \mathcal{T} \Rightarrow w(d) > w(d')$

ASSUME: $[\beta, d] \xrightarrow{e} [\beta', d'] \wedge e \in \mathcal{E} \cup \mathcal{T}$

PROVE: $w(d) > w(d')$

PROOF: by induction on the structure of d

Induction start:

$\langle 2 \rangle 1$. CASE: $d = e$ (single event)

$\langle 3 \rangle 1$. $d' = \mathsf{skip}$

PROOF: Rules (8.3) and (8.10).

$\langle 3 \rangle 2$. $w(d) = 1 < 0 = w(d')$

PROOF: $\langle 2 \rangle 1$, $\langle 3 \rangle 1$ and def. of w.

$\langle 3 \rangle 3$. Q.E.D.

PROOF: $\langle 3 \rangle 2$.

Induction step:

ASSUME: $[\beta_k, d_k] \xrightarrow{e} [\beta_k, d'_k] \wedge e \in \mathcal{E} \cup \mathcal{T} \Rightarrow w(d_k) > w(d'_k),\ k \in \{1, 2\}$ (induction hypothesis)

$\langle 2 \rangle 2$. CASE: $d = d_1\ \mathsf{alt}\ d_2$ or $d = d_1\ \mathsf{xalt}\ d_2$

$\langle 3 \rangle 1$. $(d' = d_1 \vee d' = d_2) \wedge e \in \mathcal{T}$

PROOF: Rules (8.6), (8.16) and (8.17).

$\langle 3 \rangle 2$. CASE: $d' = d_1$

$\langle 4 \rangle 1$. $w(d) = \mathbf{max}(w(d_1), w(d_2)) + 1 > w(d_1) = w(d')$

PROOF: Def. of w and $\mathbf{max}(w(d_1), w(d_2)) \geq w(d_1)$.

$\langle 4 \rangle 2$. Q.E.D.

PROOF: $\langle 4 \rangle 1$.

$\langle 3 \rangle 3$. CASE: $d' = d_2$

$\langle 4 \rangle 1$. $w(d) = \mathbf{max}(w(d_1), w(d_2)) + 1 > w(d_2) = w(d')$

PROOF: Def. of w and $\mathbf{max}(w(d_1), w(d_2)) \geq w(d_2)$.

⟨4⟩2. Q.E.D.
PROOF: ⟨4⟩1.
⟨3⟩4. Q.E.D.
PROOF: ⟨3⟩1, ⟨3⟩2 and ⟨3⟩3.
⟨2⟩3. CASE: $d = \mathsf{refuse}\ d_1$ or $d = \mathsf{assert}\ d_1$
⟨3⟩1. $d' = d_1 \wedge e \in \mathcal{T}$
PROOF: Rules (8.6), (8.18), (8.19).
⟨3⟩2. $w(d) = w(d_1) + 1 > w(d_1) = w(d')$
PROOF: Def. of w.
⟨3⟩3. Q.E.D.
PROOF: ⟨3⟩2.
⟨2⟩4. CASE: $d = \mathsf{loop}\langle n \rangle\ d_1$
⟨3⟩1. $n > 0$
PROOF: Assumption and $n = 0 \Rightarrow d = \mathsf{skip}$ (see section 8.3.9).
⟨3⟩2. $d = d_1\ \mathsf{seq}\ \mathsf{loop}\langle n-1 \rangle\ d_1 \wedge e \in \mathcal{T}$
PROOF: ⟨3⟩1 and rules (8.6) and (8.20).
⟨3⟩3. $w(d) = n \cdot (w(d_1) + 2) = n \cdot w(d_1) + 2 \cdot n$
PROOF: Def. of w.
⟨3⟩4. $w(d') = w(d_1) + (n-1) \cdot (w(d_1) + 2) + 1$
$= w(d_1) + n \cdot w(d_1) + 2 \cdot n - w(d_1) - 2 + 1 = n \cdot w(d_1) + 2 \cdot n - 1$
PROOF: ⟨3⟩2 and def. of w.
⟨3⟩5. $w(d) = n \cdot w(d_1) + 2 \cdot n > n \cdot w(d_1) + 2 \cdot n - 1 = w(d')$
PROOF: ⟨3⟩3 and ⟨3⟩4.
⟨3⟩6. Q.E.D.
PROOF: ⟨3⟩5
⟨2⟩5. CASE: $d = d_1\ \mathsf{seq}\ d_2$
⟨3⟩1. $((d' = d_1'\ \mathsf{seq}\ d_2 \wedge [\beta_1, d_1] \xrightarrow{e} [\beta_1', d_1'])$
$\vee (d' = d_1\ \mathsf{seq}\ d_2' \wedge [\beta_2, d_2] \xrightarrow{e} [\beta_2', d_2']))$
$\wedge\, e \in \mathcal{E} \cup \mathcal{T}$
PROOF: By rules (8.3), (8.11) and (8.12) either d_1 or d_2 is executed (see also proof of lemma 7).
⟨3⟩2. CASE: $d' = d_1'\ \mathsf{seq}\ d_2 \wedge [\beta_1, d_1] \xrightarrow{e} [\beta_1', d_1'] \wedge\ e \in \mathcal{E} \cup \mathcal{T}$
⟨4⟩1. $w(d_1) > w(d_1')$
PROOF: Case assumption ⟨3⟩2 and induction hypothesis.
⟨4⟩2. $w(d) = w(d_1) + w(d_2) + 1 > w(d_1') + w(d_2) + 1 = w(d')$
PROOF: Case assumption ⟨2⟩5, case assumption ⟨3⟩2, ⟨4⟩1 and def. of w.
⟨4⟩3. Q.E.D.
PROOF: ⟨4⟩2
⟨3⟩3. CASE: $d' = d_1\ \mathsf{seq}\ d_2' \wedge [\beta_2, d_2] \xrightarrow{e} [\beta_2', d_2'] \wedge\ e \in \mathcal{E} \cup \mathcal{T}$
⟨4⟩1. $w(d_2) > w(d_2')$
PROOF: Case assumption ⟨3⟩3 and induction hypothesis.
⟨4⟩2. $w(d) = w(d_1) + w(d_2) + 1 > w(d_1) + w(d_2') + 1 = w(d')$
PROOF: Case assumption ⟨2⟩5, case assumption ⟨3⟩3, ⟨4⟩1 and def. of w.
⟨4⟩3. Q.E.D.
PROOF: ⟨4⟩2
⟨3⟩4. Q.E.D.
PROOF: ⟨3⟩1, ⟨3⟩2 and ⟨3⟩3.
⟨2⟩6. CASE: $d = d_1\ \mathsf{par}\ d_2$

⟨3⟩1. $((d' = d'_1 \text{ par } d_2 \wedge [\beta_1, d_1] \xrightarrow{e} [\beta'_1, d'_1])$
$\vee (d' = d_1 \text{ par } d'_2 \wedge [\beta_2, d_2] \xrightarrow{e} [\beta'_2, d'_2]))$
$\wedge e \in \mathcal{E} \cup \mathcal{T}$
PROOF: By rules (8.3), (8.13) and (8.14) either d_1 or d_2 is executed (see also proof of lemma 9).
⟨3⟩2. CASE: $d' = d'_1 \text{ par } d_2 \wedge [\beta_1, d_1] \xrightarrow{e} [\beta'_1, d'_1] \wedge e \in \mathcal{E} \cup \mathcal{T}$
⟨4⟩1. $w(d_1) > w(d'_1)$
PROOF: Case assumption ⟨3⟩2 and induction hypothesis.
⟨4⟩2. $w(d) = w(d_1) + w(d_2) + 1 > w(d'_1) + w(d_2) + 1 = w(d')$
PROOF: Case assumption ⟨2⟩6, case assumption ⟨3⟩2, ⟨4⟩1 and def. of w.
⟨4⟩3. Q.E.D.
PROOF: ⟨4⟩2
⟨3⟩3. CASE: $d' = d_1 \text{ par } d'_2 \wedge [\beta_2, d_2] \xrightarrow{e} [\beta'_2, d'_2] \wedge e \in \mathcal{E} \cup \mathcal{T}$
⟨4⟩1. $w(d_2) > w(d'_2)$
PROOF: Case assumption ⟨3⟩3 and induction hypothesis.
⟨4⟩2. $w(d) = w(d_1) + w(d_2) + 1 > w(d_1) + w(d'_2) + 1 = w(d')$
PROOF: Case assumption ⟨2⟩6, case assumption ⟨3⟩3, ⟨4⟩1 and def. of w.
⟨4⟩3. Q.E.D.
PROOF: ⟨4⟩2
⟨3⟩4. Q.E.D.
PROOF: ⟨3⟩1, ⟨3⟩2 and ⟨3⟩3.
⟨2⟩7. CASE: $d = d_1 \text{ strict } d_2$
⟨3⟩1. $d' = d'_1 \text{ strict } d_2 \wedge [\beta_1, d_1] \xrightarrow{e} [\beta'_1, d'_1]) \wedge e \in \mathcal{E} \cup \mathcal{T}$
PROOF: Rules (8.3) and (8.15).
⟨3⟩2. $w(d_1) > w(d'_1)$
PROOF: ⟨3⟩1 and induction hypothesis.
⟨3⟩3. $w(d) = w(d_1) + w(d_2) + 1 > w(d'_1) + w(d_2) + 1 = w(d')$
PROOF: Case assumption ⟨2⟩7, ⟨3⟩2, and def. of w.
⟨3⟩4. Q.E.D.
PROOF: ⟨3⟩3
⟨2⟩8. Q.E.D.
PROOF: ⟨2⟩1-⟨2⟩7.
⟨1⟩4. Q.E.D.
PROOF: ⟨1⟩1, ⟨1⟩2 and ⟨1⟩3.

□

In the previous section we assumed that diagrams do not have repetition of events. We make the same assumption for diagrams that are have high level operators. For the operators refuse, assert, alt and xalt this is straight forward by assuming that for a diagram d:

$$d = d_1 \text{ alt } d_2 \vee d = d_1 \text{ xalt } d_2 \Rightarrow ev.d_1 \cap ev.d_2 = \emptyset$$

However, the loop is a bit more problematic. Given a diagram d, assume that we rename every signal s in d and give it the new name s^k where $k \in \mathbb{N}$. Let d^k denote the resulting diagram. We now assume, both in the denotational and the operational semantics, that we have an implicit counter and a way of doing this renaming every time the body of a loop is "copied" out of the loop. A way to do this could be to

redefine the Π rule for loop as

$$\Pi(L, \beta, \mathsf{loop}^c\langle n\rangle\ d) \xrightarrow{\tau_{loop}} \Pi(L, \beta, d^c\ \mathsf{seq}\ \mathsf{loop}^{c+1}\langle n-1\rangle\ d) \tag{A.1}$$

and the definition of semantic loop to

$$\mu_n^c\ [\![\, d \,]\!] \stackrel{\mathsf{def}}{=} [\![\, d^c \,]\!] \gtrsim \mu_{n-1}^{c+1}\ [\![\, d \,]\!] \tag{A.2}$$

where $c \in \mathbb{N}$ is a counter. Further we could redefine $ev.\_$ and $msg.\_$ as follows

$$msg.\mathsf{loop}^c\langle n\rangle\ d \stackrel{\mathsf{def}}{=} msg.d^c \cup msg.\mathsf{loop}^{c+1}\langle n-1\rangle\ d$$

$$ev.\mathsf{loop}^c\langle n\rangle\ d \stackrel{\mathsf{def}}{=} ev.d^c \cup ev.\mathsf{loop}^{c+1}\langle n-1\rangle\ d$$

In the following we assume these new definitions, also when not writing it out explicitly.

Lemma 17 *Given a diagram d. For all traces t such that (there exists β such that)*

$$[env^!_{\mathcal{M}}.d, d] \xrightarrow{t} [\beta, \mathsf{skip}]$$

we may find a simple diagram d' (and β') such that

$$[env^!_{\mathcal{M}}.d', d'] \xrightarrow{\mathcal{E}\circledS t} [\beta', \mathsf{skip}]$$

Proof of lemma 17
ASSUME: $\exists \beta : [env^!_{\mathcal{M}}.d, d] \xrightarrow{t} [\beta, \mathsf{skip}]$
PROVE: $\exists d', \beta' : [env^!_{\mathcal{M}}.d', d'] \xrightarrow{\mathcal{E}\circledS t} [\beta', \mathsf{skip}]$, d' simple
PROOF: by induction on d.
$\langle 1\rangle 1$. Induction start: d simple
PROOF: Trivial by letting $d = d'$ and $\beta = \beta'$. By lemma 5, $\mathcal{E}\circledS t = t$.
Induction step:
ASSUME: $\forall d_k, t_k : \exists \beta_k : [env^!_{\mathcal{M}}.d_k, d_k] \xrightarrow{t_k} [\beta_k, \mathsf{skip}] \Rightarrow$
$\exists d'_k, \beta'_k : [env^!_{\mathcal{M}}.d'_k, d'_k] \xrightarrow{\mathcal{E}\circledS t_k} [\beta'_k, \mathsf{skip}]$, d'_k simple (induction hypothesis)
$\langle 1\rangle 2$. CASE: $d = d_1\ \mathsf{seq}\ d_2$ or $d = d_1\ \mathsf{par}\ d_2$ or $d = d_1\ \mathsf{strict}\ d_2$
PROOF: By rules (8.3), (8.11), (8.12), (8.13), (8.14) and (8.15), t is obtained by alternately executing d_1 and d_2 in some suitable fashion. This means there must exist traces t_1 and t_2 (and β_1 and β_2) such that $[env^!_{\mathcal{M}}.d_k, d_k] \xrightarrow{t_k} [\beta_k, \mathsf{skip}]$ for $k \in \{1, 2\}$ and t is (some kind of) merge of t_1 and t_2. By the induction hypothesis we may find d'_k such that it produces $\mathcal{E}\circledS t_k$ (for $k \in \{1, 2\}$). Let $d' = d'_1\ \mathsf{seq}\ d'_2$ or $d' = d'_1\ \mathsf{par}\ d'_2$ or $d' = d'_1\ \mathsf{strict}\ d'_2$. Then clearly d' is simple. Further, by applying the same execution scheme (i.e., alternation between the left and right hand side of the operator), except for the silent events in t, d' produces $\mathcal{E}\circledS t$.
$\langle 1\rangle 3$. CASE: $d = \mathsf{refuse}\ d_1$
PROOF: By assumption and rules (8.6) and (8.18), $t = \langle \tau_{refuse}\rangle^\frown t_1$ such that $[env^!_{\mathcal{M}}.\mathsf{refuse}\ d_1, d_1] \xrightarrow{t_1} [\beta, \mathsf{skip}]$. By $env^!_{\mathcal{M}}.\mathsf{refuse}\ d_1 = env^!_{\mathcal{M}}.d_1$ and the induction hypothesis, we may find simple d'_1 such that $[env^!_{\mathcal{M}}.d'_1, d'_1] \xrightarrow{\mathcal{E}\circledS t_1} [\beta'_1, \mathsf{skip}]$. Let $d' = d'_1$. Clearly d' is simple. Further, we have that $[env^!_{\mathcal{M}}.d', d'] \xrightarrow{\mathcal{E}\circledS t} [\beta'_1, \mathsf{skip}]$ since $\mathcal{E}\circledS\langle \tau_{refuse}\rangle^\frown t_1 = \mathcal{E}\circledS t_1$ (by def. of $\circledS$ and $\tau_{refuse} \notin \mathcal{E}$).

⟨1⟩4. CASE: $d = \mathsf{assert}\ d_1$

PROOF: By assumption and rules (8.6) and (8.19), $t = \langle \tau_{assert} \rangle^\frown t_1$ such that $[env^!_{\mathcal{M}}.\mathsf{assert}\ d_1, d_1] \xrightarrow{t_1} [\beta, \mathsf{skip}]$. By $env^!_{\mathcal{M}}.\mathsf{assert}\ d_1 = env^!_{\mathcal{M}}.d_1$ and the induction hypothesis, we may find simple d_1' such that $[env^!_{\mathcal{M}}.d_1', d_1'] \xrightarrow{\mathcal{E}\circledS t_1} [\beta_1', \mathsf{skip}]$. Let $d' = d_1'$. Clearly d' is simple. Further, we have that $[env^!_{\mathcal{M}}.d', d'] \xrightarrow{\mathcal{E}\circledS t} [\beta_1', \mathsf{skip}]$ since $\mathcal{E}\circledS\langle \tau_{assert} \rangle^\frown t_1 = \mathcal{E}\circledS t_1$ (by def. of $\circledS$ and $\tau_{assert} \notin \mathcal{E}$).

⟨1⟩5. CASE: $d = d_1\ \mathsf{alt}\ d_2$

PROOF: By assumption and rules (8.6) and (8.16), $t = \langle \tau_{alt} \rangle^\frown t_k$ such that $[env^!_{\mathcal{M}}.d, d_k] \xrightarrow{t_k} [\beta, \mathsf{skip}]$, for $k = 1$ or $k = 2$. Because $env^!_{\mathcal{M}}.d = env^!_{\mathcal{M}}.d_1 \cup env^!_{\mathcal{M}}.d_2$, $env^!_{\mathcal{M}}.d_1 \cap env^!_{\mathcal{M}}.d_2 = \emptyset$ and by the induction hypothesis we may find simple d_k' such that $[env^!_{\mathcal{M}}.d_k', d_k'] \xrightarrow{\mathcal{E}\circledS t_k} [\beta_k', \mathsf{skip}]$ for the appropriate choice of k. Let $d' = d_k'$ (for this choice of k). Clearly d' is simple. Further, we have that $[env^!_{\mathcal{M}}.d', d'] \xrightarrow{\mathcal{E}\circledS t} [\beta_k', \mathsf{skip}]$ since $\mathcal{E}\circledS t = \mathcal{E}\circledS\langle \tau_{alt} \rangle^\frown t_k = \mathcal{E}\circledS t_k$ (by def. of $\circledS$ and $\tau_{alt} \notin \mathcal{E}$).

⟨1⟩6. CASE: $d = d_1\ \mathsf{xalt}\ d_2$

PROOF: By assumption and rules (8.6) and (8.17), $t = \langle \tau_{xalt} \rangle^\frown t_k$ such that $[env^!_{\mathcal{M}}.d, d_k] \xrightarrow{t_k} [\beta, \mathsf{skip}]$, for $k = 1$ or $k = 2$. Because $env^!_{\mathcal{M}}.d = env^!_{\mathcal{M}}.d_1 \cup env^!_{\mathcal{M}}.d_2$, $env^!_{\mathcal{M}}.d_1 \cap env^!_{\mathcal{M}}.d_2 = \emptyset$ and by the induction hypothesis we may find simple d_k' such that $[env^!_{\mathcal{M}}.d_k', d_k'] \xrightarrow{\mathcal{E}\circledS t_k} [\beta_k', \mathsf{skip}]$ for the appropriate choice of k. Let $d' = d_k'$ (for this choice of k). Clearly d' is simple. Further, we have that $[env^!_{\mathcal{M}}.d', d'] \xrightarrow{\mathcal{E}\circledS t} [\beta_k', \mathsf{skip}]$ since $\mathcal{E}\circledS t = \mathcal{E}\circledS\langle \tau_{xalt} \rangle^\frown t_k = \mathcal{E}\circledS t_k$ (by def. of $\circledS$ and $\tau_{xalt} \notin \mathcal{E}$).

⟨1⟩7. CASE: $d = \mathsf{loop}\langle n \rangle\ d_1$

PROOF: by induction on n.

⟨2⟩1. Induction start: $n = 1$

PROOF: By assumption and rules (8.6) and (8.20), $t = \langle \tau_{loop} \rangle^\frown t_1$ such that $[env^!_{\mathcal{M}}.\mathsf{loop}\langle 1 \rangle\ d_1, \mathsf{loop}\langle 1 \rangle\ d_1] \xrightarrow{\tau_{loop}} [env^!_{\mathcal{M}}.\mathsf{loop}\langle 1 \rangle\ d_1, d_1\ \mathsf{seq}\ \mathsf{loop}\langle 0 \rangle\ d_1] = [env^!_{\mathcal{M}}.d_1 \cup env^!_{\mathcal{M}}.\mathsf{skip}, d_1\ \mathsf{seq}\ \mathsf{skip}] = [env^!_{\mathcal{M}}.d_1, d_1]$. By the induction hypothesis we may find simple d_1' such that $[env^!_{\mathcal{M}}.d_1', d_1'] \xrightarrow{\mathcal{E}\circledS t_1} [\beta_1', \mathsf{skip}]$. Let $d' = d_1'$. Clearly d' is simple. Further, we have that $[env^!_{\mathcal{M}}.d', d'] \xrightarrow{\mathcal{E}\circledS t}$ since $\mathcal{E}\circledS t = \mathcal{E}\circledS\langle \tau_{loop} \rangle^\frown t_1 = \mathcal{E}\circledS t_1$ (by def. of $\circledS$ and $\tau_{loop} \notin \mathcal{E}$).

⟨2⟩2. Induction step: $n = k + 1$

ASSUME: For all t and d_1, if these exists β such that $[env^!_{\mathcal{M}}.\mathsf{loop}\langle k \rangle\ d_1, \mathsf{loop}\langle k \rangle\ d_1] \xrightarrow{t} [\beta, \mathsf{skip}]$ then we may find simple d' (and β') such that $[env^!_{\mathcal{M}}.d', d'] \xrightarrow{\mathcal{E}\circledS t} [\beta', \mathsf{skip}]$ (induction hypothesis 2)

PROOF: By assumption and rules (8.6) and (8.20), $t = \langle \tau_{loop} \rangle^\frown t'$ such that $[env^!_{\mathcal{M}}.\mathsf{loop}\langle k+1 \rangle\ d_1, \mathsf{loop}\langle k+1 \rangle\ d_1] \xrightarrow{\tau_{loop}} [env^!_{\mathcal{M}}.\mathsf{loop}\langle k+1 \rangle\ d_1, d_1\ \mathsf{seq}\ \ \mathsf{loop}\langle k \rangle\ d_1] \xrightarrow{t'} [\beta, \mathsf{skip}]$. We have that $env^!_{\mathcal{M}}.\mathsf{loop}\langle k+1 \rangle\ d_1 = env^!_{\mathcal{M}}.d_1 \cup env^!_{\mathcal{M}}.\mathsf{loop}\langle k \rangle\ d_1 = env^!_{\mathcal{M}}.d_1\ \mathsf{seq}\ \mathsf{loop}\langle k \rangle\ d_1$. By this, the induction hypotheses and ⟨1⟩2 we may find simple d'' such that $[env^!_{\mathcal{M}}.d'', d''] \xrightarrow{\mathcal{E}\circledS t'} [\beta', \mathsf{skip}]$. Let $d' = d''$. Clearly d' is simple. Further, we have that $[env^!_{\mathcal{M}}.d', d'] \xrightarrow{\mathcal{E}\circledS t}$ since $\mathcal{E}\circledS t = \mathcal{E}\circledS\langle \tau_{loop} \rangle^\frown t' = \mathcal{E}\circledS t'$ (by def. of $\circledS$ and $\tau_{loop} \notin \mathcal{E}$).

⟨2⟩3. Q.E.D.

PROOF: ⟨2⟩1 and ⟨2⟩2.

⟨1⟩8. Q.E.D.
PROOF: ⟨1⟩1-⟨1⟩7.

□

Lemma 18 *Given a diagram d. For all traces t such that*

$$t \in [\![d]\!]$$

we may find a simple diagram d' such that

$$t \in [\![d']\!]$$

Proof of lemma 18
ASSUME: $t \in [\![d]\!]$
PROVE: $\exists d' : t \in [\![d']\!]$, d' simple.
PROOF: by induction on d.
⟨1⟩1. Induction start: d simple
PROOF: Trivial: Let $d = d'$.
Induction step:
ASSUME: If $t_k \in [\![d_k]\!]$ then there exists simple d'_k such that $t_k \in [\![d'_k]\!]$ (induction hypothesis)
⟨1⟩2. $d = d_1$ seq d_2 or $d = d_1$ par d_2 or $d = d_1$ strict d_2
PROOF: By definitions (5.5)-(5.6), (5.8)-(5.9), and (5.11)-(5.12) there exist traces t_1 and t_2 such that t is obtained from t_1 and t_2. By the induction hypothesis we may find simple d'_1 and d'_2 such that $t_1 \in [\![d'_1]\!]$ and $t_2 \in [\![d'_2]\!]$. Let $d' = d'_1$ seq d'_2 or $d' = d'_1$ par d'_2 or $d' = d'_1$ strict d'_2. Clearly d' is simple and $t \in [\![d']\!]$.
⟨1⟩3. $d =$ refuse d_1
PROOF: By def. $[\![d_1]\!] = \{(p_1, n_1), \ldots, (p_m, n_m)\}$ iff $[\![d]\!] = \{(\emptyset, p_1 \cup n_1), \ldots, (\emptyset, p_m \cup n_m)\}$, so $t \in [\![d]\!]$ iff $t \in p_i$ or $t \in n_i$ (for some $i \in \{1, \ldots, m\}$) iff $t \in [\![d_1]\!]$. By induction hypothesis there exists simple d'_1 such that $t \in [\![d'_1]\!]$. Let $d' = d'_1$. Clearly d' is simple and $t \in [\![d']\!]$.
⟨1⟩4. $d =$ assert d_1
PROOF: We have assumed that $[\![$ assert $d_1]\!] = [\![d_1]\!]$ so $t \in [\![d]\!]$ iff $t \in [\![d_1]\!]$. By induction hypothesis there exists simple d'_1 such that $t \in [\![d'_1]\!]$. Let $d' = d'_1$. Clearly d' is simple and $t \in [\![d']\!]$.
⟨1⟩5. $d = d_1$ alt d_2 or $d = d_1$ xalt d_2
PROOF: By def. $t \in [\![d_1$ alt $d_2]\!]$ or $t \in [\![d_1$ xalt $d_2]\!]$ iff $t \in [\![d_1]\!] \uplus [\![d_2]\!]$ or $t \in [\![d_1]\!] \cup [\![d_2]\!]$ iff $t \in [\![d_1]\!]$ or $t \in [\![d_2]\!]$. Chose the appropriate $k \in \{1, 2\}$. By induction hypothesis there exist simple d'_k such that $t \in [\![d'_k]\!]$. Let $d' = d'_k$. Clearly d' is simple and $t \in [\![d']\!]$.
⟨1⟩6. $d =$ loop$\langle n \rangle$ d_1
PROOF: By def. $[\![d]\!] = [\![$ loop$\langle n \rangle$ $d_1]\!] = \mu_n [\![d_1]\!] = [\![d_1^1]\!] \succsim [\![d_1^2]\!] \succsim \cdots \succsim [\![d_1^n]\!]$. By this and defs. (5.5)-(5.6) we have that $[\![d]\!] = [\![d_1^1$ seq d_1^2 seq $\cdots$ seq $d_1^n]\!]$. By the induction hypothesis and ⟨1⟩2 we may find simple $d_1^{1\prime}, d_2^{2\prime}, \ldots, d_2^{n\prime}$ such that $t \in [\![d_1^{1\prime}$ seq $d_1^{2\prime}$ seq $\cdots$ seq $d_1^{n\prime}]\!]$. Let $d' = d_1^{1\prime}$ seq $d_1^{2\prime}$ seq $\cdots$ seq $d_1^{n\prime}$. Clearly d' is simple and $t \in [\![d']\!]$.

⟨1⟩7. Q.E.D.
PROOF: ⟨1⟩1-⟨1⟩6.

□

Theorem 4 (Soundness) *Given diagram $d \in \mathcal{D}$ without infinite loops. For all traces $t \in (\mathcal{E} \cup \mathcal{T})^*$, if there exists $\beta \in \mathcal{B}$ such that*

$$[env^!_{\mathcal{M}}.d, d] \xrightarrow{t} [\beta, \mathsf{skip}]$$

then

$$\mathcal{E} \circledS t \in [\![d]\!]$$

Proof of theorem 4
ASSUME: $\exists \beta : [env^!_{\mathcal{M}}.d, d] \xrightarrow{t} [\beta, \mathsf{skip}]$
PROVE: $\mathcal{E} \circledS t \in [\![d]\!]$
PROOF: by induction on d.
⟨1⟩1. Induction start: $d = \mathsf{skip},\ d = (!, m),\ d = (?, m)$
PROOF: This carries over from theorem 1.
Induction step:
ASSUME: $\exists \beta_k : [env^!_{\mathcal{M}}.d_k, d_k] \xrightarrow{t_k} [\beta_k, \mathsf{skip}] \Rightarrow \mathcal{E} \circledS t_k \in [\![d]\!]$
⟨1⟩2. CASE: $d = d_1$ seq d_2 or $d = d_1$ par d_2 or $d = d_1$ strict d_2
PROOF: By lemmas 17 and 18 this follows from theorem 1
⟨1⟩3. CASE: $d =$ refuse d_1
⟨2⟩1. $t = \langle \tau_{refuse} \rangle^\frown t_1$ such that $[env^!_{\mathcal{M}}.d, d] \xrightarrow{\tau_{refuse}} [env^!_{\mathcal{M}}.d, d_1] \xrightarrow{t_1} [\beta, \mathsf{skip}]$
PROOF: Assumptions and rules (8.6) and (8.18).
⟨2⟩2. $\mathcal{E} \circledS t = \mathcal{E} \circledS \langle \tau_{refuse} \rangle^\frown t_1 = \mathcal{E} \circledS t_1$
PROOF: ⟨2⟩1, def. of $\circledS$ and $\tau_{refuse} \notin \mathcal{E}$.
⟨2⟩3. $\mathcal{E} \circledS t \in [\![d_1]\!]$
PROOF: ⟨2⟩2 and induction hypothesis (by $env^!_{\mathcal{M}}.\mathsf{refuse}\ d_1 = env^!_{\mathcal{M}}.d_1$).
⟨2⟩4. $\mathcal{E} \circledS t \in [\![\mathsf{refuse}\ d_1]\!]$
PROOF: Def. (5.19) of the denotation of refuse.
⟨2⟩5. Q.E.D.
PROOF: ⟨2⟩4.
⟨1⟩4. CASE: $d =$ assert d_1
⟨2⟩1. $t = \langle \tau_{assert} \rangle^\frown t_1$ such that $[env^!_{\mathcal{M}}.d, d] \xrightarrow{\tau_{assert}} [env^!_{\mathcal{M}}.d, d_1] \xrightarrow{t_1} [\beta, \mathsf{skip}]$
PROOF: Assumptions and rules (8.6) and (8.19).
⟨2⟩2. $\mathcal{E} \circledS t = \mathcal{E} \circledS \langle \tau_{assert} \rangle^\frown t_1 = \mathcal{E} \circledS t_1$
PROOF: ⟨2⟩1, def. of $\circledS$ and $\tau_{assert} \notin \mathcal{E}$.
⟨2⟩3. $\mathcal{E} \circledS t \in [\![d_1]\!]$
PROOF: ⟨2⟩2 and induction hypothesis (by $env^!_{\mathcal{M}}.\mathsf{assert}\ d_1 = env^!_{\mathcal{M}}.d_1$).
⟨2⟩4. $\mathcal{E} \circledS t \in [\![\mathsf{assert}\ d_1]\!]$
PROOF: Assumption that $[\![\mathsf{assert}\ d_1]\!] = [\![d_1]\!]$.
⟨2⟩5. Q.E.D.
PROOF: ⟨2⟩4.
⟨1⟩5. CASE: $d = d_1$ alt d_2
⟨2⟩1. $t = \langle \tau_{alt} \rangle^\frown t_k$ such that $[env^!_{\mathcal{M}}.d, d] \xrightarrow{\tau_{alt}} [env^!_{\mathcal{M}}.d, d_k] \xrightarrow{t_k} [\beta, \mathsf{skip}]$ for $k = 1$ or $k = 2$

PROOF: Assumptions and rules (8.6) and (8.16).

⟨2⟩2. $\mathcal{E}\circledS t = \mathcal{E}\circledS\langle\tau_{alt}\rangle^\frown t_k = \mathcal{E}\circledS t_k$
PROOF: ⟨2⟩1, def. of $\circledS$ and $\tau_{alt} \notin \mathcal{E}$.

⟨2⟩3. $\mathcal{E}\circledS t \in [\![d_k]\!]$ for $k = 1$ or $k = 2$
PROOF: ⟨2⟩2 and induction hypothesis (by $env^!_{\mathcal{M}}.d = env^!_{\mathcal{M}}.d_1 \cup env^!_{\mathcal{M}}.d_2$, $env^!_{\mathcal{M}}.d_1 \cap env^!_{\mathcal{M}}.d_2 = \emptyset$).

⟨2⟩4. $\mathcal{E}\circledS t \in [\![d_1 \ \mathsf{alt}\ d_2]\!]$
PROOF: ⟨2⟩3 and def. (5.14).

⟨2⟩5. Q.E.D.
PROOF: ⟨2⟩4.

⟨1⟩6. CASE: $d = d_1 \ \mathsf{xalt}\ d_2$

⟨2⟩1. $t = \langle\tau_{xalt}\rangle^\frown t_k$ such that $[env^!_{\mathcal{M}}.d, d] \xrightarrow{\tau_{xalt}} [env^!_{\mathcal{M}}.d, d_k] \xrightarrow{t_k} [\beta, \mathsf{skip}]$ for $k = 1$ or $k = 2$
PROOF: Assumptions and rules (8.6) and (8.17).

⟨2⟩2. $\mathcal{E}\circledS t = \mathcal{E}\circledS\langle\tau_{xalt}\rangle^\frown t_k = \mathcal{E}\circledS t_k$
PROOF: ⟨2⟩1, def. of $\circledS$ and $\tau_{xalt} \notin \mathcal{E}$.

⟨2⟩3. $\mathcal{E}\circledS t \in [\![d_k]\!]$ for $k = 1$ or $k = 2$
PROOF: ⟨2⟩2 and induction hypothesis (by $env^!_{\mathcal{M}}.d = env^!_{\mathcal{M}}.d_1 \cup env^!_{\mathcal{M}}.d_2$, $env^!_{\mathcal{M}}.d_1 \cap env^!_{\mathcal{M}}.d_2 = \emptyset$).

⟨2⟩4. $\mathcal{E}\circledS t \in [\![d_1 \ \mathsf{xalt}\ d_2]\!]$
PROOF: ⟨2⟩3 and def. (5.18).

⟨2⟩5. Q.E.D.
PROOF: ⟨2⟩4.

⟨1⟩7. CASE: $d = \mathsf{loop}\langle n\rangle\ d_1$
PROOF: Induction on n

⟨2⟩1. Induction start: $n = 1$

⟨3⟩1. $t = \langle\tau_{loop}\rangle^\frown t_1$ such that $[env^!_{\mathcal{M}}.\mathsf{loop}\langle 1\rangle\ d_1, \mathsf{loop}\langle 1\rangle\ d_1] \xrightarrow{\tau_{loop}}$
$[env^!_{\mathcal{M}}.\mathsf{loop}\langle 1\rangle\ d_1, d_1\ \mathsf{seq}\ \mathsf{loop}\langle 0\rangle\ d_1] \xrightarrow{t_1} [\beta, \mathsf{skip}]$
PROOF: Assumptions and rules (8.6) and (8.20).

⟨3⟩2. $\mathcal{E}\circledS t_1 \in [\![d_1]\!]$
PROOF: ⟨3⟩1 and induction hypothesis (by $d_1\ \mathsf{seq}\ \mathsf{loop}\langle 0\rangle\ d_1 = d_1\ \mathsf{seq}\ \ \mathsf{skip} = d_1$ and $env^!_{\mathcal{M}}.\mathsf{loop}\langle 1\rangle\ d_1 = env^!_{\mathcal{M}}.d_1 \cup env^!_{\mathcal{M}}.\mathsf{skip} = env^!_{\mathcal{M}}.d_1$).

⟨3⟩3. $\mathcal{E}\circledS t = \mathcal{E}\circledS\langle\tau_{loop}\rangle^\frown t_1 = \mathcal{E}\circledS t_1$
PROOF: ⟨3⟩1, def. of $\circledS$ and $\tau_{loop} \notin \mathcal{E}$.

⟨3⟩4. $[\![\mathsf{loop}\langle 1\rangle\ d_1]\!] = \mu_1[\![d_1]\!] = [\![d_1]\!]$
PROOF: Defs. (5.22), (5.23).

⟨3⟩5. $\mathcal{E}\circledS t = \mathcal{E}\circledS t_1 \in [\![d_1]\!] = [\![\mathsf{loop}\langle 1\rangle\ d_1]\!] = [\![d]\!]$
PROOF: ⟨3⟩2, ⟨3⟩3 and ⟨3⟩4.

⟨3⟩6. Q.E.D.
PROOF: ⟨3⟩5

⟨2⟩2. Induction step: $n = k + 1$
ASSUME: $\forall d_1, s\ :\ \exists\beta\ :\ [env^!_{\mathcal{M}}.\mathsf{loop}\langle k\rangle\ d_1, \mathsf{loop}\langle k\rangle\ d_1] \xrightarrow{s} [\beta, \mathsf{skip}] \Rightarrow \mathcal{E}\circledS s \in [\![\mathsf{loop}\langle k\rangle\ ]\!]$ (induction hypothesis 2)

⟨3⟩1. $t = \langle\tau_{loop}\rangle^\frown t'$ such that $[env^!_{\mathcal{M}}.\mathsf{loop}\langle k+1\rangle\ d_1, \mathsf{loop}\langle k+1\rangle\ d_1] \xrightarrow{\tau_{loop}}$
$[env^!_{\mathcal{M}}.\mathsf{loop}\langle k+1\rangle\ , d_1\ \mathsf{seq}\ \mathsf{loop}\langle k\rangle\ d_1] \xrightarrow{t'} [\beta, \mathsf{skip}]$
PROOF: Assumptions and rules (8.6) and (8.20).

⟨3⟩2. $\mathcal{E}\circledS t' \in [\![\, d_1 \;\mathsf{seq}\; \mathsf{loop}\langle k\rangle\; d_1 \,]\!]$
PROOF: ⟨3⟩1, induction hypotheses (by $env^!_{\mathcal{M}}.\mathsf{loop}\langle k+1\rangle\; d_1 = env^!_{\mathcal{M}}.d_1 \cup env^!_{\mathcal{M}}.\mathsf{loop}\langle k\rangle\; d_1 = env^!_{\mathcal{M}}.d_1 \;\mathsf{seq}\; \mathsf{loop}\langle k\rangle\; d_1$) and ⟨1⟩2.
⟨3⟩3. $\mathcal{E}\circledS t = \mathcal{E}\circledS\langle\tau_{loop}\rangle^\frown t' = \mathcal{E}\circledS t'$
PROOF: ⟨3⟩1, def. of $\circledS$ and $\tau_{loop} \notin \mathcal{E}$.
⟨3⟩4. $[\![\, \mathsf{loop}\langle k+1\rangle\; d_1 \,]\!] = \mu_{k+1}[\![\, d_1 \,]\!] = [\![\, d_1 \,]\!] \succsim \mu_k[\![\, d_1 \,]\!]$
$= [\![\, d_1 \,]\!] \succsim [\![\, \mathsf{loop}\langle k\rangle\; d_1 \,]\!] = [\![\, d_1 \;\mathsf{seq}\; \mathsf{loop}\langle k\rangle\; d_1 \,]\!]$
PROOF: Defs. (5.5), (5.7), (5.6), (5.22), (5.23).
⟨3⟩5. $\mathcal{E}\circledS t = \mathcal{E}\circledS t' \in [\![\, d_1 \;\mathsf{seq}\; \mathsf{loop}\langle k\rangle\; d_1 \,]\!] = [\![\, \mathsf{loop}\langle k+1\rangle\; d_1 \,]\!] = [\![\, d \,]\!]$
PROOF: ⟨3⟩2, ⟨3⟩3 and ⟨3⟩4.
⟨3⟩6. Q.E.D.
PROOF: ⟨3⟩5.
⟨2⟩3. Q.E.D.
PROOF: ⟨2⟩1 and ⟨2⟩2.
⟨1⟩8. Q.E.D.
PROOF: ⟨1⟩1-⟨1⟩7.

□

Theorem 5 (Completeness) *Given a diagram $d \in \mathcal{D}$ without infinite loops. For all traces $t \in \mathcal{E}^*$, if*

$$t \in [\![\, d \,]\!]$$

then there exist trace $t' \in (\mathcal{E} \cup \mathcal{T})^*$ and $\beta \in \mathcal{B}$ such that

$$[env^!_{\mathcal{M}}.d, d] \xrightarrow{t'} [\beta, \mathsf{skip}] \text{ and } \mathcal{E}\circledS t' = t$$

Proof of theorem 5
ASSUME: $t \in [\![\, d \,]\!]$
PROVE: $\exists t', \beta : [env^!_{\mathcal{M}}.d, d] \xrightarrow{t'} [\beta, \mathsf{skip}] \wedge \mathcal{E}\circledS t' = t$
PROOF: by induction on d.
⟨1⟩1. Induction start: $d = \mathsf{skip},\ d = (!, m),\ d = (?, m)$
PROOF: This carries over from theorem 2.
Induction step:
ASSUME: $t_k \in [\![\, d_k \,]\!] \Rightarrow \exists t'_k, \beta_k : [env^!_{\mathcal{M}}.d_k, d_k] \xrightarrow{t'_k} [\beta_k, \mathsf{skip}] \wedge \mathcal{E}\circledS t'_k = t_k$ (induction hypothesis)
⟨1⟩2. CASE: $d = d_1 \;\mathsf{seq}\; d_2$ or $d = d_1 \;\mathsf{par}\; d_2$ or $d = d_1 \;\mathsf{strict}\; d_2$
PROOF: By 17 and 18 this follows from theorem 2.
⟨1⟩3. $d = \mathsf{refuse}\; d_1$
⟨2⟩1. $t \in [\![\, d_1 \,]\!]$
PROOF: Assumption and def. (5.19).
⟨2⟩2. $\exists t'_1, \beta_1 : [env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t'_1} [\beta_1, \mathsf{skip}] \ \wedge\ \mathcal{E}\circledS t'_1 = t$
PROOF: ⟨2⟩1 and induction hypothesis.
⟨2⟩3. LET: $t' = \langle\tau_{refuse}\rangle^\frown t'_1$
⟨2⟩4. $[env^!_{\mathcal{M}}.d, d] \xrightarrow{\tau_{refuse}} [env^!_{\mathcal{M}}.d, d_1] \xrightarrow{t'_1} [\beta_1, \mathsf{skip}]$
PROOF: ⟨2⟩2, ⟨2⟩3, rules (8.6) and (8.18) and $env^!_{\mathcal{M}}.d = env^!_{\mathcal{M}}.\mathsf{refuse}\; d_1 = env^!_{\mathcal{M}}.d_1$.
⟨2⟩5. $\mathcal{E}\circledS t' = \mathcal{E}\circledS\langle\tau_{refuse}\rangle^\frown t'_1 = \mathcal{E}\circledS t'_1 = t$

PROOF: $\langle 2\rangle 2$, $\langle 2\rangle 3$, def. of $\circledS$ and $\tau_{refuse} \notin \mathcal{E}$.

$\langle 2\rangle 6$. Q.E.D.

PROOF: $\langle 2\rangle 3$, $\langle 2\rangle 4$ and $\langle 2\rangle 5$.

$\langle 1\rangle 4$. $d = \mathsf{assert}\ d_1$

$\langle 2\rangle 1$. $t \in [\![d_1]\!]$

PROOF: Assumption that $[\![\mathsf{assert}\ d_1]\!] = [\![d_1]\!]$.

$\langle 2\rangle 2$. $\exists t_1', \beta_1 : [env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1'} [\beta_1, \mathsf{skip}] \ \wedge\ \mathcal{E} \circledS t_1' = t$

PROOF: $\langle 2\rangle 1$ and induction hypothesis.

$\langle 2\rangle 3$. LET: $t' = \langle \tau_{assert} \rangle^\frown t_1'$

$\langle 2\rangle 4$. $[env^!_{\mathcal{M}}.d, d] \xrightarrow{\tau_{assert}} [env^!_{\mathcal{M}}.d, d_1] \xrightarrow{t_1'} [\beta_1, \mathsf{skip}]$

PROOF: $\langle 2\rangle 2$, $\langle 2\rangle 3$, rules (8.6) and (8.19) and $env^!_{\mathcal{M}}.d = env^!_{\mathcal{M}}.\mathsf{assert}\ d_1 = env^!_{\mathcal{M}}.d_1$.

$\langle 2\rangle 5$. $\mathcal{E} \circledS t' = \mathcal{E} \circledS \langle \tau_{assert} \rangle^\frown t_1' = \mathcal{E} \circledS t_1' = t$

PROOF: $\langle 2\rangle 2$, $\langle 2\rangle 3$, def. of $\circledS$ and $\tau_{assert} \notin \mathcal{E}$.

$\langle 2\rangle 6$. Q.E.D.

PROOF: $\langle 2\rangle 3$, $\langle 2\rangle 4$ and $\langle 2\rangle 5$.

$\langle 1\rangle 5$. $d = d_1\ \mathsf{alt}\ d_2$

$\langle 2\rangle 1$. $t \in [\![d_1]\!] \vee t \in [\![d_2]\!]$

PROOF: $\langle 1\rangle 5$ and def. (5.14).

$\langle 2\rangle 2$. $\exists t_k', \beta_k : [env^!_{\mathcal{M}}.d_k, d_k] \xrightarrow{t_k'} [\beta_k, \mathsf{skip}] \ \wedge\ \mathcal{E} \circledS t_k' = t$ for $k = 1$ or $k = 2$

PROOF: $\langle 2\rangle 1$ and induction hypothesis.

$\langle 2\rangle 3$. $[env^!_{\mathcal{M}}.d, d] \xrightarrow{\tau_{alt}} [env^!_{\mathcal{M}}.d, d_k] \xrightarrow{t_k'} [\beta_k, \mathsf{skip}]$ for $k = 1$ or $k = 2$

PROOF: $\langle 2\rangle 2$, rules (8.6) and (8.16), and $env^!_{\mathcal{M}}.d = env^!_{\mathcal{M}}.d_1 \cup env^!_{\mathcal{M}}.d_2$, $env^!_{\mathcal{M}}.d_1 \cap env^!_{\mathcal{M}}.d_2 = \emptyset$.

$\langle 2\rangle 4$. LET: $t' = \langle \tau_{alt} \rangle^\frown t_k'$ for the appropriate choice of $k \in \{1, 2\}$

$\langle 2\rangle 5$. $\mathcal{E} \circledS t' = \mathcal{E} \circledS \langle \tau_{alt} \rangle^\frown t_k' = \mathcal{E} \circledS t_k' = t$

PROOF: $\langle 2\rangle 2$, $\langle 2\rangle 4$, def. of $\circledS$ and $\tau_{alt} \notin \mathcal{E}$.

$\langle 2\rangle 6$. Q.E.D.

PROOF: $\langle 2\rangle 3$, $\langle 2\rangle 4$ and $\langle 2\rangle 5$.

$\langle 1\rangle 6$. $d = d_1\ \mathsf{xalt}\ d_2$

$\langle 2\rangle 1$. $t \in [\![d_1]\!] \vee t \in [\![d_2]\!]$

PROOF: $\langle 1\rangle 6$ and def. (5.18).

$\langle 2\rangle 2$. $\exists t_k', \beta_k : [env^!_{\mathcal{M}}.d_k, d_k] \xrightarrow{t_k'} [\beta_k, \mathsf{skip}] \ \wedge\ \mathcal{E} \circledS t_k' = t$ for $k = 1$ or $k = 2$

PROOF: $\langle 2\rangle 1$ and induction hypothesis.

$\langle 2\rangle 3$. $[env^!_{\mathcal{M}}.d, d] \xrightarrow{\tau_{xalt}} [env^!_{\mathcal{M}}.d, d_k] \xrightarrow{t_k'} [\beta_k, \mathsf{skip}]$ for $k = 1$ or $k = 2$

PROOF: $\langle 2\rangle 2$, rules (8.6) and (8.17), and $env^!_{\mathcal{M}}.d = env^!_{\mathcal{M}}.d_1 \cup env^!_{\mathcal{M}}.d_2$, $env^!_{\mathcal{M}}.d_1 \cap env^!_{\mathcal{M}}.d_2 = \emptyset$.

$\langle 2\rangle 4$. LET: $t' = \langle \tau_{xalt} \rangle^\frown t_k'$ for the appropriate choice of $k \in \{1, 2\}$

$\langle 2\rangle 5$. $\mathcal{E} \circledS t' = \mathcal{E} \circledS \langle \tau_{xalt} \rangle^\frown t_k' = \mathcal{E} \circledS t_k' = t$

PROOF: $\langle 2\rangle 2$, $\langle 2\rangle 4$, def. of $\circledS$ and $\tau_{xalt} \notin \mathcal{E}$.

$\langle 2\rangle 6$. Q.E.D.

PROOF: $\langle 2\rangle 3$, $\langle 2\rangle 4$ and $\langle 2\rangle 5$.

$\langle 1\rangle 7$. $d = \mathsf{loop}\langle n \rangle\ d_1$

PROOF: by induction on n

$\langle 2\rangle 1$. Induction start: $n = 1$

⟨3⟩1. $t \in [\![\mathsf{loop}\langle 1\rangle\ d_1]\!] = \mu_1[\![d_1]\!] = [\![d_1]\!]$
PROOF: ⟨1⟩7, ⟨2⟩1 and defs. (5.22), (5.23).

⟨3⟩2. $\exists t_1', \beta_1 : [env^!_{\mathcal{M}}.d_1, d_1] \xrightarrow{t_1'} [\beta_1, \mathsf{skip}] \ \wedge\ \mathcal{E} \circledS t_1' = t$
PROOF: ⟨3⟩1 and induction hypothesis.

⟨3⟩3. $[env^!_{\mathcal{M}}.\mathsf{loop}\langle 1\rangle\ d_1, \mathsf{loop}\langle 1\rangle\ d_1] \xrightarrow{\tau_{loop}}$
$[env^!_{\mathcal{M}}.\mathsf{loop}\langle 1\rangle\ d_1, d_1\ \mathsf{seq}\ \mathsf{loop}\langle 0\rangle\ d_1] \xrightarrow{t_1'} [\beta_1, \mathsf{skip}]$
PROOF: ⟨3⟩2 and $d_1\ \mathsf{seq}\ \mathsf{loop}\langle 0\rangle\ d_1 = d_1\ \mathsf{seq}\ \mathsf{skip} = d_1$, $env^!_{\mathcal{M}}.\mathsf{loop}\langle 1\rangle\ d_1 = env^!_{\mathcal{M}}.d_1 \cup env^!_{\mathcal{M}}.\mathsf{skip} = env^!_{\mathcal{M}}.d_1$.

⟨3⟩4. LET: $t' = \langle \tau_{loop} \rangle^\frown t_1'$

⟨3⟩5. $\mathcal{E} \circledS t' = \mathcal{E} \circledS \langle \tau_{loop} \rangle^\frown t_1' = \mathcal{E} \circledS t_1' = t$
PROOF: ⟨3⟩2, ⟨3⟩4, def. of $\circledS$ and $\tau_{loop} \notin \mathcal{E}$.

⟨3⟩6. Q.E.D.
PROOF: ⟨3⟩3, ⟨3⟩4 and ⟨3⟩5.

⟨2⟩2. Induction step: $n = k + 1$
ASSUME: $s \in [\![\mathsf{loop}\langle k\rangle\ d_1]\!] \Rightarrow \exists s', \beta' : [env^!_{\mathcal{M}}.\mathsf{loop}\langle k\rangle\ d_1, \mathsf{loop}\langle k\rangle\ d_1] \xrightarrow{s'} [\beta', \mathsf{skip}] \wedge \mathcal{E} \circledS s' = s$ (induction hypothesis 2)

⟨3⟩1. $t \in [\![\mathsf{loop}\langle k+1\rangle\ d_1]\!] = \mu_{k+1}[\![d_1]\!] = [\![d_1]\!] \gtrsim \mu_k[\![d_1]\!]$
$= [\![d_1]\!] \gtrsim [\![\mathsf{loop}\langle k\rangle\ d_1]\!] = [\![d_1\ \mathsf{seq}\ \mathsf{loop}\langle k\rangle\ d_1]\!]$
PROOF: Assumption, ⟨2⟩2 and definitions (5.23), (5.22), (5.5), (5.7) and (5.6).

⟨3⟩2. $\exists t'', \beta' : [env^!_{\mathcal{M}}.(d_1\ \mathsf{seq}\ \mathsf{loop}\langle k\rangle\ d_1), d_1\ \mathsf{seq}\ \mathsf{loop}\langle k\rangle\ d_1] \xrightarrow{t''} [\beta', \mathsf{skip}] \wedge t = \mathcal{E} \circledS t''$
PROOF: ⟨1⟩2, ⟨3⟩1 and induction hypotheses.

⟨3⟩3. $[env^!_{\mathcal{M}}.\mathsf{loop}\langle k+1\rangle\ d_1, \mathsf{loop}\langle k+1\rangle\ d_1] \xrightarrow{\tau_{loop}}$
$[env^!_{\mathcal{M}}.\mathsf{loop}\langle k+1\rangle\ d_1, d_1\ \mathsf{seq}\ \mathsf{loop}\langle k\rangle\ d_1] \xrightarrow{t''} [\beta', \mathsf{skip}]$
PROOF: ⟨3⟩2, rules (8.6) and (8.20), and $env^!_{\mathcal{M}}.(d_1\ \mathsf{seq}\ \mathsf{loop}\langle k\rangle\ d_1) = env^!_{\mathcal{M}}.d_1 \cup env^!_{\mathcal{M}}.\mathsf{loop}\langle k\rangle\ d_1 = env^!_{\mathcal{M}}.\mathsf{loop}\langle k+1\rangle\ d_1$.

⟨3⟩4. LET: $t' = \langle \tau_{loop} \rangle^\frown t''$

⟨3⟩5. $\mathcal{E} \circledS t' = \mathcal{E} \circledS \langle \tau_{loop} \rangle^\frown t'' = \mathcal{E} \circledS t'' = t$
PROOF: ⟨3⟩2, ⟨3⟩4, def. of $\circledS$ and $\tau_{loop} \notin \mathcal{E}$.

⟨3⟩6. Q.E.D.
PROOF: ⟨3⟩3, ⟨3⟩4 and ⟨3⟩5.

⟨2⟩3. Q.E.D.
PROOF: ⟨2⟩1 and ⟨2⟩2.

⟨1⟩8. Q.E.D.
PROOF: ⟨1⟩1-⟨1⟩7.

□

A.4 Sequence diagrams with infinite loops

In this section we prove soundness and completeness of diagrams that also contain the infinite loop, $\mathsf{loop}\langle\infty\rangle\ d$. As in the previous sections we treat the denotation $[\![d]\!]$ of a diagram d as a flat set, i.e. we write $t \in [\![d]\!]$ for $t \in \bigcup_{(p,n)\in[\![d]\!]}(p \cup n)$. This means we disregard negative behavior and interaction obligations. For simplicity we assume that $env.d = \emptyset$.

We let d^k for $k \in \mathbb{N}$ denote the diagram d with every signal s renamed to s^k and apply the loop with implicit counter $\mathsf{loop}^c\langle n\rangle\ d$ as described by rule (A.1) and definition (A.2). Further we define:

$$\begin{array}{rcl} [\![\, d \,]\!]^{\leq 1} & \stackrel{\mathsf{def}}{=} & [\![\, d^1 \,]\!] \\ [\![\, d \,]\!]^{\leq n+1} & \stackrel{\mathsf{def}}{=} & [\![\, d \,]\!]^{\leq n} \gtrsim [\![\, d^{n+1} \,]\!] \textbf{ for } n \geq 1 \end{array} \tag{A.3}$$

Let $\mathcal{U} = \mathbb{P}(\mathcal{H})$ be the powerset of $\mathcal{H}$. We have that $[\![\, d \,]\!] \in \mathcal{U}$ and define the chains of trace sets of $[\![\, d \,]\!]$ as

$$chains([\![\, d \,]\!]) \stackrel{\mathsf{def}}{=} \{\bar{u} \in \mathcal{U}^\infty \mid \bar{u}[1] = [\![\, d^1 \,]\!] \wedge \forall j \in \mathbb{N} : \bar{u}[j+1] = \bar{u}[j] \gtrsim [\![\, d^{j+1} \,]\!]\}$$

Because we treat $[\![\, d \,]\!]$ as a flat set, we see that $chains([\![\, d \,]\!])$ is a singleton set $\{\bar{u}\}$ where

$$\bar{u} = [\![\, d \,]\!]^{\leq 1}, [\![\, d \,]\!]^{\leq 2}, [\![\, d \,]\!]^{\leq 3}, \ldots$$

which means we have that $\bar{u}[j] = [\![\, d \,]\!]^{\leq j}$ for all $j \in \mathbb{N}$. The function pos may then be simplified to

$$pos(\bar{u}) \stackrel{\mathsf{def}}{=} \{\bar{t} \in \mathcal{H}^\infty \mid \forall j \in \mathbb{N} : \bar{t}[j] \in \bar{u}[j] \wedge \exists h \in \mathcal{H} : \bar{t}[j+1] \in \{\bar{t}[j]\} \gtrsim \{h\}\}$$

The approximation $\sqcup\bar{u}$ becomes

$$\sqcup\bar{u} \stackrel{\mathsf{def}}{=} \bigcup_{\bar{t} \in pos(\bar{u})} \sqcup\bar{t}$$

where $\sqcup\bar{t}$ is defined as in appendix 5.3. Because $chains([\![\, d \,]\!])$ is a singleton set we can let

$$\mu_\infty[\![\, d \,]\!] = \sqcup\bar{u} \quad \textbf{for} \quad chains([\![\, d \,]\!]) = \{\bar{u}\}$$

which means

$$\mu_\infty[\![\, d \,]\!] = \bigcup_{\bar{t} \in pos(\bar{u})} \sqcup\bar{t} \quad \textbf{for} \quad chains([\![\, d \,]\!]) = \{\bar{u}\}$$

In fact, if we define

$$\begin{array}{l} tra([\![\, d \,]\!]) \stackrel{\mathsf{def}}{=} \\ \quad \{\bar{t} \in \mathcal{H}^\infty \mid \forall j \in \mathbb{N} : \bar{t}[j] \in [\![\, d \,]\!]^{\leq j} \wedge \exists h \in \mathcal{H} : \bar{t}[j+1] \in \{\bar{t}[j]\} \gtrsim \{h\}\} \end{array} \tag{A.4}$$

we get that

$$\mu_\infty[\![\, d \,]\!] \stackrel{\mathsf{def}}{=} \bigcup_{\bar{t} \in tra([\![\, d \,]\!])} \sqcup\bar{t} \tag{A.5}$$

In the following we use this definition of $\mu_\infty[\![\, d \,]\!]$.

We start by assuming that the body d of an infinite loop $\mathsf{loop}\langle\infty\rangle\ d$ only characterizes finite behavior, i.e., we have no nesting of infinite loops. Further we assume that $\mathsf{loop}\langle\infty\rangle$ is the outermost operator of a diagram. When soundness and completeness of this case are established we consider combinations of infinite loop with other operators.

Lemma 19 $t \in [\![\, \mathsf{loop}\langle\infty\rangle\ d \,]\!] \iff t \in \mathcal{H} \wedge \exists \bar{t} \in \mathcal{H}^\infty : (\forall j \in \mathbb{N} : (\bar{t}[j] \in [\![\, d \,]\!]^{\leq j} \wedge \exists h \in \mathcal{H} : \bar{t}[j+1] \in \{\bar{t}[j]\} \gtrsim \{h\}) \wedge \forall l \in \mathcal{L} : e.l \circledS t = \sqcup_l \bar{t})$

Proof of lemma 19

PROVE: $t \in [\![\mathsf{loop}\langle\infty\rangle\ d]\!] \iff t \in \mathcal{H} \wedge \exists \bar{t} \in \mathcal{H}^\infty : (\forall j \in \mathbb{N} : (\bar{t}[j] \in [\![d]\!]^{\leq j} \wedge \exists h \in \mathcal{H} : \bar{t}[j+1] \in \{\bar{t}[j]\} \succsim \{h\}) \wedge \forall l \in \mathcal{L} : e.l \circledS t = \sqcup_l \bar{t})$

$\langle 1\rangle 1.\ t \in [\![\mathsf{loop}\langle\infty\rangle\ d]\!] = \mu_\infty [\![d]\!] = \bigcup_{\bar{t} \in tra([\![d]\!])} \sqcup \bar{t}$

PROOF: Defs. (5.23) and (A.5).

$\langle 1\rangle 2.\ t \in \sqcup \bar{t}$ for some $\bar{t} \in tra([\![d]\!])$

PROOF: $\langle 1\rangle 1$.

$\langle 1\rangle 3.\ t \in \mathcal{H} \wedge \forall l \in \mathcal{L} : e.l \circledS t = \sqcup_l \bar{t}$ for some $\bar{t}$ such that $\bar{t} \in \mathcal{H}^\infty \wedge \forall j \in \mathbb{N} : \bar{t}[j] \in [\![d]\!]^{\leq j} \wedge \exists h \in \mathcal{H} : \bar{t}[j+1] \in \{\bar{t}[j]\} \succsim \{h\}$

PROOF: $\langle 1\rangle 2$ and defs. (5.28) and (A.4).

$\langle 1\rangle 4.$ Q.E.D.

PROOF: $\langle 1\rangle 3$.

□

Lemma 20 *Given diagram $d \in \mathcal{D}$ without infinite loops. If $[\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d]$ may produce trace t, then, for any k, there exists a trace t' such that $[\emptyset, d^1\ \mathsf{seq}\ d^2\ \mathsf{seq}\ \cdots\ \mathsf{seq}\ d^k\ \mathsf{seq}\ \mathsf{loop}^{k+1}\langle\infty\rangle\ d]$ may produce t' and $\mathcal{E} \circledS t = \mathcal{E} \circledS t'$.*

Proof of lemma 20

ASSUME: t is produced by $[\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d]$

PROVE: There exists a trace t' such that t' is produced by $[\emptyset, d^1\ \mathsf{seq}\ d^2\ \mathsf{seq}\ \cdots\ \mathsf{seq}\ d^k\ \mathsf{seq}\ \mathsf{loop}^{k+1}\langle\infty\rangle\ d]$ and $\mathcal{E} \circledS t = \mathcal{E} \circledS t'$.

PROOF: by induction on k

$\langle 1\rangle 1.$ CASE: $k = 1$

(induction start)

$\langle 2\rangle 1.$ The only possible execution step from $[\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d]$ is $[\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d] \xrightarrow{\tau_{loop}} [\emptyset, d^1\ \mathsf{seq}\ \mathsf{loop}^2\langle\infty\rangle\ d]$

PROOF: Rules (8.6) and (A.1).

$\langle 2\rangle 2.$ There exists trace s such that s is produced by $[\emptyset, d^1\ \mathsf{seq}\ \mathsf{loop}^2\langle\infty\rangle\ d]$ and $t = \langle\tau_{loop}\rangle^\frown s$

PROOF: Assumption and $\langle 2\rangle 1$.

$\langle 2\rangle 3.$ LET: $t' = s$

$\langle 2\rangle 4.\ \mathcal{E} \circledS t = \mathcal{E} \circledS (\langle\tau_{loop}\rangle^\frown t') = \mathcal{E} \circledS t'$

PROOF: $\langle 2\rangle 1$, $\langle 2\rangle 2$, $\langle 2\rangle 3$, def. of $\circledS$ and $\tau_{loop} \notin \mathcal{E}$.

$\langle 2\rangle 5.$ Q.E.D.

PROOF: $\langle 2\rangle 4$.

$\langle 1\rangle 2.$ CASE: $k = j + 1$ (induction step)

ASSUME: t' is produced by $[\emptyset, d^1\ \mathsf{seq}\ d^2\ \mathsf{seq}\ \cdots\ \mathsf{seq}\ d^j\ \mathsf{seq}\ \mathsf{loop}^{j+1}\langle\infty\rangle\ d]$ and $\mathcal{E} \circledS t = \mathcal{E} \circledS t'$ (induction hypothesis)

PROVE: t'' is produced by $[\emptyset, d^1\ \mathsf{seq}\ d^2\ \mathsf{seq}\ \cdots\ \mathsf{seq}\ d^j\ \mathsf{seq}\ d^{j+1}\ \mathsf{seq}\ \mathsf{loop}^{j+2}\langle\infty\rangle\ d]$ and $\mathcal{E} \circledS t = \mathcal{E} \circledS t''$

$\langle 2\rangle 1.$ There exist traces s, s', diagram $d' \in \mathcal{D}$, and $\beta \in \mathcal{B}$ such that

$[\emptyset, d^1\ \mathsf{seq}\ d^2\ \mathsf{seq}\ \cdots\ \mathsf{seq}\ d^j\ \mathsf{seq}\ \mathsf{loop}^{j+1}\langle\infty\rangle \xrightarrow{s} [\beta, d'\ \mathsf{seq}\ \mathsf{loop}^{j+1}\langle\infty\rangle\ d] \xrightarrow{\tau_{loop}} [\beta, d'\ \mathsf{seq}\ d^{j+2}\ \mathsf{seq}\ \mathsf{loop}^{j+1}\langle\infty\rangle\ d] \xrightarrow{s'}$

and $t' = s^\frown\langle\tau_{loop}\rangle^\frown s'$

PROOF: Induction hypothesis and rules (8.6), (8.11), (8.12) and (A.1).

$\langle 2\rangle 2$. $[\emptyset, d^1 \text{ seq } d^2 \text{ seq } \cdots \text{ seq } d^j \text{ seq } d^{j+1}\mathsf{loop}^{j+2}\langle\infty\rangle\ d] \xrightarrow{s}$
$[\beta, d' \text{ seq } d^{j+1} \text{ seq } \mathsf{loop}^{j+2}\langle\infty\rangle\ d] \xrightarrow{s'}$
PROOF: $\langle 2\rangle 1$.

$\langle 2\rangle 3$. LET: $t'' = s^\frown s'$

$\langle 2\rangle 4$. $\mathcal{E}\circledS t' = \mathcal{E}\circledS(s^\frown\langle\tau_{loop}\rangle^\frown s') = \mathcal{E}\circledS(s^\frown s') = \mathcal{E}\circledS t''$
PROOF: $\langle 2\rangle 1$, $\langle 2\rangle 3$, properties of $\circledS$ and $\tau_{loop} \notin \mathcal{E}$.

$\langle 2\rangle 5$. $\mathcal{E}\circledS t = \mathcal{E}\circledS t''$
PROOF: Induction hypothesis and $\langle 2\rangle 4$.

$\langle 2\rangle 6$. Q.E.D.
PROOF: $\langle 2\rangle 2$, $\langle 2\rangle 4$ and $\langle 2\rangle 5$.

$\langle 1\rangle 3$. Q.E.D.
PROOF: $\langle 1\rangle 1$ and $\langle 1\rangle 2$.

□

Theorem 6 (Soundness) *Let d be a diagram without infinite loops. If $[\emptyset, \mathsf{loop}\langle\infty\rangle\ d]$ produces the trace t, then $\mathcal{E}\circledS t \in [\![\ \mathsf{loop}\langle\infty\rangle\ d\ ]\!]$.*

Proof of theorem 6

ASSUME: 1. d is a diagram without infinite loops
2. $[\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d]$ produces trace t

LET: $t' = \mathcal{E}\circledS t$

PROVE: $t' \in [\![\ \mathsf{loop}\langle\infty\rangle\ d\ ]\!]$

$\langle 1\rangle 1$. $t' \in [\![\ \mathsf{loop}\langle\infty\rangle\ d\ ]\!] \iff t' \in \mathcal{H} \wedge \exists \bar{t} \in \mathcal{H}^\infty : (\forall j \in \mathbb{N} : (\bar{t}[j] \in [\![\ d\ ]\!]^{\leq j} \wedge \exists h \in \mathcal{H} : \bar{t}[j+1] \in \{\bar{t}[j]\} \gtrsim \{h\}) \wedge \forall l \in \mathcal{L} : e.l\circledS t' = \sqcup_l \bar{t})$
PROOF: Lemma 19.

$\langle 1\rangle 2$. $t' \in \mathcal{H} \wedge \exists \bar{t} \in \mathcal{H}^\infty : (\forall j \in \mathbb{N} : (\bar{t}[j] \in [\![\ d\ ]\!]^{\leq j} \wedge \exists h \in \mathcal{H} : \bar{t}[j+1] \in \{\bar{t}[j]\} \gtrsim \{h\}) \wedge \forall l \in \mathcal{L} : e.l\circledS t' = \sqcup_l \bar{t})$
PROOF: by contradiction
ASSUME: $t' \notin \mathcal{H} \vee \forall \bar{t} \in \mathcal{H}^\infty : (\exists j \in \mathbb{N} : (\bar{t}[j] \notin [\![\ d\ ]\!]^{\leq j} \vee \forall h \in \mathcal{H} : \bar{t}[j+1] \notin \{\bar{t}[j]\} \gtrsim \{h\}) \vee \exists l \in \mathcal{L} : e.l\circledS t' \neq \sqcup_l \bar{t})$
PROVE: Contradiction

$\langle 2\rangle 1$. CASE: $t' \notin \mathcal{H}$
PROOF: We have assumed that all messages are unique, so this must mean that there exist a message m, traces s, s', diagrams d_1, d_2, d_3, d_4 and states of the communication medium $\beta'_1, \beta'_2, \beta'_3, \beta'_4$ such that

$$[\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d] \xrightarrow{s} [\beta'_1, d_1] \xrightarrow{(?,m)} [\beta'_2, d_2] \xrightarrow{s'} [\beta'_3, d_3] \xrightarrow{(!,m)} [\beta'_4, d_4]$$

and

$$\mathcal{E}\circledS(s^\frown\langle(?,m)\rangle^\frown s'^\frown\langle(!,m)\rangle) \sqsubseteq t'$$

By rules (8.3) and (8.10) this is clearly not possible and we must have that $t' \in \mathcal{H}$ and the case assumption is impossible.

$\langle 2\rangle 2$. CASE: $\forall \bar{t} \in \mathcal{H}^\infty : (\exists j \in \mathbb{N} : (\bar{t}[j] \notin [\![\ d\ ]\!]^{\leq j} \vee \forall h \in \mathcal{H} : \bar{t}[j+1] \notin \{\bar{t}[j]\} \gtrsim \{h\}) \vee \exists l \in \mathcal{L} : e.l\circledS t' \neq \sqcup_l \bar{t})$

$\langle 3\rangle 1$. LET: $s_1, s_2, s_3, \ldots, s_i, \ldots$ be traces and $\beta_1, \beta_2, \beta_3, \ldots, \beta_i, \ldots$ be states of the

communication medium such that

$$[\emptyset, d^1] \xrightarrow{s_1} [\beta_1, \mathsf{skip}]$$
$$[\emptyset, d^1 \ \mathsf{seq}\ d^2] \xrightarrow{s_1} [\beta_1, d^2] \xrightarrow{s_2} [\beta_2, \mathsf{skip}]$$
$$[\emptyset, d^1 \ \mathsf{seq}\ d^2 \ \mathsf{seq}\ d^3] \xrightarrow{s_1 \frown s_2} [\beta_2, d^3] \xrightarrow{s_3} [\beta_3, \mathsf{skip}]$$
$$\vdots$$
$$[\emptyset, d^1 \ \mathsf{seq}\ d^2 \cdots \ \mathsf{seq}\ d^i] \xrightarrow{s_1 \frown s_2 \frown \cdots \frown s_{i-1}} [\beta_{i-1}, d^i] \xrightarrow{s_i} [\beta_i, \mathsf{skip}]$$
$$\vdots$$

where we choose $s_1, s_2, \ldots, s_i, \ldots$ such that $ev.d^i \circledS s_i = ev.d^i \circledS t$. Further we let $t_1, t_2, t_3, \ldots, t_i, \ldots$ be traces such that

$$t_1 = \mathcal{E} \circledS s_1$$
$$t_2 = t_1 \frown (\mathcal{E} \circledS s_2)$$
$$t_3 = t_2 \frown (\mathcal{E} \circledS s_3)$$
$$\vdots$$
$$t_i = t_{i-1} \frown (\mathcal{E} \circledS s_i)$$
$$\vdots$$

⟨3⟩2. $t_j \in [\![d]\!]^{\leq j}$ for all $j \in \mathbb{N}$
PROOF: ⟨3⟩1 and theorem 4.
⟨3⟩3. LET: $\bar{t} = t_1, t_2, t_3, \ldots, t_i, \ldots$
⟨3⟩4. CASE: $\exists j \in \mathbb{N} : (\bar{t}[j] \notin [\![d]\!]^{\leq j} \vee \forall h \in \mathcal{H} : \bar{t}[j+1] \notin \{\bar{t}[j]\} \precsim \{h\})$
⟨4⟩1. Choose arbitrary $j \in \mathbb{N}$
⟨4⟩2. CASE: $\bar{t}[j] \notin [\![d]\!]^{\leq j}$
⟨5⟩1. $\bar{t}[j] \in [\![d]\!]^{\leq j}$
PROOF: By ⟨3⟩3, $\bar{t}[j] = t_j$ and by ⟨3⟩2, $t_j \in [\![d]\!]^{\leq j}$.
⟨5⟩2. Q.E.D.
PROOF: By ⟨5⟩1, ⟨4⟩2 is impossible.
⟨4⟩3. CASE: $\forall h \in \mathcal{H} : \bar{t}[j+1] \notin \{\bar{t}[j]\} \precsim \{h\}$
⟨5⟩1. $\bar{t}[j+1] = t_{j+1} = t_j \frown (\mathcal{E} \circledS s_{j+1}) = \bar{t}[j] \frown (\mathcal{E} \circledS s_{j+1})$
PROOF: ⟨3⟩1 and ⟨3⟩3.
⟨5⟩2. $\bar{t}[j+1] \in \{\bar{t}[j]\} \precsim \{\mathcal{E} \circledS s_{j+1}\}$
PROOF: ⟨5⟩1.
⟨5⟩3. $\mathcal{E} \circledS s_{j+1} \in \mathcal{H}$
PROOF: By ⟨3⟩1, $[\beta_j, d^{j+1}] \xrightarrow{s_{j+1}} [\beta_{j+1}, \mathsf{skip}]$, so this follows from lemma 6 and theorem 4.
⟨5⟩4. Q.E.D.
PROOF: Let $h = \mathcal{E} \circledS s_{j+1}$. By ⟨5⟩2 and ⟨5⟩3 it is clear that ⟨4⟩3 is impossible.
⟨4⟩4. Q.E.D.
PROOF: Since j is chosen arbitrarily, and ⟨4⟩2 and ⟨4⟩3 are impossible for this arbitrarily chosen j, ⟨3⟩4 is impossible.
⟨3⟩5. CASE: $\exists l \in \mathcal{L} : e.l \circledS t' \neq \sqcup_l \bar{t}$
This means that there must exist l such that $e.l \circledS t' \sqsubset \sqcup_l \bar{t}$ or $\sqcup_l \bar{t} \sqsubset e.l \circledS t'$ or $\exists j \in \mathbb{N} : (e.l \circledS t')[j] \neq (\sqcup_l \bar{t})[j]$.
⟨4⟩1. Choose arbitrary $l \in \mathcal{L}$.
⟨4⟩2. For any k, $e.l \circledS s_1 \frown e.l \circledS s_2 \frown \cdots \frown e.l \circledS s_k \sqsubseteq e.l \circledS t$

⟨5⟩1. There exists t'' such that t'' is produced by $[\emptyset, d^1 \mathsf{\ seq\ } d^2 \mathsf{\ seq\ } \cdots \mathsf{\ seq\ } d^k \mathsf{\ seq\ } \mathsf{loop}^{k+1}\langle\infty\rangle\ d]$ and $\mathcal{E} \circledS t = \mathcal{E} \circledS t''$
PROOF: Lemma 20.

⟨5⟩2. $(e.l \cap ev.d^1) \circledS s_1 \frown (e.l \cap ev.d^2) \circledS s_2 \frown \cdots \frown (e.l \cap d^k) \circledS s_k$
$= e.l \cap (ev.d^1 \cup ev.d^2 \cup \cdots \cup ev.d^k) \circledS t''$
PROOF: ⟨3⟩1, $ev.d^m \cap ev.d^n = \emptyset$ for $m \neq n$, and rules (8.3), (8.11) and (8.12) (which handles the ordering on l).

⟨5⟩3. $(e.l \cap ev.d^1) \circledS s_1 \frown (e.l \cap ev.d^2) \circledS s_2 \frown \cdots \frown (e.l \cap ev.d^k) \circledS s_k$
$= e.l \circledS s_1 \frown e.l \circledS s_2 \frown \cdots \frown e.l \circledS s_k$
PROOF: $ev.d^i \circledS s_i = \mathcal{E} \circledS s_i$ for $1 \leq i \leq k$ (⟨3⟩1) and $e.l \subseteq \mathcal{E}$ (by def. of $e.\_$), which implies that $(e.l \cap ev.d^i) \circledS s_i = e.l \circledS (ev.d^i \circledS s_i) = e.l \circledS (\mathcal{E} \circledS s_i) = (e.l \cap \mathcal{E}) \circledS s_i = e.l \circledS s_i$.

⟨5⟩4. $e.l \cap (ev.d^1 \cup ev.d^2 \cup \cdots \cup ev.d^k) \circledS t'' \sqsubseteq e.l \circledS t''$
PROOF: $(ev.d^1 \cup ev.d^2 \cup \cdots \cup ev.d^k) \cap (ev.\mathsf{loop}^{k+1}\langle\infty\rangle\ d) = \emptyset$ and rules (8.3), (8.11) and (8.12).

⟨5⟩5. Q.E.D.
PROOF:
$$\begin{aligned}
& e.l \circledS s_1 \frown e.l \circledS s_2 \frown \cdots \frown e.l \circledS s_k \\
&= (e.l \cap ev.d^1) \circledS s_1 \frown (e.l \cap ev.d^2) \circledS s_2 \frown \\
&\quad \cdots \frown (e.l \cap ev.d^k) \circledS s_k && (\langle 5\rangle 3) \\
&= e.l \cap (ev.d^1 \cup ev.d^2 \cup \cdots \cup ev.d^k) \circledS t'' && (\langle 5\rangle 2) \\
&\sqsubseteq e.l \circledS t'' && (\langle 5\rangle 4) \\
&= e.l \circledS t && (\langle 5\rangle 1 \text{ and } e.l \subseteq \mathcal{E})
\end{aligned}$$

⟨4⟩3. $e.l \circledS t' = e.l \circledS t$
PROOF: Assumption that $t' = \mathcal{E} \circledS t$ and $e.l \subseteq \mathcal{E}$.

⟨4⟩4. CASE: $e.l \circledS t' \sqsubset \sqcup_l \bar{t}$

⟨5⟩1. $\exists h \in \mathcal{H} : h \neq \langle\rangle \wedge \sqcup_l \bar{t} = (e.l \circledS t') \frown h$
PROOF: Case assumption (⟨4⟩4) and the properties of $\sqsubset$.

⟨5⟩2. $\#(e.l \circledS t') \neq \infty$ (i.e. $e.l \circledS t'$ is finite)
PROOF: ⟨5⟩1 and $\#s = \infty \Rightarrow s \frown r = s$

⟨5⟩3. LET: k be the smallest number such that $e.l \circledS t_k = e.l \circledS s_1 \frown e.l \circledS s_2 \frown \cdots \frown e.l \circledS s_k = e.l \circledS t'$
PROOF: By ⟨4⟩2, ⟨4⟩3 and ⟨5⟩2 such k must exist (because at some point the s_i stops containing events from $e.l$ or else $e.l \circledS t'$ would not be finite).

⟨5⟩4. $e.l \circledS s_i = \langle\rangle$ for all $i > k$
PROOF: This follows direclty from ⟨5⟩3.

⟨5⟩5. $\sqcup_l \bar{t} = e.l \circledS t_k$
PROOF: By ⟨3⟩1, ⟨3⟩3 and def. of $\sqcup_l \bar{t}$, $\sqcup_l \bar{t}$ is the least upper bound of
$$e.l \circledS t_1, e.l \circledS t_2, \ldots, e.l \circledS t_k, e.l \circledS t_{k+1}, \ldots = e.l \circledS s_1, e.l \circledS (s_1 \frown s_2), \ldots,$$
$$e.l \circledS (s_1 \frown s_2 \frown \cdots \frown s_k), e.l \circledS (s_1 \frown s_2 \frown \cdots \frown s_k \frown s_{k+1}), \ldots$$
By ⟨5⟩4 we must have that $\sqcup_l \bar{t} = e.l \circledS t_k$ or else it would not be the *least* upper bound.

⟨5⟩6. $\sqcup_l \bar{t} = e.l \circledS t'$
PROOF: ⟨5⟩3 and ⟨5⟩5.

⟨5⟩7. Q.E.D.
PROOF: ⟨5⟩5 contradicts ⟨5⟩1, and hence $e.l \circledS t' \sqsubset \sqcup_l \bar{t}$ is impossible.

⟨4⟩5. CASE: $\sqcup_l \bar{t} \sqsubset e.l \circledS t'$

⟨5⟩1. $\exists h \in \mathcal{H} : h \neq \langle\rangle \wedge e.l \circledS t' = (\sqcup_l \bar{t}) \frown h$
PROOF: Case assumption (⟨4⟩5) and properties of $\sqsubset$.
⟨5⟩2. $\#(\sqcup_l \bar{t}) \neq \infty$ (i.e. $\sqcup_l \bar{t}$ is finite)
PROOF: ⟨5⟩1 and $\#s = \infty \Rightarrow s \frown r = s$.
⟨5⟩3. There exists k such that $e.l \circledS t_k = e.l \circledS t_i$ for all $i > k$
PROOF: By ⟨5⟩2, $\sqcup_l \bar{t}$ is finite and by def. $e.l \circledS t_i \sqsubseteq \sqcup_l \bar{t}$ for all i, so there exists a finite bound on $e.l \circledS t_i$ and a place where the $e.l \circledS t_i$ for $i \in \mathbb{N}$ stop growing.
⟨5⟩4. $e.l \circledS s_i = \langle\rangle$ for all $i > k$
PROOF: ⟨3⟩1 and ⟨5⟩3.
⟨5⟩5. $\sqcup_l \bar{t} = e.l \circledS t_k$
PROOF: ⟨3⟩3, ⟨5⟩3 and the def. of $\sqcup_l \bar{t}$ as the least upperbound of

$$e.l \circledS \bar{t}[1], e.l \circledS \bar{t}[2], \ldots, e.l \circledS \bar{t}[k], \ldots$$

⟨5⟩6. LET: $h[1] = e$. Then $(\sqcup_l \bar{t}) \frown \langle e \rangle \sqsubseteq e.l \circledS t'$
PROOF: This follows directly from ⟨5⟩1.
⟨5⟩7. There exists $n \in \mathbb{N}$ such that $t'[n] = e$
PROOF: ⟨5⟩6.
⟨5⟩8. There exists $j \in \mathbb{N}$ such that $e \in ev.d^j$
PROOF: ⟨5⟩7, assumption 2, assumption that $t' = \mathcal{E} \circledS t$ and lemma 20.
⟨5⟩9. $e.l \circledS t_j \sqsubseteq e.l \circledS t'$
PROOF: ⟨4⟩2 and ⟨4⟩3.
⟨5⟩10. $j > k$
PROOF: By ⟨5⟩5 and ⟨5⟩6, $(e.l \circledS t_k) \frown \langle e \rangle \sqsubseteq e.l \circledS t'$, so by ⟨5⟩8 and ⟨5⟩9 this must be the case (or else rule (8.12) is violated).
⟨5⟩11. $(e.l \circledS t_k) \frown \langle e \rangle \sqsubseteq e.l \circledS t_j = e.l \circledS s_1 \frown e.l \circledS s_2 \frown \cdots \frown e.l \circledS s_k \frown \cdots \frown e.l \circledS s_j$
PROOF: ⟨3⟩1, ⟨4⟩3, ⟨5⟩5, ⟨5⟩6, ⟨5⟩9 and ⟨5⟩10.
⟨5⟩12. $e.l \circledS s_j \neq \langle\rangle$
PROOF: ⟨3⟩1, ⟨5⟩8 and ⟨5⟩11.
⟨5⟩13. Q.E.D.
PROOF: ⟨5⟩10 and ⟨5⟩12 contradicts ⟨5⟩4, so we must have that ⟨4⟩5 is impossible.
⟨4⟩6. CASE: $\exists j \in \mathbb{N} : (e.l \circledS t')[j] \neq (\sqcup_l \bar{t})[j]$
ASSUME: j is the smallest such number, i.e.:

$$(e.l \circledS t')|_{j-1} = (\sqcup_l \bar{t})|_{j-1} \wedge (e.l \circledS t')[j] \neq (\sqcup_l \bar{t})[j]$$

⟨5⟩1. LET: k be the smallest number such that $\#(e.l \circledS \bar{t}[k]) \geq j$
PROOF: Such k must exist. $\sqcup_l \bar{t}$ is defined as the least upper bound of $e.l \circledS \bar{t}[1], e.l \circledS \bar{t}[2], \ldots, e.l \circledS \bar{t}[i], \ldots$ If $\sqcup_l \bar{t}$ is finite, then by the case assumption (⟨4⟩6), $\#(\sqcup_l \bar{t}) \geq j$, and there must exist k such that $e.l \circledS \bar{t}[k] = \sqcup_l \bar{t}$ (or else it is not the least upper bound), which implies $\#(e.l \circledS \bar{t}[k]) \geq j$. If $\sqcup_l \bar{t}$ is infinite this means there is no finite bound on the length of the $e.l \circledS \bar{t}[i]$ and we may find k such that $\#(e.l \circledS \bar{t}[k]) \geq n$ for any $n \in \mathbb{N}$.
⟨5⟩2. $e.l \circledS \bar{t}[k] \sqsubseteq e.l \circledS t'$
⟨6⟩1. $e.l \circledS \bar{t}[k] = e.l \circledS s_1 \frown e.l \circledS s_2 \frown \cdots \frown e.l \circledS s_k$
⟨7⟩1. $\bar{t}[k] = t_k = \mathcal{E} \circledS s_1 \frown \mathcal{E} \circledS s_2 \frown \cdots \frown \mathcal{E} \circledS s_k$
PROOF: ⟨3⟩1 and ⟨3⟩3.
⟨7⟩2. Q.E.D.

PROOF: $\langle 7\rangle 1$ and $e.l \subseteq \mathcal{E}$.

$\langle 6\rangle 2$. $e.l \circledS s_1 \frown e.l \circledS s_2 \frown \cdots \frown e.l \circledS s_k \sqsubseteq e.l \circledS t$

PROOF: $\langle 4\rangle 2$.

$\langle 6\rangle 3$. Q.E.D.

PROOF: $\langle 4\rangle 3$, $\langle 6\rangle 1$ and $\langle 6\rangle 2$.

$\langle 5\rangle 3$. $e.l \circledS \bar{t}[k] \sqsubseteq \sqcup_l \bar{t}$

PROOF: This follow directly from the definition of $\sqcup_l \bar{t}$.

$\langle 5\rangle 4$. $(e.l \circledS t')[j] = (\sqcup_l \bar{t})[j]$

PROOF: From $\langle 5\rangle 2$ and $\langle 5\rangle 3$ we have that $e.l \circledS \bar{t}[k] \sqsubseteq e.l \circledS t'$ and $e.l \circledS \bar{t}[k] \sqsubseteq \sqcup_l \bar{t}$. By assumption $j \leq \#(e.l \circledS \bar{t}[k])$, so we must have that $(e.l \circledS t')[j] = (e.l \circledS \bar{t}[k])[j] = (\sqcup_l \bar{t})[j]$.

$\langle 5\rangle 5$. Q.E.D.

PROOF: From $\langle 5\rangle 4$ it is clear that $\langle 4\rangle 6$ is impossible.

$\langle 4\rangle 7$. Q.E.D.

PROOF: Because $\langle 4\rangle 4$, $\langle 4\rangle 5$ and $\langle 4\rangle 6$ is impossible for an arbitrarily chosen l, $\langle 3\rangle 5$ is also impossible.

$\langle 3\rangle 6$. Q.E.D.

PROOF: Because $\langle 3\rangle 4$ and $\langle 3\rangle 5$ are impossible, we have found a $\bar{t}$ that contradicts $\langle 2\rangle 2$.

$\langle 2\rangle 3$. Q.E.D.

PROOF: Because $\langle 2\rangle 1$ and $\langle 2\rangle 2$ are impossible we must have that $\langle 1\rangle 2$ is true.

$\langle 1\rangle 3$. Q.E.D.

PROOF: $\langle 1\rangle 1$ and $\langle 1\rangle 2$.

□

Lemma 21 *Let d be a diagram without infinite loops and with $env.d = \emptyset$ (i.e. no external communication). If $t \in [\![\mathsf{loop}\langle\infty\rangle\ d]\!]$, then for all $k \in \mathbb{N} \cup \{0\}$ there exist trace t', $\beta \in \mathcal{B}$ and $d' \in \mathcal{D}$ such that*

$$[\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d] \xrightarrow{t'} [\beta, d'] \wedge \mathcal{E} \circledS t' = t|_k$$

Proof of lemma 21

ASSUME: 1. d is a diagram without infinite loops

2. $env.d = \emptyset$

3. $t \in [\![\mathsf{loop}\langle\infty\rangle\ d]\!]$

PROVE: For all $k \in \mathbb{N} \cup \{0\}$ there exist t', β, d' such that $[\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d] \xrightarrow{t'} [\beta, d'] \wedge \mathcal{E} \circledS t' = t|_k$

PROOF: by induction on k

$\langle 1\rangle 1$. CASE: $k = 0$ (induction start)

$\langle 2\rangle 1$. $t|_k = t|_0 = \langle\rangle$

PROOF: $\langle 1\rangle 1$ and properties of $\_|\_$.

$\langle 2\rangle 2$. LET: $t' = \langle \tau_{loop} \rangle$

$\langle 2\rangle 3$. $\mathcal{E} \circledS \langle \tau_{loop} \rangle = \langle\rangle$

PROOF: $\tau_{loop} \notin \mathcal{E}$ and properties of $\_\circledS\_$.

$\langle 2\rangle 4$. $[\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d] \xrightarrow{t'} [\emptyset, d^1\ \mathsf{seq}\ \mathsf{loop}^2\langle\infty\rangle\ d] \wedge \mathcal{E} \circledS t' = t|_k$

PROOF: This follows trivially from $\langle 2\rangle 1$, $\langle 2\rangle 2$, and rule (A.1).

$\langle 2\rangle 5$. Q.E.D.
PROOF: This follows from $\langle 2\rangle 3$ and $\langle 2\rangle 4$ by letting $\beta = \emptyset$ and $d' = d^1 \,\mathsf{seq}\, \mathsf{loop}^2\langle\infty\rangle\, d$.
$\langle 1\rangle 2$. CASE: $k+1$ (induction step)
ASSUME: $\exists t'', \beta', d'' : [\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d] \xrightarrow{t''} [\beta', d''] \wedge \mathcal{E}\circledS t'' = t|_k$ (induction hypothesis)
PROVE: $\exists t', \beta, d' : [\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d] \xrightarrow{t'} [\beta, d'] \wedge \mathcal{E}\circledS t' = t|_{k+1}$
$\langle 2\rangle 1$. LET: $t[k+1] = e$
$\langle 2\rangle 2$. $t|_{k+1} = t|_k{}^\frown\langle e\rangle \sqsubset t$
PROOF: $\langle 2\rangle 1$ (here we assume t to be infinite, but the case $t|_{k+1} = t$ would not change the following argument).
$\langle 2\rangle 3$. $\forall l \in \mathcal{L} : e.l\circledS t|_{k+1} = e.l\circledS(t|_k{}^\frown\langle e\rangle) \sqsubseteq e.l\circledS t$
PROOF: $\langle 2\rangle 1$, $\langle 2\rangle 2$ and properties of $l.\_$ and $\_\circledS\_$.
$\langle 2\rangle 4$. $t \in \mathcal{H} \wedge \exists \bar{t} \in \mathcal{H}^\infty : (\forall j \in \mathbb{N} : (\bar{t}[j] \in [\![\, d \,]\!]^{\leq j} \wedge\ \exists h \in \mathcal{H} : \bar{t}[j+1] \in \{\bar{t}[j]\} \gtrsim \{h\}) \wedge \forall l \in \mathcal{L} : e.l\circledS t = \sqcup_l \bar{t})$
PROOF: Assumption 3 and lemma 19.
$\langle 2\rangle 5$. LET: $n \in \mathbb{N}$ be the number such that $e \in d^n$ and let $m \in \mathbb{N}$ be the largest number such that $ev.d^m \cap ev.t|_k \neq \emptyset$
PROOF: By our assumption that all messages are unique and are enumerated by the iterations of the loop (cf. rule (A.1)), both n and m are uniquely identified.
$\langle 2\rangle 6$. LET: $j = \mathbf{max}(m, n)$
$\langle 2\rangle 7$. $\exists \bar{t} \in \mathcal{H}^\infty : \bar{t}[j] \in [\![\, d \,]\!]^{\leq j} \wedge \forall l \in \mathcal{L} : e.l\circledS t|_{k+1} \sqsubseteq e.l\circledS \bar{t}[j] \sqsubseteq e.l\circledS t$
$\langle 3\rangle 1$. $\exists \bar{t} \in \mathcal{H}^\infty : \bar{t}[j] \in [\![\, d \,]\!]^{\leq j} \wedge \forall l \in \mathcal{L} : e.l\circledS t = \sqcup_l \bar{t}$
PROOF: $\langle 2\rangle 4$.
$\langle 3\rangle 2$. $\exists \bar{t} \in \mathcal{H}^\infty : \bar{t}[j] \in [\![\, d \,]\!]^{\leq j} \wedge \forall l \in \mathcal{L} : e.l\circledS \bar{t}[j] \sqsubseteq \sqcup_l \bar{t} = e.l\circledS t$
PROOF: $\langle 3\rangle 1$ and def. of $\sqcup_l$.
$\langle 3\rangle 3$. $\forall l \in \mathcal{L} : e.l\circledS t|_{k+1} \sqsubseteq e.l\circledS t$
PROOF: $\langle 2\rangle 3$.
$\langle 3\rangle 4$. $\exists \bar{t} \in \mathcal{H}^\infty : \bar{t}[j] \in [\![\, d \,]\!]^{\leq j} \wedge \forall l \in \mathcal{L} : e.l\circledS t|_{k+1} \sqsubseteq e.l\circledS \bar{t}[j]$
PROOF: By $\langle 3\rangle 2$ and $\langle 3\rangle 3$ we must have that, for all l, $e.l\circledS t|_{k+1} \sqsubset e.l\circledS \bar{t}[j]$, $e.l\circledS t|_{k+1} = e.l\circledS \bar{t}[j]$ or $e.l\circledS \bar{t}[j] \sqsubset e.l\circledS t|_{k+1}$. But by $\langle 2\rangle 5$ and $\langle 2\rangle 6$ we must have that $\#\bar{t}[j] \geq \#t|_{k+1}$, so the latter is not possible.
$\langle 3\rangle 5$. Q.E.D.
PROOF: $\langle 3\rangle 2$, $\langle 3\rangle 3$ and $\langle 3\rangle 4$.
$\langle 2\rangle 8$. LET: $\bar{t}$ be such that $\bar{t} \in \mathcal{H}^\infty \wedge \bar{t}[j] \in [\![\, d \,]\!]^{\leq j} \wedge \forall l \in \mathcal{L} : e.l\circledS t|_{k+1} \sqsubseteq e.l\circledS \bar{t}[j] \sqsubseteq e.l\circledS t$
PROOF: By $\langle 2\rangle 6$, such $\bar{t}$ exists.
$\langle 2\rangle 9$. $\exists \beta', t'', d'' : \mathcal{E}\circledS t'' = t|_k \wedge [\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d] \xrightarrow{t''} [\beta', d'' \,\mathsf{seq}\, \mathsf{loop}^{j+1}\langle\infty\rangle\ d] \wedge t''[\#t''] \in \mathcal{E}$
PROOF: This follows from the induction hypothesis, $\langle 2\rangle 5$ and $\langle 2\rangle 6$. (Because m is the largest number such that $ev.d^m \cap ev.t|_k \neq \emptyset$ and $j \geq m$, j iterations of the loop is sufficient to produce t''.) $t''[\#t''] \in \mathcal{E}$ is obtained by not execution silent events after the execution of $t[k]$.
$\langle 2\rangle 10$. $\exists \beta', t'', d'' : \mathcal{E}\circledS t'' = t|_k \wedge [\emptyset, \mathsf{loop}^1\langle j\rangle\ d] \xrightarrow{t''} [\beta', d''] \wedge t''[\#t''] \in \mathcal{E}$
PROOF: This follows directly from $\langle 2\rangle 9$.
$\langle 2\rangle 11$. LET: β', t'' and d'' be such that $\mathcal{E}\circledS t'' = t|_k \wedge [\emptyset, \mathsf{loop}^1\langle j\rangle\ d] \xrightarrow{t''} [\beta', d''] \wedge t''[\#t''] \in \mathcal{E}$
PROOF: By $\langle 2\rangle 10$, such β', t'' and d'' must exist.

⟨2⟩12. $\bar{t}[j] \in [\![\mathsf{loop}\langle j \rangle\ d]\!]$
PROOF: This follows from ⟨2⟩8 (see theorem 8 on page 332).

⟨2⟩13. $\exists t_j, \beta_j : \mathcal{E} \circledS t_j = \bar{t}[j] \wedge [\emptyset, \mathsf{loop}^1\langle j \rangle\ d] \xrightarrow{t_j} [\beta_j, \mathsf{skip}]$
PROOF: ⟨2⟩12 and theorem 5.

⟨2⟩14. LET: t_j, and β_j be such that $\mathcal{E} \circledS t_j = \bar{t}[j] \wedge [\emptyset, \mathsf{loop}^1\langle j \rangle\ d] \xrightarrow{t_j} [\beta_j, \mathsf{skip}]$
PROOF: By ⟨2⟩13 such t_j and β_j exist.

⟨2⟩15. LET: $l.e = l$

⟨2⟩16. $e.l \circledS t|_{k+1} = e.l \circledS (t|_k \frown \langle e \rangle) = e.l \circledS (t|_k) \frown \langle e \rangle \sqsubseteq e.l \circledS \bar{t}[j]$
PROOF: This follows from ⟨2⟩2 and ⟨2⟩7 by ⟨2⟩2, $e.l \subseteq \mathcal{E}$ and the properties of $\_ \circledS \_$.

⟨2⟩17. $e.l \circledS t'' \frown \langle e \rangle \sqsubseteq e.l \circledS t_j$
PROOF: This follow from ⟨2⟩11, ⟨2⟩14 and ⟨2⟩16 by $e.l \subseteq \mathcal{E}$ and properties of $\_ \circledS \_$.

⟨2⟩18. $\exists s, \beta'', d''' : s \in \mathcal{T}^* \wedge [\beta', d''] \xrightarrow{s \frown \langle e \rangle} [\beta'', d''']$
PROOF: There are three way this may not be the case. 1) e is blocked by an event on lifeline e. But by ⟨2⟩14 and ⟨2⟩17 we know that a execution with e following $l.e \circledS t''$ is possible. 2) e is a receive event and its message is not available in β'. But we have assumed no external communication, t must obey the message invariant and we have assumed all messages to be unique, so this is not possible. 3) e is selected away in the execution of a choice operator. But, by ⟨2⟩8 and ⟨2⟩14 we know that all events of $t|_k$ can be executing without choosing away e, and after the execution of $t[k]$ we have full control over the choices by choosing s in the appropriate way.

⟨2⟩19. LET: s, β'' and d''' be such that $s \in \mathcal{T}^* \wedge [\beta', d''] \xrightarrow{s \frown \langle e \rangle} [\beta'', d''']$
PROOF: By ⟨2⟩18 such s, β'' and d''' exist.

⟨2⟩20. LET: $t' = t'' \frown s \frown \langle e \rangle$, $\beta = \beta''$ and $d' = d'''\ \mathsf{seq}\ \mathsf{loop}^{j+1}\langle \infty \rangle\ d$

⟨2⟩21. $[\emptyset, \mathsf{loop}^1\langle \infty \rangle\ d] \xrightarrow{t'} [\beta, d]$
PROOF: This follows from ⟨2⟩20, because from ⟨2⟩9, ⟨2⟩11 and ⟨2⟩18 we have $[\emptyset, \mathsf{loop}^1\langle \infty \rangle\ d] \xrightarrow{t''} [\beta', d''\ \mathsf{seq}\ \mathsf{loop}^{j+1}\langle \infty \rangle\ d] \xrightarrow{s \frown \langle e \rangle} [\beta'', d'''\ \mathsf{seq}\ \mathsf{loop}^{j+1}\langle \infty \rangle\ d]$.

⟨2⟩22. $\mathcal{E} \circledS t' = t|_{k+1}$
PROOF: By ⟨2⟩1, ⟨2⟩11, ⟨2⟩19 and ⟨2⟩20 we have
$\mathcal{E} \circledS t' = \mathcal{E} \circledS (t'' \frown s \frown \langle e \rangle) = (\mathcal{E} \circledS t'') \frown (\mathcal{E} \circledS s) \frown (\mathcal{E} \circledS \langle e \rangle) = t|_k \frown \langle\rangle \frown \langle e \rangle = t|_k \frown t[k+1] = t|_{k+1}$.

⟨2⟩23. Q.E.D.
PROOF: ⟨2⟩21 and ⟨2⟩22.

⟨1⟩3. Q.E.D.
PROOF: ⟨1⟩1 and ⟨1⟩2.

□

Theorem 7 (Completeness) *Let d be a diagram without infinite loop, and $env.d = \emptyset$ (i.e. without external communication). If $t \in [\![\mathsf{loop}\langle \infty \rangle\ d]\!]$ then there exists trace t' such that $\mathcal{E} \circledS t' = t$ and $[\emptyset, \mathsf{loop}\langle \infty \rangle\ d]$ produces t'.*

Proof of theorem 7
ASSUME: 1. d is a diagram without infinite loop

2. $env.d = \emptyset$
3. $t \in [\![\mathsf{loop}\langle\infty\rangle\ d]\!]$

PROVE: There exists t' such that $\mathcal{E}[\![s]\!]t' = t$ and $[\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d]$ produces t'

PROOF: by contradiction

There are two cases that must be considered: 1) There exists some point where further execution of t is impossible. 2) t does not satisfy the fairness constraints of the operational semantics.

$\langle 1\rangle 1$. CASE: $\exists k \in \mathbb{N} : \forall s, \beta, d' : (\mathcal{E}[\![s]\!] = t|_k \wedge [\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d] \xrightarrow{s} [\beta, d']) \Rightarrow [\beta, d'] \overset{t[k+1]}{\nrightarrow}$

$\langle 2\rangle 1$. $t|_{k+1} = t|_k \frown \langle t[k+1]\rangle \sqsubseteq t$

PROOF: Properties of $\_|\_$ and $\_[\_]$.

$\langle 2\rangle 2$. Q.E.D.

PROOF: By $\langle 2\rangle 1$, $t|_{k+1}$ is a finite prefix of t. But by assumptions 1, 2 and 3, and lemma 21 we are able to produce any finite prefix, so $\langle 1\rangle 1$ is impossible.

$\langle 1\rangle 2$. CASE: $\forall k \in \mathbb{N} : \exists s, \beta, d' : (\mathcal{E}[\![s]\!] = t|_k \wedge [\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d] \xrightarrow{s} [\beta, d']$
$\wedge\ [\beta, d'] \xrightarrow{t[k+1]}) \wedge \forall t' : (\mathcal{E}[\![t']\!] = t \Rightarrow \neg wft(t', \mathsf{loop}^1\langle\infty\rangle\ d))$

$\langle 2\rangle 1$. $\forall k \in \mathbb{N} : \exists s, \beta, d' : (\mathcal{E}[\![s]\!] = t|_k \wedge [\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d] \xrightarrow{s} [\beta, d'] \wedge\ [\beta, d'] \xrightarrow{t[k+1]})$

PROOF: This follows from the fact that $\langle 1\rangle 1$ is impossible.

$\langle 2\rangle 2$. $\forall t' : (\mathcal{E}[\![t']\!] = t \Rightarrow \neg wft(t', \mathsf{loop}^1\langle\infty\rangle\ d))$

PROOF: $\langle 1\rangle 2$ and $\langle 2\rangle 1$.

$\langle 2\rangle 3$. ASSUME: t' is such that $\mathcal{E}[\![t']\!] = t$

$\langle 2\rangle 4$. $\neg wft(\mathsf{loop}^1\langle\infty\rangle\ d, t')$

PROOF: $\langle 2\rangle 2$ and $\langle 2\rangle 3$.

$\langle 2\rangle 5$. $\neg\exists \sigma \in \Xi : \pi_2(head(\sigma)) = \mathsf{loop}^1\langle\infty\rangle\ d \wedge tr(\sigma) = t' \wedge wfe(\sigma)$

PROOF: $\langle 2\rangle 4$ and def. (8.27) of *wft*.

$\langle 2\rangle 6$. $\forall \sigma \in \Xi : \pi_2(head(\sigma)) \neq \mathsf{loop}^1\langle\infty\rangle\ d \vee tr(\sigma) \neq t' \vee \neg wfe(\sigma)$

PROOF: $\langle 2\rangle 5$.

$\langle 2\rangle 7$. ASSUME: σ is an execution such that
$\sigma = [\beta_1, d_1] \xrightarrow{x_1} [\beta_2, d_2] \xrightarrow{x_2} [\beta_3, d_3] \xrightarrow{x_3} \cdots \wedge$
$[\beta_1, d_1] = [\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d] \wedge tr(\sigma) = \langle x_1, x_2, x_3, \ldots\rangle = t'$

PROOF: By $\langle 2\rangle 1$ such σ must exist.

$\langle 2\rangle 8$. $\neg wfe(\sigma)$

PROOF: $\langle 2\rangle 6$ and $\langle 2\rangle 7$.

$\langle 2\rangle 9$. $\exists d' \in \mathcal{D}, i \in \mathbb{N} : (\forall j \in \mathbb{N} \cup \{0\} : enabled(d', [\beta_{i+j}, d_{i+j}]) \ \wedge\ \neg\exists k \in \mathbb{N} \cup \{0\} :$
$executed(d', [\beta_{i+k}, d_{i+k}], x_{i+k}, [\beta_{i+k+1}, d_{i+k+1}]))$

PROOF: $\langle 2\rangle 8$ and def. (8.25) of *wfe*.

$\langle 2\rangle 10$. LET: d' be the least complex diagram projection part (as by the weight function w defined in the proof of theorem 3) such that
$\exists i \in \mathbb{N} : (\forall j \in \mathbb{N} \cup \{0\} : enabled(d', [\beta_{i+j}, d_{i+j}]) \ \wedge\ \forall k \in \mathbb{N} \cup \{0\} :$
$\neg executed(d', [\beta_{i+k}, d_{i+k}], x_{i+k}, [\beta_{i+k+1}, d_{i+k+1}]))$

PROOF: By $\langle 2\rangle 9$ such d' exists, and we must then be able to find the least complex d'.

$\langle 2\rangle 11$. LET: i be the smallest number such that
$\forall j \in \mathbb{N} \cup \{0\} : enabled(d', [\beta_{i+j}, d_{i+j}]) \wedge$
$\forall k \in \mathbb{N} \cup \{0\} : \neg executed(d', [\beta_{i+k}, d_{i+k}], x_{i+k}, [\beta_{i+k+1}, d_{i+k+1}])$

PROOF: By $\langle 2\rangle 10$ such i exists, and we may then find the smallest.

$\langle 2\rangle 12$. $enabled(d', [\beta_i, d_i])$

PROOF: $\langle 2\rangle 11$.

⟨2⟩13. d' is a single event or d' is a diagram with a high-level operator as its most significant operator

PROOF: Assume not. Then there exists d'' and d''' such that $d' = d''$ seq d''', $d' = d''$ par d''' or $d' = d''$ strict d'''. But then by defs. (8.21), (8.22) and (8.23) we must have $enabled(d', [\beta_n, d_n]) \Rightarrow enabled(d'', [\beta_n, d_n]) \vee enabled(d''', [\beta_n, d_n])$ and likewise for *executed*. But d'' and d''' are less complex that d', so then d' is not the least complex diagram projection part that satisfy ⟨2⟩10.

⟨2⟩14. CASE: d' is a single event

⟨3⟩1. LET: $d' = e \in \mathcal{E}$

⟨3⟩2. $e \triangleleft d_1$

PROOF: ⟨2⟩12 and defs. (8.22) and (8.23) of *executed* and *enabled*.

⟨3⟩3. $\exists c \in \mathbb{N}, d'' \in \mathcal{D} : d_i = d''$ seq $\mathsf{loop}^c\langle\infty\rangle\ d \wedge e \triangleleft d''$

PROOF: By ⟨2⟩7, rules (A.1) and (8.6), and def. (8.21) of $\triangleleft$ this is the only way to obtain ⟨3⟩2.

⟨3⟩4. LET: c be the smallest number and d'' be such that $d_i = d''$ seq $\mathsf{loop}^c\langle\infty\rangle\ d \wedge e \triangleleft d''$

PROOF: By ⟨3⟩3 such c and d'' exists.

⟨3⟩5. $e \in ev.d^{c-1}$

PROOF: By ⟨3⟩4, c is the smallest number such that $d_i = d''$ seq $\mathsf{loop}^c\langle\infty\rangle\ d \wedge e \triangleleft d''$. If there existed $n < c-1$ such that $e \in ev.d^n$ there would have existed d''' such that $d_i = d'''$ seq $\mathsf{loop}^{n+1}\langle\infty\rangle\ d$, but then c would not be the smallest number to satisfy ⟨3⟩3.

⟨3⟩6. $[\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d] \xrightarrow{\langle x_1, x_2, \ldots, x_{i-1}\rangle} [\beta_i, d''$ seq $\mathsf{loop}^c\langle\infty\rangle\ d]$

PROOF: ⟨2⟩7 and ⟨3⟩4.

⟨3⟩7. $[\emptyset, \mathsf{loop}^1\langle c-1\rangle\ d] \xrightarrow{\langle x_1, x_2, \ldots, x_{i-1}\rangle} [\beta_i, d'']$

PROOF: By ⟨3⟩6 and definition of the loop rules.

⟨3⟩8. There exist s, β such that $[\beta_i, d''] \xrightarrow{s} [\beta, \mathsf{skip}]$

PROOF: Because of the syntactical constraints we can assume this to be the case, i.e. that there is nothing in d'' that prevent the execution to reach skip.

⟨3⟩9. LET: s and β be such that $[\beta_i, d''] \xrightarrow{s} [\beta, \mathsf{skip}]$

PROOF: By ⟨3⟩8, s and β must exist.

⟨3⟩10. $e \in ev.(\mathcal{E} \circledS s)$

PROOF: By ⟨3⟩3, $e \triangleleft d''$, so by the rules of the operational semantics this must be the case in an execution from d'' to skip.

⟨3⟩11. $\mathcal{E} \circledS (\langle x_1, x_2, \ldots, x_{i-1}\rangle^\frown s) \in [\![\, d \,]\!]^{\leq c-1}$

PROOF: $\mathsf{loop}^1\langle c-1\rangle\ d$ and be considered as d^1 seq d^2 seq $\cdots$ seq d^{c-1}, so by ⟨3⟩7, ⟨3⟩9, defs. (5.23), (A.2) and (A.3), and theorem 4 this must be the case.

⟨3⟩12. $t \in \mathcal{H} \wedge \exists \bar{t} \in \mathcal{H}^\infty : (\forall j \in \mathbb{N} : (\bar{t}[j] \in [\![\, d \,]\!]^{\leq j} \wedge \exists h \in \mathcal{H} : \bar{t}[j+1] \in \{\bar{t}[j]\} \gtrsim \{h\}) \wedge \forall l \in \mathcal{L} : e.l \circledS t = \sqcup_l \bar{t})$

PROOF: Assumption 3 and lemma 19.

⟨3⟩13. LET: $\bar{t} \in \mathcal{H}^\infty$ be such that $\forall j \in \mathbb{N} : \bar{t}[j] \in [\![\, d \,]\!]^{\leq j} \wedge \exists h \in \mathcal{H} : \bar{t}[j+1] \in \{\bar{t}[j]\} \gtrsim \{h\}) \wedge \forall l \in \mathcal{L} : e.l \circledS t = \sqcup_l \bar{t}$

PROOF: By ⟨3⟩12 such $\bar{t}$ exists.

⟨3⟩14. $e \in ev.\bar{t}[c-1]$

PROOF: We have established that $\mathcal{E} \circledS (\langle x_1, x_2, \ldots, x_{i-1}\rangle^\frown s) \in [\![\, d \,]\!]^{\leq c-1}$ with $e \in ev.(\mathcal{E} \circledS s)$ (⟨3⟩10 and ⟨3⟩11), $\bar{t}[c-1] \in [\![\, d \,]\!]^{\leq c-1}$ (⟨3⟩13) and $e \in ev.d^{c-1}$ (⟨3⟩5).

The only way we may have $e \notin ev.\bar{t}[c-1]$ is that e is inside and alt or xalt in d^{c-1}. But then, by theorem 4, there does not exist any finite execution

$$\sigma' = [\beta'_1, d'_1] \xrightarrow{x'_1} [\beta'_2, d'_2] \xrightarrow{x'_2} \cdots \xrightarrow{x'_{n-1}} [\beta'_n, d'_n]$$

with $\beta'_1 = \emptyset$, $d'_1 = \mathsf{loop}^1\langle c-1\rangle\ d$ and $d'_n = \mathsf{skip}$ such that $\exists j \in [1..n-1] : executed(e, [\beta'_j, d'_j], e, [\beta'_{j+1}, d'_{j+1}])$. I.e. we have that

$$\forall j \in [1..n-1] : \neg executed(e, [\beta_j,' d'_j], e, [\beta'_{j+1}, d'_{j+1}])$$

but then we we also have that

$$\forall j \in [1..n] : \neg enabled(e, [\beta'_j, d'_j])$$

because reaching skip means either that e is executed or is never enabled, and $enabled(e, [\beta'_n, d'_n])$ is impossible since $d'_n = \mathsf{skip}$. Because we have chosen an execution $\sigma = [\beta_1, d_1] \xrightarrow{x_1} [\beta_2, d_2] \xrightarrow{x_2} \cdots$ with $enabled(e, [\beta_i, d_i])$ ($\langle 2\rangle 7$, $\langle 2\rangle 12$ and $\langle 3\rangle 1$) this is not an interesting situation, and we may safely assume $\langle 3\rangle 14$.

$\langle 3\rangle 15$. $\exists m \in \mathbb{N} : t[m] = e$

$\langle 4\rangle 1$. Let: $l.e = l$

$\langle 4\rangle 2$. $\exists p \in \mathbb{N} : (\sqcup_l \bar{t})[p] = e$

Proof: $\langle 3\rangle 14$, def. of $\sqcup_l \bar{t}$ (because $e.l \circledS \bar{t}[c-1] \sqsubseteq \sqcup_l \bar{t}$) and $\langle 4\rangle 1$ (because $l.e = l \Leftrightarrow e \in e.l$)

$\langle 4\rangle 3$. $\exists p \in \mathbb{N} : (e.l \circledS t)[p] = e$

Proof: $\sqcup_l \bar{t} = e.l \circledS t$ (by $\langle 3\rangle 13$) and $\langle 4\rangle 2$.

$\langle 4\rangle 4$. Q.E.D.

Proof: By $\langle 4\rangle 3$ and properties of $e.\_$, $l.\_$ and $\_ \circledS \_$ such m must exist.

$\langle 3\rangle 16$. $\exists q \in \mathbb{N} : q \geq i \wedge x_q = e$

Proof: $\langle 2\rangle 3$, $\langle 2\rangle 7$, $\langle 2\rangle 12$ and $\langle 3\rangle 15$.

$\langle 3\rangle 17$. $executed(e, [\beta_q, d_q], e, [\beta_{q+1}, d_{q+1}])$

Proof: $\langle 3\rangle 16$ and the assumption that all events are unique.

$\langle 3\rangle 18$. Q.E.D.

Proof: $\langle 3\rangle 17$ contradict $\langle 2\rangle 10$.

$\langle 2\rangle 15$. Case: d' is a diagram with a high-level operator as its most significant operator

$\langle 3\rangle 1$. $\exists x, \beta^\dagger, d^\dagger : executed(d', [\beta_i, d_i], x, [\beta^\dagger, d^\dagger])$

Proof: $\langle 2\rangle 12$ and def. (8.23) of *enabled*.

$\langle 3\rangle 2$. Let: x, $\beta^\dagger$ and $d^\dagger$ be such that $executed(d', [\beta_i, d_i], x, [\beta^\dagger, d^\dagger])$

Proof: By $\langle 3\rangle 1$ such x, $\beta^\dagger$ and $d^\dagger$ must exist.

$\langle 3\rangle 3$. $x \in \mathcal{T} \wedge \beta^\dagger = \beta_i$

Proof: By $\langle 2\rangle 15$ and rule (8.6) this must be the case.

$\langle 3\rangle 4$. Let: $x_i = x$, $\beta_{i+i} = \beta_i = \beta^\dagger$ and $d_{i+1} = d^\dagger$

Proof: This not affect t because, by $\langle 2\rangle 3$ and $\langle 2\rangle 7$, $t = \mathcal{E} \circledS tr(\sigma)$ and, by $\langle 3\rangle 3$, $x_i = x \in \mathcal{T}$. By $\langle 2\rangle 2$ and $\langle 2\rangle 6$ we can use any σ and t' that have the property that $tr(\sigma) = t'$ and $\mathcal{E} \circledS t' = t$, and hence, by $\langle 3\rangle 2$ and $\langle 3\rangle 3$, we can safely do the assumption that $x_i = x$, $\beta_{i+1} = \beta_i$ and $d_{i+1} = d^\dagger$.

$\langle 3\rangle 5$. $executed(d', [\beta_i, d_i], x_i, [\beta_{i+1}, d_{i+1}])$

Proof: $\langle 3\rangle 2$, $\langle 3\rangle 3$ and $\langle 3\rangle 4$.

$\langle 3\rangle 6$. Q.E.D.

Proof: $\langle 3\rangle 5$ contradicts $\langle 2\rangle 10$.

$\langle 2\rangle 16$. Q.E.D.

Proof: Both $\langle 2\rangle 14$ and $\langle 2\rangle 15$ leads to contradictions, so $\langle 1\rangle 2$ is impossible.

$\langle 1\rangle 3$. Q.E.D.

PROOF: Both $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$ are impossible, so we must have that there exists trace t' such that $\mathcal{E} \circledS t' = t$ and $[\emptyset, \mathsf{loop}^1\langle\infty\rangle\ d]$ produces t'.

□

The above soundness and completeness theorems are concerned with diagrams that have only one infinite loop, and the infinite loop as the most significant operator. In the following we make some reflections on how these results carry over to other more complex cases. There are three main cases that must be considered:

1. Combinations d_1 op d_2 where op is an simple operator and d_1 or d_2 or both characterize infinite behavior.

2. Infinite loops inside other high-level operators.

3. The body d of an infinite loop $\mathsf{loop}\langle\infty\rangle\ d$ itself contains infinite loops.

In the first case, we have two simple sub-cases, and one more complicated. Consider the diagram $(\mathsf{loop}\langle\infty\rangle\ d_1)$ strict $(\mathsf{loop}\langle\infty\rangle\ d_2)$. It is obvious that the left operand of the strict will block the right operand, so that the diagram is semantically equivalent to $\mathsf{loop}\langle\infty\rangle\ d_1$. The other simple sub-case is the diagram $(\mathsf{loop}\langle\infty\rangle\ d_1)$ par $(\mathsf{loop}\langle\infty\rangle\ d_2)$, which simply yields the merges of the traces of $\mathsf{loop}\langle\infty\rangle\ d_1$ and $\mathsf{loop}\langle\infty\rangle\ d_2$. The weak fairness ensures that none of the par operands are starved infinitely long. Soundness and completeness considerations are then reduced to considerations of the oracle p in the denotational definition of par with respect to our definition of weak fairness. The complicated sub-case is the case of $(\mathsf{loop}\langle\infty\rangle\ d_1)$ seq $(\mathsf{loop}\langle\infty\rangle\ d_2)$. If we have that $ll.d_2 \subseteq ll.d_1$ this is semantically equivalent to $(\mathsf{loop}\langle\infty\rangle\ d_1)$ strict $(\mathsf{loop}\langle\infty\rangle\ d_2)$. If, on the other hand $ll.d_1 \cap ll.d_2 = \emptyset$, the case is equivalent to $(\mathsf{loop}\langle\infty\rangle\ d_1)$ par $(\mathsf{loop}\langle\infty\rangle\ d_2)$. The difficult situation is when $ll.d_1 \neq ll.d_2$ and $ll.d_1 \cap ll.d_2 \neq \emptyset$, because d_1 will block some, but not all of the lifelines, in d_2. However, this means that the result of $(\mathsf{loop}\langle\infty\rangle\ d_1)$ seq $(\mathsf{loop}\langle\infty\rangle\ d_2)$ is the behavior of $\mathsf{loop}\langle\infty\rangle\ d_1$ merged with the behavior of $\mathsf{loop}\langle\infty\rangle\ d_2$ not blocked by $\mathsf{loop}\langle\infty\rangle\ d_1$, which easily can be handled.

The case of a infinite loop inside the high-level operators refuse, assert, alt and xalt poses no problems as its result is just the execution of a silent event before execution of the loop itself. The diagram $\mathsf{loop}\langle n\rangle$ $(\mathsf{loop}\langle\infty\rangle\ d)$ is equivalent to $\mathsf{loop}\langle\infty\rangle\ d$, because the first iteration of the outer loop will block all subsequent iterations.

Finally, consider a diagram of the form $\mathsf{loop}\langle\infty\rangle$ $(d_1$ seq $(\mathsf{loop}\langle\infty\rangle\ d_2)$ seq $d_3)$. The behavior characterized by this diagram is actually the same behavior as characterized by a diagram d_1 seq $((\mathsf{loop}\langle\infty\rangle\ d_2)$ par $(\mathsf{loop}\langle\infty\rangle\ (d'_3$ seq $d'_1))$ where d'_1 and d'_3 are the parts of d_1 and d_3 not blocked by $\mathsf{loop}\langle\infty\rangle\ d_2$.

Because diagrams with nested infinite loops can be reduced to diagrams without nested infinite loops, and because composition of diagrams with infinite loop can be accounted for, there are strong indications that the soundness and completeness results carry over to these more complex situations.

Appendix B

The normal form used in the "all traces" execution strategy

In the "all traces" meta-strategy for execution of sequence diagrams defined in section 9.2, a normal form for the sequence diagrams is applied. This appendix provides additional information on this normal form. In section B.1 we provides details on how to obtain the normal of a diagram. In section B.2 we elaborate on the justification for using such a normal form. In section B.3, proofs of the equivalence of the normal form are given.

B.1 Normal form for sequence diagrams

The normal form is such that xalts, if present, must be the most significant operators of the diagram. After xalts, alts are the most significant, and after alts, the refuse operator is the most significant operator. Below these high-level operators, the simple operators seq, par and strict may occur in any order. The "all traces" meta-strategy is not defined for the assert operator, so this operator is not part of the normal form. Neither does the normal form have loops, but finite loops may be represented by the weak sequencing operator seq as we will see below.

For a diagram that contains the operators seq, par, strict, xalt, alt, refuse and loop$\langle n \rangle$ (finite loop), the normal from can be obtained by application of the 24 transformation rules defined below. The rules should be applied from left to right and should be applied repeatedly until no more applications are possible.

1. $\mathsf{loop}\langle n \rangle\ d_1 = \underbrace{d_1\ \mathsf{seq}\ d_1\ \mathsf{seq}\ \cdots\ \mathsf{seq}\ d_1}_{n\ \text{times}}$
2. $d_1\ \mathsf{alt}\ (d_2\ \mathsf{xalt}\ d_3) = (d_1\ \mathsf{alt}\ d_2)\ \mathsf{xalt}\ (d_1\ \mathsf{alt}\ d_3)$
3. $(d_1\ \mathsf{xalt}\ d_2)\ \mathsf{alt}\ d_3 = (d_1\ \mathsf{alt}\ d_3)\ \mathsf{xalt}\ (d_2\ \mathsf{alt}\ d_3)$
4. $d_1\ \mathsf{par}\ (d_2\ \mathsf{xalt}\ d_3) = (d_1\ \mathsf{par}\ d_2)\ \mathsf{xalt}\ (d_1\ \mathsf{par}\ d_3)$
5. $(d_1\ \mathsf{xalt}\ d_2)\ \mathsf{par}\ d_3 = (d_1\ \mathsf{par}\ d_3)\ \mathsf{xalt}\ (d_2\ \mathsf{par}\ d_3)$
6. $d_1\ \mathsf{seq}\ (d_2\ \mathsf{xalt}\ d_3) = (d_1\ \mathsf{seq}\ d_2)\ \mathsf{xalt}\ (d_1\ \mathsf{seq}\ d_3)$
7. $(d_1\ \mathsf{xalt}\ d_2)\ \mathsf{seq}\ d_3 = (d_1\ \mathsf{seq}\ d_3)\ \mathsf{xalt}\ (d_2\ \mathsf{seq}\ d_3)$

8. d_1 strict $(d_2$ xalt $d_3) = (d_1$ strict $d_2)$ xalt $(d_1$ strict $d_3)$
9. $(d_1$ xalt $d_2)$ strict $d_3 = (d_1$ strict $d_3)$ xalt $(d_2$ strict $d_3)$
10. refuse $(d_1$ xalt $d_2) =$ (refuse d_1) xalt (refuse d_2)
11. d_1 par $(d_2$ alt $d_3) = (d_1$ par $d_2)$ alt $(d_1$ par $d_3)$
12. $(d_1$ alt $d_2)$ par $d_3 = (d_1$ par $d_3)$ alt $(d_2$ par $d_3)$
13. d_1 seq $(d_2$ alt $d_3) = (d_1$ seq $d_2)$ alt $(d_1$ seq $d_3)$
14. $(d_1$ alt $d_2)$ seq $d_3 = (d_1$ seq $d_3)$ alt $(d_2$ seq $d_3)$
15. d_1 strict $(d_2$ alt $d_3) = (d_1$ strict $d_2)$ alt $(d_1$ strict $d_3)$
16. $(d_1$ alt $d_2)$ strict $d_3 = (d_1$ strict $d_3)$ alt $(d_2$ strict $d_3)$
17. refuse $(d_1$ alt $d_2) =$ (refuse d_1) alt (refuse d_2)
18. refuse refuse $d_1 =$ refuse d_1
19. d_1 par refuse $d_2 =$ refuse $(d_1$ par $d_2)$
20. (refuse d_1) par $d_2 =$ refuse $(d_1$ par $d_2)$
21. d_1 seq refuse $d_2 =$ refuse $(d_1$ seq $d_2)$
22. (refuse d_1) seq $d_2 =$ refuse $(d_1$ seq $d_2)$
23. d_1 strict refuse $d_2 =$ refuse $(d_1$ strict $d_2)$
24. (refuse d_1) strict $d_2 =$ refuse $(d_1$ strict $d_2)$

Each of these transformations preserves the semantics of the diagram being transformed. This result is stated and proved as theorem 8 on page 332 in section B.3 below.

With respect to the constraint that an event syntactically should occur only once in a diagram, we assume that events are implicitly given a unique identity (similar to the handling of loop in section A.3) if they are duplicated in the transformation to the normal form.

B.2 Justification for the normal form

Consider the diagram

$$d = (e_1 \text{ alt } e_2) \text{ seq } (e_3 \text{ xalt } e_4)$$

where (for simplicity) the events e_1, e_2, e_3, e_4 are all on the same lifeline. The denotation of d is:

$$[\![\, d \,]\!] = \{(\{\langle e_1, e_3\rangle, \langle e_2, e_3\rangle\}, \emptyset), (\{\langle e_1, e_4\rangle, \langle e_2, e_4\rangle\}, \emptyset)\}$$

The operational semantics gives us executions:

$$[\beta_0, (e_1 \text{ alt } e_2) \text{ seq } (e_3 \text{ xalt } e_4)] \xrightarrow{\tau_{alt}} [\beta_0, e_i \text{ seq } (e_3 \text{ xalt } e_4)]$$
$$\xrightarrow{e_i} [\beta_1, e_3 \text{ xalt } e_4] \xrightarrow{\tau_{xalt}} [\beta_1, e_j] \xrightarrow{e_j} [\beta_2, \text{skip}]$$

with $i \in \{1,2\}$ and $j \in \{3,4\}$. If the meta-strategy for all traces is applied on this diagram (without applying the normal form), the execution first branches because of the alt, and then the xalt makes a split of the interaction obligation for each of these executions. Because of this we would get four interaction obligations:

$$\{(\{\langle e_1,e_3\rangle,\langle e_2,e_3\rangle\},\emptyset),(\{\langle e_1,e_4\rangle,\langle e_2,e_4\rangle\},\emptyset),\\ (\{\langle e_1,e_3\rangle,\langle e_2,e_4\rangle\},\emptyset),(\{\langle e_1,e_4\rangle,\langle e_2,e_3\rangle\},\emptyset)\}$$

By applying the normal form we avoid this.

B.3 Proofs of equivalence of the normal form

Definition 8 *Let $O, O' \subseteq \mathcal{O}$ be sets of interaction obligations and let $p, p', n, n' \subseteq \mathcal{H}$ be trace sets. We then define $\gtrsim$, $\uplus$ and $\neg$ as follows:*

$$\begin{aligned} O \gtrsim O' &\stackrel{\text{def}}{=} \{o \gtrsim o' \mid o \in O \wedge o' \in O'\} \\ (p,n) \uplus (p',n') &\stackrel{\text{def}}{=} (p \cup p', n \cup n') \\ O \uplus O' &\stackrel{\text{def}}{=} \{o \uplus o' \mid o \in O \wedge o' \in O'\} \\ \neg(p,n) &\stackrel{\text{def}}{=} (\emptyset, p \cup n) \\ \neg O &\stackrel{\text{def}}{=} \{\neg o \mid o \in O\} \end{aligned}$$

Lemma 22 *Let d_1, d_2 be diagrams. Then*

1. $[\![d_1 \text{ seq } d_2]\!] = [\![d_1]\!] \gtrsim [\![d_2]\!]$
2. $[\![d_1 \text{ alt } d_2]\!] = [\![d_1]\!] \uplus [\![d_2]\!]$
3. $[\![\text{refuse } d_1]\!] = \neg [\![d_1]\!]$

Proof of lemma 22

⟨1⟩1. PROVE: $[\![d_1 \text{ seq } d_2]\!] = [\![d_1]\!] \gtrsim [\![d_2]\!]$
PROOF: $[\![d_1 \text{ seq } d_2]\!]$
$= \{o_1 \gtrsim o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\}$ (Def. seq)
$= [\![d_1]\!] \gtrsim [\![d_2]\!]$ (Def. $\gtrsim$)

⟨1⟩2. PROVE: $[\![d_1 \text{ alt } d_2]\!] = [\![d_1]\!] \uplus [\![d_2]\!]$
PROOF: $[\![d_1 \text{ alt } d_2]\!]$
$= \{(p_1 \cup p_2, n_1 \cup n_2) \mid (p_1,p_2) \in [\![d_1]\!] \wedge (p_2,n_2) \in [\![d_2]\!]\}$ (Def. alt)
$= \{(p_1,n_1) \uplus (p_2,n_2) \mid (p_1,p_2) \in [\![d_1]\!] \wedge (p_2,n_2) \in [\![d_2]\!]\}$ (Def. $\uplus$)
$= [\![d_1]\!] \uplus [\![d_2]\!]$ (Def. $\uplus$)

⟨1⟩3. PROVE: $[\![\text{refuse } d_1]\!] = \neg [\![d_1]\!]$
PROOF: $[\![\text{refuse } d_1]\!]$
$= \{(\emptyset, p \cup n) \mid (p,n) \in [\![d_1]\!]\}$ (Def. refuse)
$= \{\neg(p,n) \mid (p,n) \in [\![d_1]\!]\}$ (Def. $\neg$)
$= \neg [\![d_1]\!]$ (Def. $\neg$)

⟨1⟩4. Q.E.D.
PROOF: ⟨1⟩1, ⟨1⟩2 and ⟨1⟩3.

□

Lemma 23 $o_1 \| (o_2 \uplus o_3) = (o_1 \| o_2) \uplus (o_1 \| o_3)$

Proof of lemma 23

PROVE: $o_1 \| (o_2 \uplus o_3) = (o_1 \| o_2) \uplus (o_1 \| o_3)$

PROOF:

$$
\begin{array}{lll}
 & o_1 \| (o_2 \uplus o_3) & \\
= & (p_1, n_1) \| ((p_2, n_2) \uplus (p_3, n_3)) & \\
= & (p_1, n_1) \| (p_2 \cup p_3, n_2 \cup n_3) & \text{(Def. } \uplus\text{)} \\
= & (p_1 \| (p_2 \cup p_3), & \\
 & n_1 \| (p_2 \cup p_3) \cup n_1 \| (n_2 \cup n_3) \cup p_1 \| (n_2 \cup n_3)) & \text{(Def. } \|\text{)} \\
= & (p_1 \| p_2 \cup p_1 \| p_3, & \\
 & n_1 \| p_2 \cup n_1 \| p_3 \cup n_1 \| n_2 \cup n_1 \| n_3 \cup p_1 \| n_2 \cup p_1 \| n_3) & (1) \\
= & (p_1 \| p_2 \cup p_1 \| p_3, & \\
 & (n_1 \| p_2 \cup n_1 \| n_2 \cup p_1 \| n_2) \cup (n_1 \| p_3 \cup n_1 \| n_3 \cup p_1 \| n_3)) & (2) \\
= & (p_1 \| p_2, n_1 \| p_2 \cup n_1 \| n_2 \cup p_1 \| n_2) & \\
 & \uplus\ (p_1 \| p_3, n_1 \| p_3 \cup n_1 \| n_3 \cup p_1 \| n_3) & \text{(Def. } \uplus\text{)} \\
= & ((p_1, n_1) \| (p_2, n_2)) \uplus ((p_1, n_1) \| (p_3, n_3)) & \text{(Def. } \|\text{)} \\
= & (o_1 \| o_2) \uplus (o_1 \| o_3) &
\end{array}
$$

Notes:

1. $s_1 \| (s_2 \cup s_3) = (s_1 \| s_2) \cup (s_1 \| s_3)$ [70, Lemma 12].
2. Associativity and commutativity of $\cup$.

□

Lemma 24 $o_1 \succsim (o_2 \uplus o_3) = (o_1 \succsim o_2) \uplus (o_1 \succsim o_3)$

Proof of lemma 24

PROVE: $o_1 \succsim (o_2 \uplus o_3) = (o_1 \succsim o_2) \uplus (o_1 \succsim o_3)$

PROOF:

$$
\begin{array}{lll}
 & o_1 \succsim (o_2 \uplus o_3) & \\
= & (p_1, n_1) \succsim ((p_2, n_2) \uplus (p_3, n_3)) & \\
= & (p_1, n_1) \succsim (p_2 \cup p_3, n_2 \cup n_3) & \text{(Def. } \uplus\text{)} \\
= & (p_1 \succsim (p_2 \cup p_3), & \\
 & n_1 \succsim (p_2 \cup p_3) \cup n_1 \succsim (n_2 \cup n_3) \cup p_1 \succsim (n_2 \cup n_3)) & \text{(Def. } \succsim\text{)} \\
= & (p_1 \succsim p_2 \cup p_1 \succsim p_3, & \\
 & n_1 \succsim p_2 \cup n_1 \succsim p_3 \cup n_1 \succsim n_2 \cup n_1 \succsim n_3 \cup p_1 \succsim n_2 \cup p_1 \succsim n_3) & (1) \\
= & (p_1 \succsim p_2 \cup p_1 \succsim p_3, & \\
 & (n_1 \succsim p_2 \cup n_1 \succsim n_2 \cup p_1 \succsim n_2) \cup (n_1 \succsim p_3 \cup n_1 \succsim n_3 \cup p_1 \succsim n_3)) & (2) \\
= & (p_1 \succsim p_2, n_1 \succsim p_2 \cup n_1 \succsim n_2 \cup p_1 \succsim n_2) & \\
 & \uplus\ (p_1 \succsim p_3, n_1 \succsim p_3 \cup n_1 \succsim n_3 \cup p_1 \succsim n_3) & \text{(Def. } \uplus\text{)} \\
= & ((p_1, n_1) \succsim (p_2, n_2)) \uplus ((p_1, n_1) \succsim (p_3, n_3)) & \text{(Def. } \succsim\text{)} \\
= & (o_1 \succsim o_2) \uplus (o_1 \succsim o_3) &
\end{array}
$$

Notes:

1. $s_1 \succsim (s_2 \cup s_3) = (s_1 \succsim s_2) \cup (s_1 \succsim s_3)$ [70, Lemma 14].
2. Associativity and commutativity of $\cup$.

□

Lemma 25 $(o_1 \uplus o_2) \succsim o_3 = (o_1 \succsim o_3) \uplus (o_2 \succsim o_3)$

Proof of lemma 25

PROVE: $(o_1 \uplus o_2) \succsim o_3 = (o_1 \succsim o_3) \uplus (o_2 \succsim o_3)$

PROOF:

$$
\begin{aligned}
& (o_1 \uplus o_2) \succsim o_3) \\
= \;& ((p_1, n_1) \uplus (p_2, n_2)) \succsim (p_3, n_3)) \\
= \;& (p_1 \cup p_2, n_1 \cup n_2) \succsim (p_3, n_3) && \text{(Def. } \uplus\text{)} \\
= \;& ((p_1 \cup p_2) \succsim p_3, \\
& (n_1 \cup n_2) \succsim p_3 \cup (n_1 \cup n_2) \succsim n_3 \cup (p_1 \cup p_2) \succsim n_3) && \text{(Def. } \succsim\text{)} \\
= \;& (p_1 \succsim p_3 \cup p_2 \succsim p_3, \\
& n_1 \succsim p_3 \cup n_2 \succsim p_3 \cup n_1 \succsim n_3 \cup n_2 \succsim n_3 \cup p_1 \succsim n_3 \cup p_2 \succsim n_3) && (1) \\
= \;& (p_1 \succsim p_3 \cup p_2 \succsim p_3, \\
& (n_1 \succsim p_3 \cup n_1 \succsim n_3 \cup p_1 \succsim n_3) \cup (n_2 \succsim p_3 \cup n_2 \succsim n_3 \cup p_2 \succsim n_3)) && (2) \\
= \;& (p_1 \succsim p_3, n_1 \succsim p_3 \cup n_1 \succsim n_3 \cup p_1 \succsim n_3) \\
& \uplus (p_2 \succsim p_3, n_2 \succsim p_3 \cup n_2 \succsim n_3 \cup p_2 \succsim n_3) && \text{(Def. } \uplus\text{)} \\
= \;& ((p_1, n_1) \succsim (p_3, n_3)) \uplus ((p_2, n_2) \succsim (p_3, n_3)) && \text{(Def. } \succsim\text{)} \\
= \;& (o_1 \succsim o_3) \uplus (o_2 \succsim o_3)
\end{aligned}
$$

Notes:

1. $(s_1 \cup s_2) \succsim s_3 = (s_1 \succsim s_3) \cup (s_2 \succsim s_3)$ [70, Lemma 15].
2. Associativity and commutativity of $\cup$.

□

Lemma 26 $s_1 \succ (s_2 \cup s_3) = (s_1 \succ s_2) \cup (s_1 \succ s_3)$

Proof of lemma 26

PROVE: $s_1 \succ (s_2 \cup s_3) = (s_1 \succ s_2) \cup (s_1 \succ s_3)$

PROOF:

$$
\begin{aligned}
& s_1 \succ (s_2 \cup s_3) \\
= \;& \{h \in \mathcal{H} \,|\, \exists h_1 \in s_1, h_2 \in (s_2 \cup s_3) : h = h_1 \frown h_2\} && \text{(Def. } \succ\text{)} \\
= \;& \{h \in \mathcal{H} \,|\, \exists h_1, h_2 : h_1 \in s_1 \wedge h_2 \in (s_2 \cup s_3) \wedge h = h_1 \frown h_2\} && (1) \\
= \;& \{h \in \mathcal{H} \,|\, \exists h_1, h_2 : h_1 \in s_1 \wedge (h_2 \in s_2 \vee h_2 \in s_3) \wedge h = h_1 \frown h_2\} && (2) \\
= \;& \{h \in \mathcal{H} \,|\, \exists h_1, h_2 : (h_1 \in s_1 \wedge h_2 \in s_2 \wedge h = h_1 \frown h_2) \\
& \qquad \vee (h_1 \in s_1 \wedge h_2 \in s_3 \wedge h = h_1 \frown h_2)\} && (3) \\
= \;& \{h \in \mathcal{H} \,|\, \exists h_1, h_2 : (h_1 \in s_1 \wedge h_2 \in s_2 \wedge h = h_1 \frown h_2) \\
& \qquad \vee \exists h_1, h_2 : (h_1 \in s_1 \wedge h_2 \in s_3 \wedge h = h_1 \frown h_2)\} && (4) \\
= \;& \{h \in \mathcal{H} \,|\, \exists h_1, h_2 : (h_1 \in s_1 \wedge h_2 \in s_2 \wedge h = h_1 \frown h_2)\} \\
& \cup \{h \in \mathcal{H} \,|\, \exists h_1, h_2 : (h_1 \in s_1 \wedge h_2 \in s_3 \wedge h = h_1 \frown h_2)\} && (5) \\
= \;& \{h \in \mathcal{H} \,|\, \exists h_1 \in s_1, h_2 \in s_2 : h = h_1 \frown h_2)\} \\
& \cup \{h \in \mathcal{H} \,|\, \exists h_1 \in s_1, h_2 \in s_3 : h = h_1 \frown h_2)\} && (1) \\
= \;& (s_1 \succ s_2) \cup (s_1 \succ s_3) && \text{(Def. } \succ\text{)}
\end{aligned}
$$

Notes:

1. $\exists x \in X : P(x) \Leftrightarrow \exists x : x \in X \wedge P(x)$.
2. Set theory: $u \in A \cup B \Leftrightarrow u \in A \vee u \in B$.
3. DeMorgan: $(A \vee B) \wedge C \Leftrightarrow (A \wedge C) \vee (B \wedge C)$.
4. Propositional logic: $\exists x : A(x) \vee B(x) \Leftrightarrow \exists x : A(x) \vee \exists x : B(x)$.
5. Set theory: $\{x \,|\, \phi_1(x) \vee \phi_2(x)\} = \{x \,|\, \phi_1(x)\} \cup \{x \,|\, \phi_2(x)\}$.

□

Lemma 27 $(s_1 \cup s_2) \succ s_3 = (s_1 \succ s_3) \cup (s_2 \succ s_3)$

Proof of lemma 27

PROVE: $(s_1 \cup s_2) \succ s_3 = (s_1 \succ s_3) \cup (s_2 \succ s_3)$

PROOF: $(s_1 \cup s_2) \succ s_3$

$$
\begin{aligned}
&= \{h \in \mathcal{H} \mid \exists h_1 \in (s_1 \cup s_2), h_2 \in s_3 : h = h_1 \frown h_2\} && \text{(Def. } \succ\text{)}\\
&= \{h \in \mathcal{H} \mid \exists h_1, h_2 : h_1 \in (s_1 \cup s_2) \wedge h_2 \in s_3 \wedge h = h_1 \frown h_2\} && (1)\\
&= \{h \in \mathcal{H} \mid \exists h_1, h_2 : (h_1 \in s_1 \vee h_1 \in s_2) \wedge h_2 \in s_3 \wedge h = h_1 \frown h_2\} && (2)\\
&= \{h \in \mathcal{H} \mid \exists h_1, h_2 : (h_1 \in s_1 \wedge h_2 \in s_3 \wedge h = h_1 \frown h_2)\\
&\qquad\qquad \vee (h_1 \in s_2 \wedge h_2 \in s_3 \wedge h = h_1 \frown h_2)\} && (3)\\
&= \{h \in \mathcal{H} \mid \exists h_1, h_2 : (h_1 \in s_1 \wedge h_2 \in s_3 \wedge h = h_1 \frown h_2)\\
&\qquad\qquad \vee \exists h_1, h_2 : (h_1 \in s_2 \wedge h_2 \in s_3 \wedge h = h_1 \frown h_2)\} && (4)\\
&= \{h \in \mathcal{H} \mid \exists h_1, h_2 : (h_1 \in s_2 \wedge h_2 \in s_3 \wedge h = h_1 \frown h_2)\}\\
&\quad \cup \{h \in \mathcal{H} \mid \exists h_1, h_2 : (h_1 \in s_2 \wedge h_2 \in s_3 \wedge h = h_1 \frown h_2)\} && (5)\\
&= \{h \in \mathcal{H} \mid \exists h_1 \in s_1, h_2 \in s_3 : h = h_1 \frown h_2)\}\\
&\quad \cup \{h \in \mathcal{H} \mid \exists h_1 \in s_2, h_2 \in s_3 : h = h_1 \frown h_2)\} && (1)\\
&= (s_1 \succ s_3) \cup (s_2 \succ s_3) && \text{(Def. } \succ\text{)}
\end{aligned}
$$

Notes:

1. $\exists x \in X : P(x) \Leftrightarrow \exists x : x \in X \wedge P(x)$.
2. Set theory: $u \in A \cup B \Leftrightarrow u \in A \vee u \in B$.
3. DeMorgan: $(A \vee B) \wedge C \Leftrightarrow (A \wedge C) \vee (B \wedge C)$.
4. Propositional logic: $\exists x : A(x) \vee B(x) \Leftrightarrow \exists x : A(x) \vee \exists x : B(x)$.
5. Set theory: $\{x \mid \phi_1(x) \vee \phi_2(x)\} = \{x \mid \phi_1(x)\} \cup \{x \mid \phi_2(x)\}$.

□

Lemma 28 $o_1 \succ (o_2 \uplus o_3) = (o_1 \succ o_2) \uplus (o_1 \succ o_3)$

Proof of lemma 28

PROVE: $o_1 \succ (o_2 \uplus o_3) = (o_1 \succ o_2) \uplus (o_1 \succ o_3)$

PROOF: $o_1 \succ (o_2 \uplus o_3)$

$$
\begin{aligned}
&= (p_1, n_1) \succ ((p_2, n_2) \uplus (p_3, n_3))\\
&= (p_1, n_1) \succ (p_2 \cup p_3, n_2 \cup n_3) && \text{(Def. } \uplus\text{)}\\
&= (p_1 \succ (p_2 \cup p_3),\\
&\quad n_1 \succ (p_2 \cup p_3) \cup n_1 \succ (n_2 \cup n_3) \cup p_1 \succ (n_2 \cup n_3)) && \text{(Def. } \succ\text{)}\\
&= (p_1 \succ p_2 \cup p_1 \succ p_3,\\
&\quad n_1 \succ p_2 \cup n_1 \succ p_3 \cup n_1 \succ n_2 \cup n_1 \succ n_3 \cup p_1 \succ n_2 \cup p_1 \succ n_3) && (1)\\
&= (p_1 \succ p_2 \cup p_1 \succ p_3,\\
&\quad (n_1 \succ p_2 \cup n_1 \succ n_2 \cup p_1 \succ n_2) \cup (n_1 \succ p_3 \cup n_1 \succ n_3 \cup p_1 \succ n_3)) && (2)\\
&= (p_1 \succ p_2, n_1 \succ p_2 \cup n_1 \succ n_2 \cup p_1 \succ n_2)\\
&\quad \uplus (p_1 \succ p_3, n_1 \succ p_3 \cup n_1 \succ n_3 \cup p_1 \succ n_3) && \text{(Def. } \uplus\text{)}\\
&= ((p_1, n_1) \succ (p_2, n_2)) \uplus ((p_1, n_1) \succ (p_3, n_3)) && \text{(Def. } \succ\text{)}\\
&= (o_1 \succ o_2) \uplus (o_1 \succ o_3)
\end{aligned}
$$

Notes:

1. $s_1 \succ (s_2 \cup s_3) = (s_1 \succ s_2) \cup (s_1 \succ s_3)$ (Lemma 26).
2. Associativity and commutativity of $\cup$.

□

Lemma 29 $(o_1 \uplus o_2) \succ o_3 = (o_1 \succ o_3) \uplus (o_2 \succ o_3)$

Proof of lemma 29

PROVE: $(o_1 \uplus o_2) \succ o_3 = (o_1 \succ o_3) \uplus (o_2 \succ o_3)$

PROOF:

$$
\begin{array}{lll}
 & (o_1 \uplus o_2) \succ o_3) & \\
= & ((p_1, n_1) \uplus (p_2, n_2)) \succ (p_3, n_3)) & \\
= & (p_1 \cup p_2, n_1 \cup n_2) \succ (p_3, n_3) & \text{(Def. } \uplus\text{)} \\
= & ((p_1 \cup p_2) \succ p_3, & \\
 & (n_1 \cup n_2) \succ p_3 \cup (n_1 \cup n_2) \succ n_3 \cup (p_1 \cup p_2) \succ n_3) & \text{(Def. } \succ\text{)} \\
= & (p_1 \succ p_3 \cup p_2 \succ p_3, & \\
 & n_1 \succ p_3 \cup n_2 \succ p_3 \cup n_1 \succ n_3 \cup n_2 \succ n_3 \cup p_1 \succ n_3 \cup p_2 \succ n_3) & (1) \\
= & (p_1 \succ p_3 \cup p_2 \succ p_3, & \\
 & (n_1 \succ p_3 \cup n_1 \succ n_3 \cup p_1 \succ n_3) \cup (n_2 \succ p_3 \cup n_2 \succ n_3 \cup p_2 \succ n_3)) & (2) \\
= & (p_1 \succ p_3, n_1 \succ p_3 \cup n_1 \succ n_3 \cup p_1 \succ n_3) & \\
 & \uplus\ (p_2 \succ p_3, n_2 \succ p_3 \cup n_2 \succ n_3 \cup p_2 \succ n_3) & \text{(Def. } \uplus\text{)} \\
= & ((p_1, n_1) \succ (p_3, n_3)) \uplus ((p_2, n_2) \succ (p_3, n_3)) & \text{(Def. } \succ\text{)} \\
= & (o_1 \succ o_3) \uplus (o_2 \succ o_3) &
\end{array}
$$

Notes:

1. $(s_1 \cup s_2) \succ s_3 = (s_1 \succ s_3) \cup (s_2 \succ s_3)$ (Lemma 27).
2. Associativity and commutativity of $\cup$.

□

Lemma 30 $\neg(o_1 \uplus o_2) = \neg o_1 \uplus \neg o_2$

Proof of lemma 30

PROVE: $\neg(o_1 \uplus o_2) = \neg o_1 \uplus \neg o_2$

PROOF:

$$
\begin{array}{lll}
 & \neg(o_1 \uplus o_2) & \\
= & \neg((p_1, n_1) \uplus (p_2, n_2)) & \\
= & \neg(p_1 \cup p_2, n_1 \cup n_2) & \text{(Def. } \uplus\text{)} \\
= & (\emptyset, p_1 \cup p_2 \cup n_1 \cup n_2) & \text{(Def. } \neg\text{)} \\
= & (\emptyset, (p_1 \cup n_1) \cup (p_2 \cup n_2)) & (1) \\
= & (\emptyset, p_1 \cup n_1) \uplus (\emptyset, p_2 \cup n_2) & \text{(Def. } \uplus\text{)} \\
= & \neg(p_1, n_1) \uplus \neg(p_2, n_2) & \text{(Def. } \neg\text{)} \\
= & \neg o_1 \uplus \neg o_2 &
\end{array}
$$

Notes:

1. Associativity and commutativity of $\cup$.

□

Lemma 31 $\neg\neg o = \neg o$

Proof of lemma 31

PROVE: $\neg\neg o = \neg o$

PROOF:

$$\begin{aligned}
& \neg\neg o \\
= \; & \neg\neg(p, n) \\
= \; & \neg(\emptyset, p \cup n) && \text{(Def. } \neg\text{)} \\
= \; & (\emptyset, \emptyset \cup p \cup n) && \text{(Def. } \neg\text{)} \\
= \; & (\emptyset, p \cup n) \\
= \; & \neg(p, n) && \text{(Def. } \neg\text{)} \\
= \; & \neg o
\end{aligned}$$

□

Lemma 32 $o_1 \| \neg o_2 = \neg(o_1 \| o_2)$

Proof of lemma 32

PROVE: $o_1 \| \neg o_2 = \neg(o_1 \| o_2)$

PROOF:

$$\begin{aligned}
& o_1 \| \neg o_2 \\
= \; & (p_1, n_1) \| \neg(p_2, n_2) \\
= \; & (p_1, n_1) \| (\emptyset, p_2 \cup n_2) && \text{(Def. } \neg\text{)} \\
= \; & (p_1 \| \emptyset, n_1 \| \emptyset \cup n_1 \| (p_2 \cup n_2) \cup p_1 \| (p_2 \cup n_2)) && \text{(Def. } \|\text{)} \\
= \; & (\emptyset, n_1 \| (p_2 \cup n_2) \cup p_1 \| (p_2 \cup n_2)) && (1) \\
= \; & (\emptyset, n_1 \| p_2 \cup n_1 \| n_2 \cup p_1 \| p_2 \cup p_1 \| n_2) && (2) \\
= \; & (\emptyset, p_1 \| p_2 \cup n_1 \| p_2 \cup n_1 \| n_2 \cup p_1 \| n_2) && (3) \\
= \; & \neg(p_1 \| p_2, n_1 \| p_2 \cup n_1 \| n_2 \cup p_1 \| n_2) && \text{(Def. } \neg\text{)} \\
= \; & \neg((p_1, n_1) \| (p_2, n_2)) && \text{(Def. } \|\text{)} \\
= \; & \neg(o_1 \| o_2)
\end{aligned}$$

Notes:

1. $s \| \emptyset = \emptyset \| s = \emptyset$.
2. $s_1 \| (s_2 \cup s_3) = (s_1 \| s_2) \cup (s_1 \| s_3)$ [70, Lemma 12].
3. Associativity and commutativity of $\cup$.

□

Lemma 33 $o_1 \succsim \neg o_2 = \neg(o_1 \succsim o_2)$

Proof of lemma 33

PROVE: $o_1 \succsim \neg o_2 = \neg(o_1 \succsim o_2)$

PROOF:

$$\begin{aligned}
& o_1 \succsim \neg o_2 \\
= \; & (p_1, n_1) \succsim \neg(p_2, n_2) \\
= \; & (p_1, n_1) \succsim (\emptyset, p_2 \cup n_2) && \text{(Def. } \neg\text{)} \\
= \; & (p_1 \succsim \emptyset, n_1 \succsim \emptyset \cup n_1 \succsim (p_2 \cup n_2) \cup p_1 \succsim (p_2 \cup n_2)) && \text{(Def. } \succsim\text{)} \\
= \; & (\emptyset, n_1 \succsim (p_2 \cup n_2) \cup p_1 \succsim (p_2 \cup n_2)) && (1) \\
= \; & (\emptyset, n_1 \succsim p_2 \cup n_1 \succsim n_2 \cup p_1 \succsim p_2 \cup p_1 \succsim n_2) && (2) \\
= \; & (\emptyset, p_1 \succsim p_2 \cup n_1 \succsim p_2 \cup n_1 \succsim n_2 \cup p_1 \succsim n_2) && (3) \\
= \; & \neg(p_1 \succsim p_2, n_1 \succsim p_2 \cup n_1 \succsim n_2 \cup p_1 \succsim n_2) && \text{(Def. } \neg\text{)} \\
= \; & \neg((p_1, n_1) \succsim (p_2, n_2)) && \text{(Def. } \succsim\text{)} \\
= \; & \neg(o_1 \succsim o_2)
\end{aligned}$$

Notes:

1. $s \succsim \emptyset = \emptyset \succsim s = \emptyset$.

2. $s_1 \succsim (s_2 \cup s_3) = (s_1 \succsim s_2) \cup (s_1 \succsim s_3)$ [70, Lemma 14].
3. Associativity and commutativity of $\cup$.

□

Lemma 34 $(\neg o_1) \succsim o_2 = \neg(o_1 \succsim o_2)$

Proof of lemma 34

PROVE: $(\neg o_1) \succsim o_2 = \neg(o_1 \succsim o_2)$

PROOF:

$$
\begin{array}{lll}
 & (\neg o_1) \succsim o_2 & \\
= & (\neg(p_1, n_1)) \succsim (p_2, n_2) & \\
= & (\emptyset, p_1 \cup n_1) \succsim (p_2, n_2) & \text{(Def. } \neg) \\
= & (\emptyset \succsim p_2, (p_1 \cup n_1) \succsim p_2 \cup (p_1 \cup n_1) \succsim n_2 \cup \emptyset \succsim n_2) & \text{(Def. } \succsim) \\
= & (\emptyset, (p_1 \cup n_1) \succsim p_2 \cup (p_1 \cup n_1) \succsim n_2) & (1) \\
= & (\emptyset, p_1 \succsim p_2 \cup n_1 \succsim p_2 \cup p_1 \succsim n_2 \cup n_1 \succsim n_2) & (2) \\
= & (\emptyset, p_1 \succsim p_2 \cup n_1 \succsim p_2 \cup n_1 \succsim n_2 \cup p_1 \succsim n_2) & (3) \\
= & \neg(p_1 \succsim p_2, n_1 \succsim p_2 \cup n_1 \succsim n_2 \cup p_1 \succsim n_2) & \text{(Def. } \neg) \\
= & \neg((p_1, n_1) \succsim (p_2, n_2)) & \text{(Def. } \succsim) \\
= & \neg(o_1 \succsim o_2) &
\end{array}
$$

Notes:
1. $s \succsim \emptyset = \emptyset \succsim s = \emptyset$.
2. $(s_1 \cup s_2) \succsim s_3 = (s_1 \succsim s_3) \cup (s_2 \succsim s_3)$ [70, Lemma 15].
3. Associativity and commutativity of $\cup$.

□

Lemma 35 $o_1 \succ \neg o_2 = \neg(o_1 \succ o_2)$

Proof of lemma 35

PROVE: $o_1 \succ \neg o_2 = \neg(o_1 \succ o_2)$

PROOF:

$$
\begin{array}{lll}
 & o_1 \succ \neg o_2 & \\
= & (p_1, n_1) \succ \neg(p_2, n_2) & \\
= & (p_1, n_1) \succ (\emptyset, p_2 \cup n_2) & \text{(Def. } \neg) \\
= & (p_1 \succ \emptyset, n_1 \succ \emptyset \cup n_1 \succ (p_2 \cup n_2) \cup p_1 \succ (p_2 \cup n_2)) & \text{(Def. } \succ) \\
= & (\emptyset, n_1 \succ (p_2 \cup n_2) \cup p_1 \succ (p_2 \cup n_2)) & (1) \\
= & (\emptyset, n_1 \succ p_2 \cup n_1 \succ n_2 \cup p_1 \succ p_2 \cup p_1 \succ n_2) & (2) \\
= & (\emptyset, p_1 \succ p_2 \cup n_1 \succ p_2 \cup n_1 \succ n_2 \cup p_1 \succ n_2) & (3) \\
= & \neg(p_1 \succ p_2, n_1 \succ p_2 \cup n_1 \succ n_2 \cup p_1 \succ n_2) & \text{(Def. } \neg) \\
= & \neg((p_1, n_1) \succ (p_2, n_2)) & \text{(Def. } \succ) \\
= & \neg(o_1 \succ o_2) &
\end{array}
$$

Notes:
1. $s \succ \emptyset = \emptyset \succ s = \emptyset$.
2. $s_1 \succ (s_2 \cup s_3) = (s_1 \succ s_2) \cup (s_1 \succ s_3)$ (Lemma 26).
3. Associativity and commutativity of $\cup$.

□

Lemma 36 $(\neg o_1) \succ o_2 = \neg(o_1 \succ o_2)$

Proof of lemma 36

PROVE: $(\neg o_1) \succ o_2 = \neg(o_1 \succ o_2)$

PROOF:

$$\begin{array}{lll}
 & (\neg o_1) \succ o_2 & \\
= & (\neg(p_1, n_1)) \succ (p_2, n_2) & \\
= & (\emptyset, p_1 \cup n_1) \succ (p_2, n_2) & (\text{Def. } \neg) \\
= & (\emptyset \succ p_2, (p_1 \cup n_1) \succ p_2 \cup (p_1 \cup n_1) \succ n_2 \cup \emptyset \succ n_2) & (\text{Def. } \succ) \\
= & (\emptyset, (p_1 \cup n_1) \succ p_2 \cup (p_1 \cup n_1) \succ n_2) & (1) \\
= & (\emptyset, p_1 \succ p_2 \cup n_1 \succ p_2 \cup p_1 \succ n_2 \cup n_1 \succ n_2) & (2) \\
= & (\emptyset, p_1 \succ p_2 \cup n_1 \succ p_2 \cup n_1 \succ n_2 \cup p_1 \succ n_2) & (3) \\
= & \neg(p_1 \succ p_2, n_1 \succ p_2 \cup n_1 \succ n_2 \cup p_1 \succ n_2) & (\text{Def. } \neg) \\
= & \neg((p_1, n_1) \succ (p_2, n_2)) & (\text{Def. } \succ) \\
= & \neg(o_1 \succ o_2) &
\end{array}$$

Notes:

1. $s \succ \emptyset = \emptyset \succ s = \emptyset$.
2. $(s_1 \cup s_2) \succ s_3 = (s_1 \succ s_3) \cup (s_2 \succ s_3)$ (Lemma 27).
3. Associativity and commutativity of $\cup$.

□

Theorem 8 *Given diagrams d_1, d_2, d_3 we have the following equations:*

1. $[\![\, \mathsf{loop}\langle n\rangle\ d_1 \,]\!] = [\![\underbrace{d_1 \ \mathsf{seq}\ d_1 \ \mathsf{seq}\ \cdots \ \mathsf{seq}\ d_1}_{n \text{ times}}]\!]$
2. $[\![\, d_1 \ \mathsf{alt}\ (d_2 \ \mathsf{xalt}\ d_3) \,]\!] = [\![\, (d_1 \ \mathsf{alt}\ d_2) \ \mathsf{xalt}\ (d_1 \ \mathsf{alt}\ d_3) \,]\!]$
3. $[\![\, (d_1 \ \mathsf{xalt}\ d_2) \ \mathsf{alt}\ d_3 \,]\!] = [\![\, (d_1 \ \mathsf{alt}\ d_3) \ \mathsf{xalt}\ (d_2 \ \mathsf{alt}\ d_3) \,]\!]$
4. $[\![\, d_1 \ \mathsf{par}\ (d_2 \ \mathsf{xalt}\ d_3) \,]\!] = [\![\, (d_1 \ \mathsf{par}\ d_2) \ \mathsf{xalt}\ (d_1 \ \mathsf{par}\ d_3) \,]\!]$
5. $[\![\, (d_1 \ \mathsf{xalt}\ d_2) \ \mathsf{par}\ d_3 \,]\!] = [\![\, (d_1 \ \mathsf{par}\ d_3) \ \mathsf{xalt}\ (d_2 \ \mathsf{par}\ d_3) \,]\!]$
6. $[\![\, d_1 \ \mathsf{seq}\ (d_2 \ \mathsf{xalt}\ d_3) \,]\!] = [\![\, (d_1 \ \mathsf{seq}\ d_2) \ \mathsf{xalt}\ (d_1 \ \mathsf{seq}\ d_3) \,]\!]$
7. $[\![\, (d_1 \ \mathsf{xalt}\ d_2) \ \mathsf{seq}\ d_3 \,]\!] = [\![\, (d_1 \ \mathsf{seq}\ d_3) \ \mathsf{xalt}\ (d_2 \ \mathsf{seq}\ d_3) \,]\!]$
8. $[\![\, d_1 \ \mathsf{strict}\ (d_2 \ \mathsf{xalt}\ d_3) \,]\!] = [\![\, (d_1 \ \mathsf{strict}\ d_2) \ \mathsf{xalt}\ (d_1 \ \mathsf{strict}\ d_3) \,]\!]$
9. $[\![\, (d_1 \ \mathsf{xalt}\ d_2) \ \mathsf{strict}\ d_3 \,]\!] = [\![\, (d_1 \ \mathsf{strict}\ d_3) \ \mathsf{xalt}\ (d_2 \ \mathsf{strict}\ d_3) \,]\!]$
10. $[\![\, \mathsf{refuse}\ (d_1 \ \mathsf{xalt}\ d_2) \,]\!] = [\![\, (\mathsf{refuse}\ d_1) \ \mathsf{xalt}\ (\mathsf{refuse}\ d_2) \,]\!]$
11. $[\![\, d_1 \ \mathsf{par}\ (d_2 \ \mathsf{alt}\ d_3) \,]\!] = [\![\, (d_1 \ \mathsf{par}\ d_2) \ \mathsf{alt}\ (d_1 \ \mathsf{par}\ d_3) \,]\!]$
12. $[\![\, (d_1 \ \mathsf{alt}\ d_2) \ \mathsf{par}\ d_3 \,]\!] = [\![\, (d_1 \ \mathsf{par}\ d_3) \ \mathsf{alt}\ (d_2 \ \mathsf{par}\ d_3) \,]\!]$
13. $[\![\, d_1 \ \mathsf{seq}\ (d_2 \ \mathsf{alt}\ d_3) \,]\!] = [\![\, (d_1 \ \mathsf{seq}\ d_2) \ \mathsf{alt}\ (d_1 \ \mathsf{seq}\ d_3) \,]\!]$
14. $[\![\, (d_1 \ \mathsf{alt}\ d_2) \ \mathsf{seq}\ d_3 \,]\!] = [\![\, (d_1 \ \mathsf{seq}\ d_3) \ \mathsf{alt}\ (d_2 \ \mathsf{seq}\ d_3) \,]\!]$
15. $[\![\, d_1 \ \mathsf{strict}\ (d_2 \ \mathsf{alt}\ d_3) \,]\!] = [\![\, (d_1 \ \mathsf{strict}\ d_2) \ \mathsf{alt}\ (d_1 \ \mathsf{strict}\ d_3) \,]\!]$
16. $[\![\, (d_1 \ \mathsf{alt}\ d_2) \ \mathsf{strict}\ d_3 \,]\!] = [\![\, (d_1 \ \mathsf{strict}\ d_3) \ \mathsf{alt}\ (d_2 \ \mathsf{strict}\ d_3) \,]\!]$

17. $[\![\mathsf{refuse}\ (d_1\ \mathsf{alt}\ d_2)]\!] = [\![(\mathsf{refuse}\ d_1)\ \mathsf{alt}\ (\mathsf{refuse}\ d_2)]\!]$

18. $[\![\mathsf{refuse}\ \mathsf{refuse}\ d_1]\!] = [\![\mathsf{refuse}\ d_1]\!]$

19. $[\![d_1\ \mathsf{par}\ \mathsf{refuse}\ d_2]\!] = [\![\mathsf{refuse}\ (d_1\ \mathsf{par}\ d_2)]\!]$

20. $[\![(\mathsf{refuse}\ d_1)\ \mathsf{par}\ d_2]\!] = [\![\mathsf{refuse}\ (d_1\ \mathsf{par}\ d_2)]\!]$

21. $[\![d_1\ \mathsf{seq}\ \mathsf{refuse}\ d_2]\!] = [\![\mathsf{refuse}\ (d_1\ \mathsf{seq}\ d_2)]\!]$

22. $[\![(\mathsf{refuse}\ d_1)\ \mathsf{seq}\ d_2]\!] = [\![\mathsf{refuse}\ (d_1\ \mathsf{seq}\ d_2)]\!]$

23. $[\![d_1\ \mathsf{strict}\ \mathsf{refuse}\ d_2]\!] = [\![\mathsf{refuse}\ (d_1\ \mathsf{strict}\ d_2)]\!]$

24. $[\![(\mathsf{refuse}\ d_1)\ \mathsf{strict}\ d_2]\!] = [\![\mathsf{refuse}\ (d_1\ \mathsf{strict}\ d_2)]\!]$

Proof of theorem 8

$\langle 1 \rangle 1$. PROVE: $[\![\mathsf{loop}\langle n \rangle\ d_1]\!] = [\![\underbrace{d_1\ \mathsf{seq}\ d_1\ \mathsf{seq}\ \cdots\ \mathsf{seq}\ d_1}_{n\ \text{times}}]\!]$

PROOF:

$$
\begin{array}{lll}
[\![\mathsf{loop}\langle n \rangle\ d_1]\!] & & \\
= \mu_n [\![d_1]\!] & & \text{(Def. loop)} \\
= [\![d_1]\!] \succsim \mu_{n-1} [\![d_1]\!] & & \text{(Def. } \mu\text{)} \\
= [\![d_1]\!] \succsim [\![d_1]\!] \succsim \mu_{n-2} [\![d_1]\!] & & \text{(Def. } \mu\text{)} \\
\qquad \vdots & & \\
= \underbrace{[\![d_1]\!] \succsim [\![d_1]\!] \succsim \cdots \succsim [\![d_1]\!]}_{n-1\ \text{times}} \succsim \mu_1 [\![d_1]\!] & & \text{(Def. } \mu\text{)} \\
= \underbrace{[\![d_1]\!] \succsim [\![d_1]\!] \succsim \cdots \succsim [\![d_1]\!]}_{n-1\ \text{times}} \succsim [\![d_1]\!] & & \text{(Def. } \mu\text{)} \\
= \underbrace{[\![d_1]\!] \succsim [\![d_1]\!] \succsim \cdots \succsim [\![d_1]\!]}_{n\ \text{times}} & & \\
= [\![\underbrace{d_1\ \mathsf{seq}\ d_1\ \mathsf{seq}\ \cdots\ \mathsf{seq}\ d_1}_{n\ \text{times}}]\!] & & \text{(Lemma 22)}
\end{array}
$$

$\langle 1 \rangle 2$. PROVE: $[\![d_1\ \mathsf{alt}\ (d_2\ \mathsf{xalt}\ d_3)]\!] = [\![(d_1\ \mathsf{alt}\ d_2)\ \mathsf{xalt}\ (d_1\ \mathsf{alt}\ d_3)]\!]$

PROOF:

$$
\begin{array}{lll}
[\![d_1\ \mathsf{alt}\ (d_2\ \mathsf{xalt}\ d_3)]\!] & & \\
= \{ o_1 \uplus o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2\ \mathsf{xalt}\ d_3]\!] \} & & \text{(Def. alt)} \\
= \{ o_1 \uplus o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!] \cup [\![d_3]\!] \} & & \text{(Def. xalt)} \\
= \{ o_1 \uplus o_2 \mid o_1 \in [\![d_1]\!] \wedge (o_2 \in [\![d_2]\!] \vee o_2 \in [\![d_3]\!]) \} & & (1) \\
= \{ o_1 \uplus o_2 \mid (o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]) & & \\
\qquad \vee (o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!]) \} & & (2) \\
= \{ o_1 \uplus o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!] \} & & \\
\quad \cup \{ o_1 \uplus o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!] \} & & (3) \\
= [\![d_1\ \mathsf{alt}\ d_2]\!] \cup [\![d_1\ \mathsf{alt}\ d_3]\!] & & \text{(Def. alt)} \\
= [\![(d_1\ \mathsf{alt}\ d_2)\ \mathsf{xalt}\ (d_1\ \mathsf{alt}\ d_3)]\!] & & \text{(Def. xalt)}
\end{array}
$$

Notes:

1. Set theory: $u \in A \cup B \Leftrightarrow u \in A \vee u \in B$.
2. DeMorgan: $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$.
3. Set theory: $\{x \mid \phi_1(x) \vee \phi_2(x)\} = \{x \mid \phi_1(x)\} \cup \{x \mid \phi_2(x)\}$.

$\langle 1 \rangle 3$. PROVE: $[\![(d_1\ \mathsf{xalt}\ d_2)\ \mathsf{alt}\ d_3]\!] = [\![(d_1\ \mathsf{alt}\ d_3)\ \mathsf{xalt}\ (d_2\ \mathsf{alt}\ d_3)]\!]$

PROOF: $[\![(d_1 \text{ xalt } d_2) \text{ alt } d_3]\!]$
$= [\![d_3 \text{ alt } (d_1 \text{ xalt } d_2)]\!]$ (1)
$= [\![(d_3 \text{ alt } d_1) \text{ xalt } (d_3 \text{ alt } d_2)]\!]$ ($\langle 1 \rangle 2$)
$= [\![(d_1 \text{ alt } d_3) \text{ xalt } (d_2 \text{ alt } d_3)]\!]$ (1)

Notes:
1. of alt [70, theorem 1].

$\langle 1 \rangle 4$. PROVE: $[\![d_1 \text{ par } (d_2 \text{ xalt } d_3)]\!] = [\![(d_1 \text{ par } d_2) \text{ xalt } (d_1 \text{ par } d_3)]\!]$

PROOF: $[\![d_1 \text{ par } (d_2 \text{ xalt } d_3)]\!]$
$= \{o_1 \| o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2 \text{ xalt } d_3]\!]\}$ (Def. par)
$= \{o_1 \| o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!] \cup [\![d_3]\!]\}$ (Def. xalt)
$= \{o_1 \| o_2 \mid o_1 \in [\![d_1]\!] \wedge (o_2 \in [\![d_2]\!] \vee o_2 \in [\![d_3]\!])\}$ (1)
$= \{o_1 \| o_2 \mid (o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!])$
$\vee (o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!])\}$ (2)
$= \{o_1 \| o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\}$
$\cup \{o_1 \| o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!]\}$ (3)
$= [\![d_1 \text{ par } d_2]\!] \cup [\![d_1 \text{ par } d_3]\!]$ (Def. par)
$= [\![(d_1 \text{ par } d_2) \text{ xalt } (d_1 \text{ par } d_3)]\!]$ (Def. xalt)

Notes:
1. Set theory: $u \in A \cup B \Leftrightarrow u \in A \vee u \in B$.
2. DeMorgan: $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$.
3. Set theory: $\{x \mid \phi_1(x) \vee \phi_2(x)\} = \{x \mid \phi_1(x)\} \cup \{x \mid \phi_2(x)\}$.

$\langle 1 \rangle 5$. PROVE: $[\![(d_1 \text{ xalt } d_2) \text{ par } d_3]\!] = [\![(d_1 \text{ par } d_3) \text{ xalt } (d_2 \text{ par } d_3)]\!]$

PROOF: $[\![(d_1 \text{ xalt } d_2) \text{ par } d_3]\!]$
$= [\![d_3 \text{ par } (d_1 \text{ xalt } d_2)]\!]$ (1)
$= [\![(d_3 \text{ par } d_1) \text{ xalt } (d_3 \text{ par } d_2)]\!]$ ($\langle 1 \rangle 4$)
$= [\![(d_1 \text{ par } d_3) \text{ xalt } (d_2 \text{ par } d_3)]\!]$ (1)

Notes:
1. Commutativity of par [70, theorem 5].

$\langle 1 \rangle 6$. PROVE: $[\![d_1 \text{ seq } (d_2 \text{ xalt } d_3)]\!] = [\![(d_1 \text{ seq } d_2) \text{ xalt } (d_1 \text{ seq } d_3)]\!]$

PROOF: $[\![d_1 \text{ seq } (d_2 \text{ xalt } d_3)]\!]$
$= \{o_1 \succsim o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2 \text{ xalt } d_3]\!]\}$ (Def. seq)
$= \{o_1 \succsim o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!] \cup [\![d_3]\!]\}$ (Def. xalt)
$= \{o_1 \succsim o_2 \mid o_1 \in [\![d_1]\!] \wedge (o_2 \in [\![d_2]\!] \vee o_2 \in [\![d_3]\!])\}$ (1)
$= \{o_1 \succsim o_2 \mid (o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!])$
$\vee (o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!])\}$ (2)
$= \{o_1 \succsim o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\}$
$\cup \{o_1 \succsim o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!]\}$ (3)
$= [\![d_1 \text{ seq } d_2]\!] \cup [\![d_1 \text{ seq } d_3]\!]$ (Def. seq)
$= [\![(d_1 \text{ seq } d_2) \text{ xalt } (d_1 \text{ seq } d_3)]\!]$ (Def. xalt)

Notes:
1. Set theory: $u \in A \cup B \Leftrightarrow u \in A \vee u \in B$.
2. DeMorgan: $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$.
3. Set theory: $\{x \mid \phi_1(x) \vee \phi_2(x)\} = \{x \mid \phi_1(x)\} \cup \{x \mid \phi_2(x)\}$.

$\langle 1 \rangle 7$. PROVE: $[\![(d_1 \text{ xalt } d_2) \text{ seq } d_3]\!] = [\![(d_1 \text{ seq } d_3) \text{ xalt } (d_2 \text{ seq } d_3)]\!]$

PROOF: $[\![(d_1 \text{ xalt } d_2) \text{ seq } d_3]\!]$

$= \{o_1 \succsim o_2 \mid o_1 \in [\![d_1 \text{ xalt } d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (Def. seq)

$= \{o_1 \succsim o_2 \mid o_1 \in [\![d_1]\!] \cup [\![d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (Def. xalt)

$= \{o_1 \succsim o_2 \mid (o_1 \in [\![d_1]\!] \vee o_1 \in [\![d_2]\!]) \wedge o_2 \in [\![d_3]\!]\}$ (1)

$= \{o_1 \succsim o_2 \mid (o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!])$
$\quad \vee (o_1 \in [\![d_2]\!] \wedge o_2 \in [\![d_3]\!])\}$ (2)

$= \{o_1 \succsim o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!]\}$
$\quad \cup \{o_1 \succsim o_2 \mid o_1 \in [\![d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (3)

$= [\![d_1 \text{ seq } d_3]\!] \cup [\![d_2 \text{ seq } d_3]\!]$ (Def. seq)

$= [\![(d_1 \text{ seq } d_3) \text{ xalt } (d_2 \text{ seq } d_3)]\!]$ (Def. xalt)

Notes:

1. Set theory: $u \in A \cup B \Leftrightarrow u \in A \vee u \in B$.
2. DeMorgan: $(A \vee B) \wedge C \Leftrightarrow (A \wedge C) \vee (B \wedge C)$.
3. Set theory: $\{x \mid \phi_1(x) \vee \phi_2(x)\} = \{x \mid \phi_1(x)\} \cup \{x \mid \phi_2(x)\}$.

⟨1⟩8. PROVE: $[\![d_1 \text{ strict } (d_2 \text{ xalt } d_3)]\!] = [\![(d_1 \text{ strict } d_2) \text{ xalt } (d_1 \text{ strict } d_3)]\!]$

PROOF: $[\![d_1 \text{ strict } (d_2 \text{ xalt } d_3)]\!]$

$= \{o_1 \succ o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2 \text{ xalt } d_3]\!]\}$ (Def. strict)

$= \{o_1 \succ o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!] \cup [\![d_3]\!]\}$ (Def. xalt)

$= \{o_1 \succ o_2 \mid o_1 \in [\![d_1]\!] \wedge (o_2 \in [\![d_2]\!] \vee o_2 \in [\![d_3]\!])\}$ (1)

$= \{o_1 \succ o_2 \mid (o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!])$
$\quad \vee (o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!])\}$ (2)

$= \{o_1 \succ o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\}$
$\quad \cup \{o_1 \succ o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!]\}$ (3)

$= [\![d_1 \text{ strict } d_2]\!] \cup [\![d_1 \text{ strict } d_3]\!]$ (Def. strict)

$= [\![(d_1 \text{ strict } d_2) \text{ xalt } (d_1 \text{ strict } d_3)]\!]$ (Def. xalt)

Notes:

1. Set theory: $u \in A \cup B \Leftrightarrow u \in A \vee u \in B$.
2. DeMorgan: $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$.
3. Set theory: $\{x \mid \phi_1(x) \vee \phi_2(x)\} = \{x \mid \phi_1(x)\} \cup \{x \mid \phi_2(x)\}$.

⟨1⟩9. PROVE: $[\![(d_1 \text{ xalt } d_2) \text{ strict } d_3]\!] = [\![(d_1 \text{ strict } d_3) \text{ xalt } (d_2 \text{ strict } d_3)]\!]$

PROOF: $[\![(d_1 \text{ xalt } d_2) \text{ strict } d_3]\!]$

$= \{o_1 \succ o_2 \mid o_1 \in [\![d_1 \text{ xalt } d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (Def. strict)

$= \{o_1 \succ o_2 \mid o_1 \in [\![d_1]\!] \cup [\![d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (Def. xalt)

$= \{o_1 \succ o_2 \mid (o_1 \in [\![d_1]\!] \vee o_1 \in [\![d_2]\!]) \wedge o_2 \in [\![d_3]\!]\}$ (1)

$= \{o_1 \succ o_2 \mid (o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!])$
$\quad \vee (o_1 \in [\![d_2]\!] \wedge o_2 \in [\![d_3]\!])\}$ (2)

$= \{o_1 \succ o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!]\}$
$\quad \cup \{o_1 \succ o_2 \mid o_1 \in [\![d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (3)

$= [\![d_1 \text{ strict } d_3]\!] \cup [\![d_2 \text{ strict } d_3]\!]$ (Def. strict)

$= [\![(d_1 \text{ strict } d_3) \text{ xalt } (d_2 \text{ strict } d_3)]\!]$ (Def. xalt)

Notes:

1. Set theory: $u \in A \cup B \Leftrightarrow u \in A \vee u \in B$.
2. DeMorgan: $(A \vee B) \wedge C \Leftrightarrow (A \wedge C) \vee (B \wedge C)$.
3. Set theory: $\{x \mid \phi_1(x) \vee \phi_2(x)\} = \{x \mid \phi_1(x)\} \cup \{x \mid \phi_2(x)\}$.

⟨1⟩10. PROVE: $[\![\text{refuse } (d_1 \text{ xalt } d_2)]\!] = [\![(\text{refuse } d_1) \text{ xalt } (\text{refuse } d_2)]\!]$

PROOF: $[\![\mathsf{refuse}\ (d_1\ \mathsf{xalt}\ d_2)]\!]$
$= \{(\emptyset, p \cup n) \mid (p,n) \in [\![d_1\ \mathsf{xalt}\ d_2]\!]\}$ (Def. refuse)
$= \{(\emptyset, p \cup n) \mid (p,n) \in [\![d_1]\!] \cup [\![d_2]\!]\}$ (Def. xalt)
$= \{(\emptyset, p \cup n) \mid (p,n) \in [\![d_1]\!] \vee (p,n) \in [\![d_2]\!]\}$ (1)
$= \{(\emptyset, p \cup n) \mid (p,n) \in [\![d_1]\!]\}$
$\cup \{(\emptyset, p \cup n) \mid (p,n) \in [\![d_2]\!]\}$ (2)
$= [\![\mathsf{refuse}\ d_1]\!] \cup [\![\mathsf{refuse}\ d_2]\!]$ (Def. refuse)
$= [\![(\mathsf{refuse}\ d_1)\ \mathsf{xalt}\ (\mathsf{refuse}\ d_2)]\!]$ (Def. xalt)

Notes:
1. Set theory: $u \in A \cup B \Leftrightarrow u \in A \vee u \in B$.
2. Set theory: $\{x \mid \phi_1(x) \vee \phi_2(x)\} = \{x \mid \phi_1(x)\} \cup \{x \mid \phi_2(x)\}$.

⟨1⟩11. PROVE: $[\![d_1\ \mathsf{par}\ (d_2\ \mathsf{alt}\ d_3)]\!] = [\![(d_1\ \mathsf{par}\ d_2)\ \mathsf{alt}\ (d_1\ \mathsf{par}\ d_3)]\!]$
PROOF: $[\![d_1\ \mathsf{par}\ (d_2\ \mathsf{alt}\ d_3)]\!]$
$= \{o_1 \| o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2\ \mathsf{alt}\ d_3]\!]\}$ (Def. par)
$= \{o_1 \| o_2 \mid o_1 \in [\![d_1]\!] \wedge$
$o_2 \in \{o_3 \uplus o_4 \mid o_3 \in [\![d_2]\!] \wedge o_4 \in [\![d_3]\!]\}\}$ (Def. alt)
$= \{o_1 \| (o_3 \uplus o_4) \mid o_1 \in [\![d_1]\!] \wedge$
$o_3 \in [\![d_2]\!] \wedge o_4 \in [\![d_3]\!]\}$ (1)
$= \{(o_1 \| o_3) \uplus (o_1 \| o_4) \mid o_1 \in [\![d_1]\!] \wedge$
$o_3 \in [\![d_2]\!] \wedge o_4 \in [\![d_3]\!]\}$ (Lemma 23)
$= \{o_1 \| o_3 \mid o_1 \in [\![d_1]\!] \wedge o_3 \in [\![d_2]\!]\}$
$\uplus \{o_1 \| o_4 \mid o_1 \in [\![d_1]\!] \wedge o_4 \in [\![d_3]\!]\}$ (Def. $\uplus$)
$= [\![d_1\ \mathsf{par}\ d_2]\!] \uplus [\![d_1\ \mathsf{par}\ d_3]\!]$ (Def. par)
$= [\![(d_1\ \mathsf{par}\ d_2)\ \mathsf{alt}\ (d_1\ \mathsf{par}\ d_3)]\!]$ (Lemma 22)

Notes:
1. This can be shown using the set theory operation
$\forall u : [u \in \{\langle x_1, \ldots, x_n \rangle \mid \phi(x_1, \ldots, x_n, Y_1, \ldots, Y_m)\}$
$\Leftrightarrow \exists x_1, \ldots, x_n : u = \langle x_1, \ldots, x_n, Y_1, \ldots, Y_m \rangle \wedge \phi(x_1, \ldots, x_n, Y_1, \ldots, Y_m)]$ like this:
$\{o_1 \| o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in \{o_3 \uplus o_4 \mid o_3 \in [\![d_2]\!] \wedge o_4 \in [\![d_3]\!]\}\}$
$= \{o_1 \| o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 = o_3 \uplus o_4 \wedge o_3 \in [\![d_2]\!] \wedge o_4 \in [\![d_3]\!]\}$
$= \{o_1 \| (o_3 \uplus o_4) \mid o_1 \in [\![d_1]\!] \wedge o_3 \in [\![d_2]\!] \wedge o_4 \in [\![d_3]\!]\}$

⟨1⟩12. PROVE: $[\![(d_1\ \mathsf{alt}\ d_2)\ \mathsf{par}\ d_3]\!] = [\![(d_1\ \mathsf{par}\ d_3)\ \mathsf{alt}\ (d_2\ \mathsf{par}\ d_3)]\!]$
PROOF: $[\![(d_1\ \mathsf{alt}\ d_2)\ \mathsf{par}\ d_3]\!]$
$= [\![d_3\ \mathsf{par}\ (d_1\ \mathsf{alt}\ d_2)]\!]$ (1)
$= [\![(d_3\ \mathsf{par}\ d_1)\ \mathsf{alt}\ (d_3\ \mathsf{par}\ d_2)]\!]$ (⟨1⟩11)
$= [\![(d_1\ \mathsf{par}\ d_3)\ \mathsf{alt}\ (d_2\ \mathsf{par}\ d_3)]\!]$ (1)

Notes:
1. Commutativity of par [70, theorem 5].

⟨1⟩13. PROVE: $[\![d_1\ \mathsf{seq}\ (d_2\ \mathsf{alt}\ d_3)]\!] = [\![(d_1\ \mathsf{seq}\ d_2)\ \mathsf{alt}\ (d_1\ \mathsf{seq}\ d_3)]\!]$

PROOF: $[\![d_1 \text{ seq } (d_2 \text{ alt } d_3)]\!]$

$= \{o_1 \succsim o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2 \text{ alt } d_3]\!]\}$ (Def. seq)

$= \{o_1 \succsim o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in \{o_3 \uplus o_4 \mid o_3 \in [\![d_2]\!] \wedge o_4 \in [\![d_3]\!]\}\}$ (Def. alt)

$= \{o_1 \succsim (o_3 \uplus o_4) \mid o_1 \in [\![d_1]\!] \wedge o_3 \in [\![d_2]\!] \wedge o_4 \in [\![d_3]\!]\}$ (1)

$= \{(o_1 \succsim o_3) \uplus (o_1 \succsim o_4) \mid o_1 \in [\![d_1]\!] \wedge o_3 \in [\![d_2]\!] \wedge o_4 \in [\![d_3]\!]\}$ (Lemma 24)

$= \{o_1 \succsim o_3 \mid o_1 \in [\![d_1]\!] \wedge o_3 \in [\![d_2]\!]\} \uplus \{o_1 \succsim o_4 \mid o_1 \in [\![d_1]\!] \wedge o_4 \in [\![d_3]\!]\}$ (Def. $\uplus$)

$= [\![d_1 \text{ seq } d_2]\!] \uplus [\![d_1 \text{ seq } d_3]\!]$ (Def. seq)

$= [\![(d_1 \text{ seq } d_2) \text{ alt } (d_1 \text{ seq } d_3)]\!]$ (Lemma 22)

Notes:

1. This operation is the same as the one found in ⟨1⟩11.

⟨1⟩14. PROVE: $[\![(d_1 \text{ alt } d_2) \text{ seq } d_3]\!] = [\![(d_1 \text{ seq } d_3) \text{ alt } (d_2 \text{ seq } d_3)]\!]$

PROOF: $[\![(d_1 \text{ alt } d_2) \text{ seq } d_3]\!]$

$= \{o_1 \succsim o_2 \mid o_1 \in [\![d_1 \text{ alt } d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (Def. seq)

$= \{o_1 \succsim o_2 \mid o_1 \in \{o_3 \uplus o_4 \mid o_3 \in [\![d_1]\!] \wedge o_4 \in [\![d_2]\!]\} \wedge o_2 \in [\![d_3]\!]\}$ (Def. alt)

$= \{(o_3 \uplus o_4) \succsim o_2 \mid o_3 \in [\![d_1]\!] \wedge o_4 \in [\![d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (1)

$= \{(o_3 \succsim o_2) \uplus (o_4 \succsim o_2) \mid o_3 \in [\![d_1]\!] \wedge o_4 \in [\![d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (Lemma 25)

$= \{o_3 \succsim o_2 \mid o_3 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!]\} \uplus \{o_4 \succsim o_2 \mid o_4 \in [\![d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (Def. $\uplus$)

$= [\![d_1 \text{ seq } d_3]\!] \uplus [\![d_2 \text{ seq } d_3]\!]$ (Def. seq)

$= [\![(d_1 \text{ seq } d_3) \text{ alt } (d_2 \text{ seq } d_3)]\!]$ (Lemma 22)

Notes:

1. This operation is the same as the one found in ⟨1⟩11.

⟨1⟩15. PROVE: $[\![d_1 \text{ strict } (d_2 \text{ alt } d_3)]\!] = [\![(d_1 \text{ strict } d_2) \text{ alt } (d_1 \text{ strict } d_3)]\!]$

PROOF: $[\![d_1 \text{ strict } (d_2 \text{ alt } d_3)]\!]$

$= \{o_1 \succ o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2 \text{ alt } d_3]\!]\}$ (Def. strict)

$= \{o_1 \succ o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in \{o_3 \uplus o_4 \mid o_3 \in [\![d_2]\!] \wedge o_4 \in [\![d_3]\!]\}\}$ (Def. alt)

$= \{o_1 \succ (o_3 \uplus o_4) \mid o_1 \in [\![d_1]\!] \wedge o_3 \in [\![d_2]\!] \wedge o_4 \in [\![d_3]\!]\}$ (1)

$= \{(o_1 \succ o_3) \uplus (o_1 \succ o_4) \mid o_1 \in [\![d_1]\!] \wedge o_3 \in [\![d_2]\!] \wedge o_4 \in [\![d_3]\!]\}$ (Lemma 28)

$= \{o_1 \succ o_3 \mid o_1 \in [\![d_1]\!] \wedge o_3 \in [\![d_2]\!]\} \uplus \{o_1 \succ o_4 \mid o_1 \in [\![d_1]\!] \wedge o_4 \in [\![d_3]\!]\}$ (Def. $\uplus$)

$= [\![d_1 \text{ strict } d_2]\!] \uplus [\![d_1 \text{ strict } d_3]\!]$ (Def. strict)

$= [\![(d_1 \text{ strict } d_2) \text{ alt } (d_1 \text{ strict } d_3)]\!]$ (Lemma 22)

Notes:

1. This operation is the same as the one found in ⟨1⟩11.

⟨1⟩16. PROVE: $[\![(d_1 \text{ alt } d_2) \text{ strict } d_3]\!] = [\![(d_1 \text{ strict } d_3) \text{ alt } (d_2 \text{ strict } d_3)]\!]$

PROOF: $[\![(d_1 \text{ alt } d_2) \text{ strict } d_3)]\!]$

$= \{o_1 \succ o_2 \mid o_1 \in [\![d_1 \text{ alt } d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (Def. strict)

$= \{o_1 \succ o_2 \mid o_1 \in \{o_3 \uplus o_4 \mid o_3 \in [\![d_1]\!] \wedge o_4 \in [\![d_2]\!]\} \wedge o_2 \in [\![d_3]\!]\}$ (Def. alt)

$= \{(o_3 \uplus o_4) \succ o_2 \mid o_3 \in [\![d_1]\!] \wedge o_4 \in [\![d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (1)

$= \{(o_3 \succ o_2) \uplus (o_4 \succ o_2) \mid o_3 \in [\![d_1]\!] \wedge o_4 \in [\![d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (Lemma 29)

$= \{o_3 \succ o_2 \mid o_3 \in [\![d_1]\!] \wedge o_2 \in [\![d_3]\!]\} \uplus \{o_4 \succ o_2 \mid o_4 \in [\![d_2]\!] \wedge o_2 \in [\![d_3]\!]\}$ (Def. $\uplus$)

$= [\![d_1 \text{ strict } d_3]\!] \uplus [\![d_2 \text{ strict } d_3]\!]$ (Def. strict)

$= [\![(d_1 \text{ strict } d_3) \text{ alt } (d_2 \text{ strict } d_3)]\!]$ (Lemma 22)

Notes:

1. This operation is the same as the one found in ⟨1⟩11.

⟨1⟩17. PROVE: $[\![\text{refuse } (d_1 \text{ alt } d_2)]\!] = [\![(\text{refuse } d_1) \text{ alt } (\text{refuse } d_2)]\!]$

PROOF: $[\![\text{refuse } (d_1 \text{ alt } d_2)]\!]$

$= \{\neg o \mid o \in [\![d_1 \text{ alt } d_2]\!]\}$ (Def. refuse)

$= \{\neg o \mid o \in \{o_1 \uplus o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\}\}$ (Def. alt)

$= \{\neg(o_1 \uplus o_2) \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\}$ (1)

$= \{\neg o_1 \uplus \neg o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\}$ (Lemma 30)

$= \{\neg o_1 \mid o_1 \in [\![d_1]\!]\} \uplus \{\neg o_2 \mid o_2 \in [\![d_2]\!]\}$ (Def. $\uplus$)

$= [\![\text{refuse } d_1]\!] \uplus [\![\text{refuse } d_2]\!]$ (Def. refuse)

$= [\![(\text{refuse } d_1) \text{ alt } (\text{refuse } d_2)]\!]$ (Lemma 22)

Notes:

1. This operation is the same as the one found in ⟨1⟩11.

⟨1⟩18. PROVE: $[\![\text{refuse refuse } d_1]\!] = [\![\text{refuse } d_1]\!]$

PROOF: $[\![\text{refuse refuse } d_1]\!]$

$= \{\neg o \mid o \in [\![\text{refuse } d_1]\!]\}$ (Def. refuse)

$= \{\neg o \mid o \in \{\neg o' \mid o' \in [\![d_1]\!]\}\}$ (Def. refuse)

$= \{\neg\neg o' \mid o' \in [\![d_1]\!]\}$ (1)

$= \{\neg o' \mid o' \in [\![d_1]\!]\}$ (Lemma 31)

$= \neg\{o' \mid o' \in [\![d_1]\!]\}$ (Def. $\neg$)

$= \neg[\![d_1]\!]$

$= [\![\text{refuse } d_1]\!]$ (Lemma 22)

Notes:

1. This operation is the same as the one found in ⟨1⟩11.

⟨1⟩19. PROVE: $[\![d_1 \text{ par refuse } d_2]\!] = [\![\text{refuse } (d_1 \text{ par } d_2)]\!]$

PROOF: $[\![d_1 \text{ par refuse } d_2]\!]$

$= \{o_1 \| o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![\text{refuse } d_2]\!]\}$ (Def. par)

$= \{o_1 \| o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in \{\neg o \mid o \in [\![d_2]\!]\}\}$ (Def. par)

$= \{o_1 \| \neg o \mid o_1 \in [\![d_1]\!] \wedge o \in [\![d_2]\!]\}$ (1)

$= \{\neg(o_1 \| o) \mid o_1 \in [\![d_1]\!] \wedge o \in [\![d_2]\!]\}$ (Lemma 32)

$= \neg\{o_1 \| o \mid o_1 \in [\![d_1]\!] \wedge o \in [\![d_2]\!]\}$ (Def. $\neg$)

$= \neg[\![d_1 \text{ par } d_2]\!]$ (Def. par)

$= [\![\text{refuse } (d_1 \text{ par } d_2)]\!]$ (Lemma 22)

Notes:

1. This operation is the same as the one found in ⟨1⟩11.

⟨1⟩20. PROVE: $[\![(\text{refuse } d_1) \text{ par } d_2]\!] = [\![\text{refuse } (d_1 \text{ par } d_2)]\!]$

PROOF: $[\![(\mathsf{refuse}\ d_1)\ \mathsf{par}\ d_2]\!]$
$= [\![d_2\ \mathsf{par}\ \mathsf{refuse}\ d_1]\!]$ (1)
$= [\![\mathsf{refuse}\ (d_2\ \mathsf{par}\ d_1)]\!]$ ($\langle 1 \rangle 20$)
$= [\![\mathsf{refuse}\ (d_1\ \mathsf{par}\ d_2)]\!]$ (1)

Notes:
1. Commutativety of par [70, theorem 5].

$\langle 1 \rangle 21$. PROVE: $[\![d_1\ \mathsf{seq}\ \mathsf{refuse}\ d_2]\!] = [\![\mathsf{refuse}\ (d_1\ \mathsf{seq}\ d_2)]\!]$

PROOF: $[\![d_1\ \mathsf{seq}\ \mathsf{refuse}\ d_2]\!]$
$= \{o_1 \succsim o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![\mathsf{refuse}\ d_2]\!]\}$ (Def. seq)
$= \{o_1 \succsim o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in \{\neg o \mid o \in [\![d_2]\!]\}\}$ (Def. seq)
$= \{o_1 \succsim \neg o \mid o_1 \in [\![d_1]\!] \wedge o \in [\![d_2]\!]\}$ (1)
$= \{\neg(o_1 \succsim o) \mid o_1 \in [\![d_1]\!] \wedge o \in [\![d_2]\!]\}$ (Lemma 33)
$= \neg\{o_1 \succsim o \mid o_1 \in [\![d_1]\!] \wedge o \in [\![d_2]\!]\}$ (Def. $\neg$)
$= \neg[\![d_1\ \mathsf{seq}\ d_2]\!]$ (Def. seq)
$= [\![\mathsf{refuse}\ (d_1\ \mathsf{seq}\ d_2)]\!]$ (Lemma 22)

Notes:
1. This operation is the same as the one found in $\langle 1 \rangle 11$.

$\langle 1 \rangle 22$. PROVE: $[\![(\mathsf{refuse}\ d_1)\ \mathsf{seq}\ d_2]\!] = [\![\mathsf{refuse}\ (d_1\ \mathsf{seq}\ d_2)]\!]$

PROOF: $[\![(\mathsf{refuse}\ d_1)\ \mathsf{seq}\ d_2]\!]$
$= \{o_1 \succsim o_2 \mid o_1 \in [\![\mathsf{refuse}\ d_1]\!] \wedge o_2 \in [\![d_2]\!]\}$ (Def. seq)
$= \{o_1 \succsim o_2 \mid o_1 \in \{\neg o \mid o \in [\![d_2]\!]\} \wedge o_2 \in [\![d_2]\!]\}$ (Def. seq)
$= \{(\neg o) \succsim o_2 \mid o \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\}$ (1)
$= \{\neg(o \succsim o_2) \mid o \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\}$ (Lemma 34)
$= \neg\{o \succsim o_2 \mid o \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\}$ (Def. $\neg$)
$= \neg[\![d_1\ \mathsf{seq}\ d_2]\!]$ (Def. seq)
$= [\![\mathsf{refuse}\ (d_1\ \mathsf{seq}\ d_2)]\!]$ (Lemma 22)

Notes:
1. This operation is the same as the one found in $\langle 1 \rangle 11$.

$\langle 1 \rangle 23$. PROVE: $[\![d_1\ \mathsf{strict}\ \mathsf{refuse}\ d_2]\!] = [\![\mathsf{refuse}\ (d_1\ \mathsf{strict}\ d_2)]\!]$

PROOF: $[\![d_1\ \mathsf{strict}\ \mathsf{refuse}\ d_2]\!]$
$= \{o_1 \succ o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in [\![\mathsf{refuse}\ d_2]\!]\}$ (Def. strict)
$= \{o_1 \succ o_2 \mid o_1 \in [\![d_1]\!] \wedge o_2 \in \{\neg o \mid o \in [\![d_2]\!]\}\}$ (Def. strict)
$= \{o_1 \succ \neg o \mid o_1 \in [\![d_1]\!] \wedge o \in [\![d_2]\!]\}$ (1)
$= \{\neg(o_1 \succ o) \mid o_1 \in [\![d_1]\!] \wedge o \in [\![d_2]\!]\}$ (Lemma 35)
$= \neg\{o_1 \succ o \mid o_1 \in [\![d_1]\!] \wedge o \in [\![d_2]\!]\}$ (Def. $\neg$)
$= \neg[\![d_1\ \mathsf{strict}\ d_2]\!]$ (Def. strict)
$= [\![\mathsf{refuse}\ (d_1\ \mathsf{strict}\ d_2)]\!]$ (Lemma 22)

Notes:
1. This operation is the same as the one found in $\langle 1 \rangle 11$.

$\langle 1 \rangle 24$. PROVE: $[\![(\mathsf{refuse}\ d_1)\ \mathsf{strict}\ d_2]\!] = [\![\mathsf{refuse}\ (d_1\ \mathsf{strict}\ d_2)]\!]$

PROOF: $[\![(\mathsf{refuse}\ d_1)\ \mathsf{strict}\ d_2]\!]$

$$
\begin{array}{lll}
= & \{o_1 \succ o_2 \mid o_1 \in [\![\mathsf{refuse}\ d_1]\!] \wedge o_2 \in [\![d_2]\!]\} & \text{(Def. strict)} \\
= & \{o_1 \succ o_2 \mid o_1 \in \{\neg o \mid o \in [\![d_2]\!]\} \wedge o_2 \in [\![d_2]\!]\} & \text{(Def. strict)} \\
= & \{(\neg o) \succ o_2 \mid o \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\} & (1) \\
= & \{\neg(o \succ o_2) \mid o \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\} & \text{(Lemma 36)} \\
= & \neg\{o \succ o_2 \mid o \in [\![d_1]\!] \wedge o_2 \in [\![d_2]\!]\} & \text{(Def. } \neg) \\
= & \neg[\![d_1\ \mathsf{strict}\ d_2]\!] & \text{(Def. strict)} \\
= & [\![\mathsf{refuse}\ (d_1\ \mathsf{strict}\ d_2)]\!] & \text{(Lemma 22)}
\end{array}
$$

Notes:

1. This operation is the same as the one found in ⟨1⟩11.

⟨1⟩25. Q.E.D.

PROOF: ⟨1⟩1-⟨1⟩24.

□

Appendix C

Maude implementation of the operational semantics

This appendix provides all the details of the implementation of the operational semantics presented in chapter 10. Section C.1 contains the implementation of the syntax presented in section 5.2, section C.2 contains the implementation of the operational semantics presented in chapter 8, and section C.3 contains the implementation of the meta-strategies presented in chapter 9.

C.1 Syntax

```
fmod LIFELINE is
  sorts LLName Lifeline Gate .
  subsort Gate < Lifeline .

  op l : LLName -> Lifeline [ctor] .
  op g : LLName -> Gate [ctor] .
endfm

fmod LLSET is
  including LIFELINE .
  including NAT .
  sort LLSet .
  subsort Lifeline < LLSet .

  op emptyLLSet : -> LLSet [ctor] .
  op _;_ : LLSet LLSet -> LLSet [ctor assoc comm id: emptyLLSet] .

  op _in_ : Lifeline LLSet -> Bool .
  op intersect : LLSet LLSet -> LLSet .
  op _\_ : LLSet Lifeline -> LLSet .
  op _\_ : LLSet LLSet -> LLSet .

  vars L L' : Lifeline .
  vars LS LS' : LLSet .

  eq L ; L = L .
```

```
  eq L in emptyLLSet = false .
  eq L in (L' ; LS) =
     if L == L' then
       true
     else
       L in LS
     fi .

  eq intersect(emptyLLSet, LS) = emptyLLSet .
  eq intersect(L ; LS, LS') =
     (if L in LS' then L else emptyLLSet fi) ;
     intersect(LS, LS') .

  eq LS \ emptyLLSet = LS .
  ceq LS \ (L ; LS') = (LS \ L) \ LS' if LS' =/= emptyLLSet .
  eq (L ; LS) \ L' =
    if L == L' then
      LS
    else
      L ; (LS \ L')
   fi .

  eq emptyLLSet \ L = emptyLLSet .
endfm

fmod MESSAGE is
  including LIFELINE .

  sorts Signal Msg .

  op (_,_,_) : Signal Lifeline Lifeline -> Msg [ctor] .

  ops tr re : Msg -> Lifeline .

  var S : Signal .
  vars T R L : Lifeline .

  eq tr(S, T, R) = T .
  eq re(S, T, R) = R .
endfm

fmod EVENT is
  including MESSAGE .

  sorts Event EventKind SEvent SEventKind .
  subsort SEvent < Event .

  ops ! ? : -> EventKind [ctor] .
  op (_,_) : EventKind Msg -> Event [ctor] .
  op tau : SEventKind -> SEvent [ctor] .
```

```
  op k : Event -> EventKind .
  op m : Event -> Msg .
  ops tr re l l-1 : Event -> Lifeline .

  var M : Msg .
  var K : EventKind .

  eq k(K,M) = K .
  eq m(K,M) = M .
  eq tr(K,M) = tr(M) .
  eq re(K,M) = re(M) .
  ceq l(K,M) = tr(M) if K = ! .
  ceq l(K,M) = re(M) if K = ? .
  ceq l-1(K,M) = re(M) if K = ! .
  ceq l-1(K,M) = tr(M) if K = ? .
endfm

fmod SD-SYNTAX is
  including EVENT .
  including LLSET .

  sort SD .
  subsort Event < SD .

  op skip : -> SD [ctor] .
  op _seq_ : SD SD -> SD [ctor assoc prec 20] .
  op _strict_ : SD SD -> SD [ctor assoc prec 25] .
  op _par_ : SD SD -> SD [ctor assoc comm prec 30] .

  op ll : SD -> LLSet .
  op gates : SD -> LLSet .

  vars D D' : SD .
  var E : Event .

  eq D par skip = D .
  eq skip seq D = D .
  eq D seq skip = D .
  eq D strict skip = D .
  eq skip strict D = D .

  eq ll(skip) = emptyLLSet .
  eq ll(E) = l(E) .
  eq ll(D seq D') = ll(D) ; ll(D') .
  eq ll(D par D') = ll(D) ; ll(D') .
  eq ll(D strict D') = ll(D) ; ll(D') .

  eq gates(skip) = emptyLLSet .
  eq gates(E) = if l-1(E) :: Gate then l-1(E) else emptyLLSet fi .
  eq gates(D seq D') = gates(D) ; gates(D') .
```

```
  eq gates(D par D') = gates(D) ; gates(D') .
  eq gates(D strict D') = gates(D) ; gates(D') .
endfm

fmod EXTENSION-ALT is
  including SD-SYNTAX .

  op _alt_ : SD SD -> SD [ctor assoc comm prec 35] .
  op opt_ : SD -> SD [ctor prec 15] .
  op alt : -> SEventKind [ctor] .

  vars D D' : SD .

  eq skip alt skip = skip .
  eq opt D = D alt skip .

  eq ll(D alt D') = ll(D) ; ll(D') .
  eq gates(D alt D') = gates(D) ; gates(D') .
endfm

fmod EXTENSION-XALT is
  including SD-SYNTAX .

  op _xalt_ : SD SD -> SD [ctor assoc comm prec 40] .
  op xalt : -> SEventKind [ctor] .

  vars D D' : SD .

  eq skip xalt skip = skip .

  eq ll(D xalt D') = ll(D) ; ll(D') .
  eq gates(D xalt D') = gates(D) ; gates(D') .
endfm

fmod EXTENSION-REFUSE is
  including SD-SYNTAX .
  including EXTENSION-ALT .

  op refuse_ : SD -> SD [ctor prec 1] .
  op refuse : -> SEventKind [ctor] .
  op neg_ : SD -> SD [ctor prec 10] .

  var D : SD .

  eq neg D = refuse D alt skip .

  eq ll(refuse D) = ll(D) .
  eq gates(refuse D) = gates(D) .
endfm
```

```
fmod EXTENSION-ASSERT is
  including SD-SYNTAX .

  op assert_ : SD -> SD [ctor prec 5] .
  op assert : -> SEventKind [ctor] .

  var D : SD .

  eq ll(assert D)  = ll(D) .
  eq gates(assert D) = gates(D) .
endfm

fmod NAT-SET is
  including NAT .

  sort NatSet .
  subsort Nat < NatSet .

  op emptyNatSet : -> NatSet [ctor] .
  op _,_ : NatSet NatSet -> NatSet [ctor assoc comm id: emptyNatSet] .

  op toSet : Nat Nat -> NatSet .

  vars N N' : Nat .
  var NS : NatSet .

  eq N , N = N .

  ceq toSet(N,N') = N , toSet(s N, N') if N <= N' .
  ceq toSet(N,N') = emptyNatSet if N > N' .
endfm

fmod EXTENSION-LOOP is
  including SD-SYNTAX .
  including NAT-SET .
  including EXTENSION-ALT .

  op loop{_}_ : NatSet SD -> SD [ctor prec 7] .
  op loop<_>_ : Nat SD -> SD [ctor prec 7] .
  op loop(_,_)_ : Nat Nat SD -> SD [ctor prec 7] .
  op loop : -> SEventKind [ctor] .

  var D : SD .
  vars N N' : Nat .
  var I : NatSet .

  eq loop< 0 > D = skip .
  eq loop(N,N') D = loop{toSet(N,N')} D .
  eq loop{emptyNatSet} D = skip .
  eq loop{N} D = loop< N > D .
  ceq loop{N , I} D = loop< N > D alt loop{I} D if I =/= emptyNatSet .
```

```
  eq ll(loop< N > D) = ll(D) .
  eq gates(loop< N > D) = gates(D) .
endfm

fmod SD-FULL-SYNTAX is
  including SD-SYNTAX .
  including EXTENSION-ALT .
  including EXTENSION-XALT .
  including EXTENSION-REFUSE .
  including EXTENSION-ASSERT .
  including EXTENSION-LOOP .
endfm
```

C.2 Operational semantics

C.2.1 Communication medium

```
fmod COMMUNICATION is
  including MESSAGE .
  including SD-FULL-SYNTAX .

  sort Comm .

  op emptyComm : -> Comm [ctor] .
  ops add rm : Comm Msg -> Comm .
  op ready : Comm Msg -> Bool .
  op initC : SD -> Comm .
endfm

fmod ONE-FIFO-MESSAGE is
  including COMMUNICATION .

  subsort Msg < Comm .

  op _::_ : Comm Comm -> Comm [ctor assoc comm id: emptyComm] .

  vars M M' : Msg .
  var B : Comm .
  vars D1 D2 : SD .
  var N : Nat .
  var E : Event .

  eq add(B,M) = B :: M .
  eq rm(emptyComm, M) = emptyComm .
  eq rm(M :: B, M) = B .
  ceq rm(M :: B, M') = M :: rm(B,M') if M =/= M' .
  eq ready(emptyComm, M) = false .
  eq ready(M :: B, M') = (M == M') or ready(B,M') .
```

```
  eq initC(skip) = emptyComm .
  ceq initC(E) = emptyComm if k(E) == ! or not tr(E) :: Gate .
  ceq initC(E) = m(E) if k(E) == ? and tr(E) :: Gate .
  eq initC(D1 seq D2) = initC(D1) :: initC(D2) .
  eq initC(D1 par D2) = initC(D1) :: initC(D2) .
  eq initC(D1 strict D2) = initC(D1) :: initC(D2) .
  eq initC(D1 alt D2) = initC(D1) :: initC(D2) .
  eq initC(D1 xalt D2) = initC(D1) :: initC(D2) .
  eq initC(refuse D1) = initC(D1) .
  eq initC(assert D1) = initC(D1) .
  eq initC(loop< s N > D1) = initC(D1) :: initC(loop< N > D1)  .
endfm

fmod ONE-FIFO-PAIR-LIFELINES is
  including COMMUNICATION .

  sort Channel .
  sort Fifo .
  subsort Msg < Fifo .
  subsort Channel < Comm .

  op emptyFifo : -> Fifo [ctor] .
  op _::_ : Fifo Fifo -> Fifo [ctor assoc id: emptyFifo] .

  op _,_:_ : Lifeline Lifeline Fifo -> Channel [ctor] .
  op _;_ : Comm Comm -> Comm [ctor assoc comm id: emptyComm] .

  op _<<_ : Comm Channel -> Comm .
  op _<<_ : Comm Comm -> Comm .

  vars T T' R R' : Lifeline .
  vars B B' : Fifo .
  vars C C' : Comm .
  vars M M' : Msg .
  vars D1 D2 : SD .
  var N : Nat .
  var E : Event .

  eq emptyComm << (T,R : B) = T,R : B .
  eq ((T,R : B) ; C) << (T,R : B') = (T,R : B :: B') ; C .
  ceq ((T,R : B) ; C) << (T',R' : B') = (T,R : B) ; (C << (T',R' : B'))
      if T =/= T' or R =/= R' .

  eq C << emptyComm = C .
  ceq C << ((T,R : B) ; C') = (C << (T,R : B)) << C'
     if C' =/= emptyComm .

  eq add(C,M) = C << (tr(M),re(M) : M) .
  eq rm((T,R : M :: B) ; C , M') =
     if tr(M) == T and re(M) == R and M == M' then
        (T,R : B) ; C
```

```
     else
        (T,R : M :: B) ; rm(C,M')
     fi .
  eq ready (emptyComm,M) = false .
  eq ready((T,R : M :: B) ; C, M') = (tr(M) == T and re(M) == R and M == M')
                                     or ready(C,M') .

  eq initC(skip) = emptyComm .
  ceq initC(E) = emptyComm if k(E) == ! or not tr(E) :: Gate .
  ceq initC(E) = tr(E),re(E) : m(E) if k(E) == ? and tr(E) :: Gate .
  eq initC(D1 seq D2) = initC(D1) << initC(D2) .
  eq initC(D1 par D2) = initC(D1) << initC(D2) .
  eq initC(D1 strict D2) = initC(D1) << initC(D2) .
  eq initC(D1 alt D2) = initC(D1) << initC(D2) .
  eq initC(D1 xalt D2) = initC(D1) << initC(D2) .
  eq initC(refuse D1) = initC(D1) .
  eq initC(assert D1) = initC(D1) .
  eq initC(loop< s N > D1) = initC(D1) << initC(loop< N > D1)  .
endfm

fmod ONE-FIFO-LIFELINE is
  including COMMUNICATION .

  sort Channel .
  sort Fifo .
  subsort Msg < Fifo .
  subsort Channel < Comm .

  op emptyFifo : -> Fifo [ctor] .
  op _::_ : Fifo Fifo -> Fifo [ctor assoc id: emptyFifo] .

  op _:_ : Lifeline Fifo -> Channel [ctor] .
  op _;_ : Comm Comm -> Comm [ctor assoc comm id: emptyComm] .

  op _<<_ : Comm Channel -> Comm .
  op _<<_ : Comm Comm -> Comm .

  vars L L' : Lifeline .
  vars F F' : Fifo .
  var B : Comm .
  vars M M' : Msg .
  vars D1 D2 : SD .
  var N : Nat .
  var E : Event .

  eq L : emptyFifo = emptyComm .

  eq emptyComm << (L : F) = L : F .
  eq ((L : F) ; B) << (L : F') = (L : F :: F') ; B .
  ceq ((L : F) ; B) << (L' : F') = (L : F) ; (B << (L' : F'))
     if L =/= L' .
```

```
  eq add(B,M) = B << (re(M) : M) .
  eq rm(emptyComm, M) = emptyComm .
  eq rm((L : (M :: F)) ; B, M') =
     if re(M) == L and M == M' then
        (L : F) ; B
     else
         (L : (M :: F)) ; rm(B, M')
     fi .
  eq ready(emptyComm, M) = false .
  eq ready((L : (M :: F)) ; B, M') = (re(M) == L and M == M')
                                     or ready(B,M') .
endfm

fmod GLOBAL-FIFO is
  including COMMUNICATION .

  subsort Msg < Comm .

  op _::_ : Comm Comm -> Comm [ctor assoc id: emptyComm] .

  vars M M' : Msg .
  var B : Comm .

  eq add(B,M) = B :: M .
  eq rm(emptyComm, M) = emptyComm .
  eq rm(M :: B, M) = B .
  ceq rm(M :: B, M') = M :: B if M =/= M' .
  eq ready(emptyComm, M) = false .
  eq ready(M :: B, M') = (M == M') .
endfm
```

C.2.2 Projection system

```
fmod PI is
  including SD-SYNTAX .
  including COMMUNICATION .

  sort Pi .

  op PI : LLSet Comm SD -> Pi [ctor] .
  op {_}_ : Event Pi -> Pi [ctor] .
endfm

mod PI-SIMPLE is
  including PI .

  vars D1 D1' D2 D2' : SD .
  var E : Event .
  vars L L' : LLSet .
  var B : Comm .
```

```
  crl [PI-event] :
    PI(L, B, E) => {E}PI(L, B, skip)
    if l(E) in L /\
       k(E) == ! or ready(B,m(E)) .

  crl [PI-seq-left] :
    PI(L, B, D1 seq D2) => {E}PI(L, B,  D1' seq D2)
    if intersect(ll(D1),L) =/= emptyLLSet /\
       PI(intersect(ll(D1),L), B, D1) => {E}PI(L', B, D1') .

  crl [PI-seq-right] :
    PI(L, B, D1 seq D2) => {E}PI(L, B, D1 seq D2')
    if L \ ll(D1) =/= emptyLLSet /\
       PI(L \ ll(D1), B, D2) => {E}PI(L', B, D2') .

  crl [PI-par] :
     PI(L, B, D1 par D2) => {E}PI(L, B, D1' par D2)
     if PI(intersect(ll(D1),L), B, D1) => {E}PI(L', B, D1') .

  crl [PI-strict] :
     PI(L, B, D1 strict D2) => {E}PI(L, B, D1' strict D2)
     if intersect(ll(D1), L) =/= emptyLLSet /\
        PI(intersect(ll(D1), L), B, D1) => {E}PI(L', B, D1') .
endm

mod PI-ALT is
  including PI .
  including EXTENSION-ALT .

  vars D1 D2 : SD .
  var L : LLSet .
  var B : Comm .

  crl [PI-alt] :
    PI(L, B, D1 alt D2) => {tau(alt)}PI(L, B, D1)
    if intersect(ll(D1 alt D2), L) =/= emptyLLSet .
endm

mod PI-XALT is
  including PI .
  including EXTENSION-XALT .

  vars D1 D2 : SD .
  var L : LLSet .
  var B : Comm .

  crl [PI-xalt] :
    PI(L, B, D1 xalt D2) => {tau(xalt)}PI(L, B, D1)
    if intersect(ll(D1 alt D2), L) =/= emptyLLSet .
endm
```

```
mod PI-REFUSE is
  including PI .
  including EXTENSION-REFUSE .

  var D : SD .
  var L : LLSet .
  var B : Comm .

  crl [PI-refuse] :
    PI(L, B, refuse D) => {tau(refuse)}PI(L, B, D)
    if intersect(ll(refuse D), L) =/= emptyLLSet .
endm

mod PI-ASSERT is
  including PI .
  including EXTENSION-ASSERT .

  var D : SD .
  var L : LLSet .
  var B : Comm .

  crl [PI-assert] :
    PI(L, B, assert D) => {tau(assert)}PI(L, B, D)
    if intersect(ll(assert D), L) =/= emptyLLSet .
endm

mod PI-LOOP is
  including PI .
  including EXTENSION-LOOP .

  var D : SD .
  var L : LLSet .
  var B : Comm .
  var N : Nat .

  crl [PI-loop-dec] :
    PI(L, B, loop< s N > D) => {tau(loop)}PI(L, B, D seq loop< N > D)
    if intersect(ll(loop< s N > D), L) =/= emptyLLSet .
endm

mod PI-FULL is
  including PI-SIMPLE .
  including PI-ALT .
  including PI-XALT .
  including PI-REFUSE .
  including PI-ASSERT .
  including PI-LOOP .
endm
```

C.2.3 Execution system

```
fmod EXEC is
  including PI .

  sort Exec .

  op {_}_ : Event Exec -> Exec [ctor] .
  op init : SD -> Exec .
  op update : Comm Event -> Comm .

  var B : Comm .
  var E : Event .

  ceq update(B,E) =
    (if k(E) == ! then
      add(B,m(E))
     else
       rm(B,m(E))
     fi)
    if not E :: SEvent .

   ceq update(B,E) = B if E :: SEvent .
endfm

mod EXECUTION is
  including EXEC .

  op [_,_] : Comm SD -> Exec [ctor] .

  var B : Comm .
  vars D D' : SD .
  var E : Event .
  var L : LLSet .

  eq init(D) = [initC(D), D] .

  crl [execute] :
    [B, D] => {E}[update(B,E), D']
    if PI(ll(D), B, D) => {E}PI(L, B, D') .
endm
```

C.3 Meta-strategies

```
fmod TRACE is
  including EVENT .

  sort SDTrace .
  subsort Event < SDTrace .

  op emptyTrace : -> SDTrace [ctor] .
```

```
  op _._ : SDTrace SDTrace -> SDTrace [ctor assoc id: emptyTrace] .
endfm

mod TRACES-COMMON is
  including META-LEVEL .
  including EXEC .

  op mod : -> Module .
  op getExec : Substitution -> Exec .
  op getEvent : Substitution -> Event .
  op getComm : Substitution -> Comm .
  op getDiagram : Substitution -> SD .
  op external : Event LLSet -> Bool .

  op error : -> [Exec] .
  op error : -> [Event] .
  op error : -> [SD] .
  op error : -> [Comm] .

  var S : Substitution .
  var TERM : Term .
  var V : Variable .
  var EV : Event .
  var L : LLSet .

  ceq getExec(S ; V <- TERM) = downTerm(TERM, (error).Exec)
      if V == 'E:Exec .
  ceq getExec(S ; V <- TERM) = getExec(S) if V =/= 'E:Exec .

  ceq getComm(S ; V <- TERM) = downTerm(TERM, (error).Comm)
      if V == 'B:Comm .
  ceq getComm(S ; V <- TERM) = getComm(S) if V =/= 'B:Comm .

  ceq getDiagram(S ; V <- TERM) = downTerm(TERM, (error).SD)
      if V == 'D':SD .
  ceq getDiagram(S ; V <- TERM) = getDiagram(S) if V =/= 'D':SD .

  ceq getEvent(S ; V <- TERM) = downTerm(TERM, (error).Event)
      if V == 'F:Event .
  ceq getEvent(S ; V <- TERM) = getEvent(S) if V =/= 'F:Event .

  eq external(EV, L) = (l(EV) in L) or (l-1(EV) in L) .
endm
```

C.3.1 One trace with mode

```
mod META-EXEC-MODE is
  including EXEC .
  including TRACE .

  sort Meta .
```

```
  sort Mode .

  op <_,_,_,_> : SDTrace Exec LLSet Mode -> Meta [ctor] .

  op mode : SEvent Mode -> Mode .
endm

mod DEFAULT-MODE is
  including META-EXEC-MODE .

  ops positive negative : -> Mode [ctor] .

  var T : SEvent .
  var MO : Mode .

  ceq mode(T,MO) = positive if MO == positive /\ T =/= tau(refuse) .
  ceq mode(T,MO) = negative if MO == negative or T == tau(refuse) .
endm

fmod MYRANDOM is
  protecting RANDOM .

  op ran : Nat Nat Nat -> Nat .

  vars N M K R MAX : Nat .

  ceq ran(N, M, K) =  N + ((sd(N,M) + 1) * R) quo MAX
      if R := random(K) /\
         MAX := 4294967295 .
endfm

mod EXTERNAL-TRACE-MODE is
  including META-EXEC-MODE .
  including SD-FULL-SYNTAX .

  var T : SDTrace .
  var E : Event .
  vars V V' : Exec .
  var L : LLSet .
  vars M M' : Mode .
  var D : SD .
  var B : Comm .
  var TAU : SEvent .

  crl < T, V, L, M > => < T . E , V', L, M >
     if V => {E}V' /\
        not E :: SEvent /\
        (l(E) in L or l-1(E) in L) .

  crl < T, V, L, M > => < T, V', L, M >
     if V => {E}V' /\
```

```
        not E :: SEvent /\
        not l(E) in L /\
        not l-1(E) in L .

  crl < T , V, L, M > => < T, V', L, mode(TAU,M)  >
      if V => {TAU}V' .
endm

mod RESULT-LIST is
  including META-LEVEL .

  sort ResultList TraceAndMode .
  subsort ResultTriple < ResultList .

  op emptyRL : -> ResultList [ctor] .
  op _::_ : ResultList ResultList -> ResultList [ctor assoc id: emptyRL] .

  op length : ResultList -> Nat .
  op _[_] : ResultList Nat -> ResultTriple .

  var R3 : ResultTriple .
  var RL : ResultList .
  var N : Nat .

  eq length(emptyRL) = 0 .
  eq length(R3 :: RL) = s length(RL) .

  eq (R3 :: RL)[1] = R3 .
  ceq (R3 :: RL)[s N] = RL[N] if N > 0 .
endm

mod RANDOM-TRACE is
  including META-EXEC-MODE .
  including DEFAULT-MODE .
  including RESULT-LIST .
  including MYRANDOM .
  including TRACES-COMMON .
  including PI-FULL .

  op {_|_} : SDTrace Mode -> TraceAndMode [ctor] .

  op randomTrace : Exec LLSet Nat -> TraceAndMode .
  op exec : Meta Nat -> TraceAndMode .
  op getExecs : Exec Nat -> ResultList .

  var RL : ResultList .
  var R3? : ResultTriple? .
  vars N K R : Nat .
  var X : Exec .
  var L : LLSet .
  var T : SDTrace .
```

```
  var M : Mode .
  var EV : Event .
  var TAU : SEvent .

  eq randomTrace(X, L, N) = exec( < emptyTrace, X, L, positive > , N ) .

  ceq exec( < T , X , L , M > , N ) =
     if K == 0 then
         {T | M}
     else
        exec( < (if not (EV :: SEvent) and external(EV, L) then
                   T . EV
                 else
                   T
                 fi),
                 getExec(getSubstitution(RL[R])),
                 L,
                 (if EV :: SEvent then
                    mode(EV, M)
                 else
                    M
                 fi) > , s N)
      fi
      if RL := getExecs(X, 0) /\
         K := length(RL) /\
         R := if K == 0 then 0 else ran(1,K,N) fi /\
         EV := if K == 0 then tau(loop)
               else getEvent(getSubstitution(RL[R])) fi .

  ceq getExecs(X, N) =
      if R3? :: ResultTriple then
        R3? :: getExecs(X, s N)
      else
        emptyRL
      fi
      if R3? := metaSearch(mod, upTerm(X), ''{_'}_['F:Event,'E:Exec],
                           nil, '+, 1, N) .
endm
```

C.3.2 All traces

```
mod META-EXEC is
  including EXEC .
  including TRACE .

  sort Meta .

  op {|_,_,_|} : SDTrace Exec LLSet -> Meta [ctor] .
endm

mod SEMANTICS is
```

```
  including TRACE .

  sort TraceSet .
  sort InteractionObligation .
  sort IOSet .
  subsort SDTrace < TraceSet .
  subsort InteractionObligation < IOSet .

  op emptyTSet : -> TraceSet [ctor] .
  op _+_ : TraceSet TraceSet -> TraceSet [ctor assoc comm id: emptyTSet] .
  op _in_ : SDTrace TraceSet -> Bool .
  op subseteq : TraceSet TraceSet -> Bool .

  op (_,_) : TraceSet TraceSet -> InteractionObligation [ctor] .

  op emptyIOSet : -> IOSet [ctor] .
  op _&_ : IOSet IOSet -> IOSet [ctor assoc comm id: emptyIOSet] .

  vars T T' : SDTrace .
  vars TS P N TS' : TraceSet .
  var O : InteractionObligation .

  eq T in emptyTSet = false .
  eq T in (T' + TS) =
     if T == T' then
       true
     else
       T in TS
     fi .

  eq T + T = T .
  eq O & O = O .

  eq subseteq(emptyTSet, TS') = true .
  eq subseteq(T + TS, TS') = T in TS' and subseteq(TS, TS') .
endm

fmod TRANSFORMATIONS is
  including SD-FULL-SYNTAX .

  vars D1 D2 D3 : SD .
  var N : Nat .

  eq loop< s N > D1 = D1 seq loop< N > D1 .
  eq D1 alt (D2 xalt D3) = (D1 alt D2) xalt (D1 alt D3) .
  eq D1 par (D2 xalt D3) = (D1 par D2) xalt (D1 par D3) .
  eq D1 seq (D2 xalt D3) = (D1 seq D2) xalt (D1 seq D3) .
  eq (D1 xalt D2) seq D3 = (D1 seq D3) xalt (D2 seq D3) .
  eq D1 strict (D2 xalt D3) = (D1 strict D2) xalt (D1 strict D3) .
  eq (D1 xalt D2) strict D3 = (D1 strict D3) xalt (D2 strict D3) .
  eq refuse (D1 xalt D2) = (refuse D1) xalt (refuse D2) .
```

```
  eq D1 par (D2 alt D3) = (D1 par D2) alt (D1 par D3) .
  eq D1 seq (D2 alt D3) = (D1 seq D2) alt (D1 seq D3) .
  eq (D1 alt D2) seq D3 = (D1 seq D3) alt (D2 seq D3) .
  eq D1 strict (D2 alt D3) = (D1 strict D2) alt (D1 strict D3) .
  eq (D1 alt D2) strict D3 = (D1 strict D3) alt (D2 strict D3) .
  eq refuse (D1 alt D2) = (refuse D1) alt (refuse D2) .
  eq refuse refuse D1 = refuse D1 .
  eq D1 par refuse D2 = refuse (D1 par D2) .
  eq D1 seq refuse D2 = refuse (D1 seq D2) .
  eq (refuse D1) seq D2 = refuse (D1 seq D2) .
  eq D1 strict refuse D2 = refuse (D1 strict D2) .
  eq (refuse D1) strict D2 = refuse (D1 strict D2) .
endfm

mod ALL-TRACES is
  including META-EXEC .
  including TRANSFORMATIONS .
  including SEMANTICS .
  including TRACES-COMMON .
  including EXECUTION .

  subsort Meta < SDTrace .

  op allTraces : Exec LLSet -> IOSet .
  op execXalt : IOSet -> IOSet .
  op getXalts : InteractionObligation -> IOSet .
  op execAlt : IOSet -> IOSet .
  op execAlt : TraceSet -> TraceSet .
  op getAlts : SDTrace -> TraceSet .
  op execRefuse : IOSet -> IOSet .
  ops getPos getNeg : TraceSet -> TraceSet .
  op execEvents : IOSet -> IOSet .
  op execEvents : TraceSet -> TraceSet .
  op getEvents : SDTrace Nat -> TraceSet .
  ops xaltTop altTop refuseTop : SD -> Bool .

  var X : Exec .
  var L : LLSet .
  var IOS  : IOSet .
  var R3? : ResultTriple? .
  var T : SDTrace .
  var N : Nat .
  vars TS TS' POS NEG : TraceSet .
  vars D1 D2 D : SD .
  var B : Comm .
  var E : Event .

  eq xaltTop(D1 xalt D2) = true .
  eq xaltTop(D1 alt D2) = false .
  eq xaltTop(refuse D) = false .
  eq xaltTop(D1 seq D2) = false .
```

```
eq xaltTop(D1 par D2) = false .
eq xaltTop(D1 strict D2) = false .
eq xaltTop(E) = false .
eq xaltTop(skip) = false .

eq altTop(D1 xalt D2) = false .
eq altTop(D1 alt D2) = true .
eq altTop(refuse D) = false .
eq altTop(D1 seq D2) = false .
eq altTop(D1 par D2) = false .
eq altTop(D1 strict D2) = false .
eq altTop(E) = false .
eq altTop(skip) = false .

eq refuseTop(D1 xalt D2) = false .
eq refuseTop(D1 alt D2) = false .
eq refuseTop(refuse D) = true .
eq refuseTop(D1 seq D2) = false .
eq refuseTop(D1 par D2) = false .
eq refuseTop(D1 strict D2) = false .
eq refuseTop(E) = false .
eq refuseTop(skip) = false .

eq allTraces(X, L) = execEvents(execRefuse(execAlt(
                     execXalt(({| emptyTrace, X, L |}, emptyTSet))))) .

ceq getXalts( ( {| T, [B,D] , L |}, emptyTSet ) ) =
    ( {| T , [B, D] , L |}, emptyTSet )
    if not xaltTop(D) .

eq getXalts( ( {| T, [B,D1 xalt D2], L |}, emptyTSet ) ) =
   getXalts( ( {| T, [B, D1], L |} , emptyTSet ) ) &
   getXalts( ( {| T, [B, D2], L |} , emptyTSet ) ) .

eq execXalt(emptyIOSet) = emptyIOSet .
eq execXalt( IOS & ( {| T, X, L |}, emptyTSet ) ) =
    execXalt(IOS) & getXalts( ( {| T, X, L |}, emptyTSet ) ) .

ceq getAlts( {| T, [B, D], L |} ) = {| T, [B, D], L |}
    if not altTop(D) .

eq getAlts( {| T, [B, D1 alt D2], L |} ) =
   getAlts( {| T, [B,D1], L |} ) + getAlts( {| T, [B,D2], L |} ) .

eq execAlt(emptyIOSet) = emptyIOSet .
eq execAlt( IOS & ( TS , emptyTSet ) ) =
   execAlt(IOS) & ( execAlt(TS), emptyTSet ) .

eq execAlt(emptyTSet) = emptyTSet .
eq execAlt( TS + {| T, X, L |} ) =
   execAlt(TS) + getAlts( {| T, X, L |} ) .
```

```
  eq getPos(emptyTSet) = emptyTSet .
  eq getPos( TS + {| T, [B, refuse D], L |} ) = getPos(TS) .
  ceq getPos( TS + {| T, [B,D], L |} ) = getPos(TS) + {| T, [B,D], L |}
      if not refuseTop(D) .

  eq getNeg(emptyTSet) = emptyTSet .
  eq getNeg( TS + {| T, [B, refuse D], L |} ) =
     getNeg(TS) + {| T, [B, D], L |} .
  ceq getNeg( TS + {| T, [B,D], L |} ) = getNeg(TS)
      if not refuseTop(D) .

  eq execRefuse(emptyIOSet) = emptyIOSet .
   ceq execRefuse(IOS & (TS , emptyTSet)) = execRefuse(IOS) &  ( POS, NEG )
       if POS := getPos(TS) /\
          NEG := getNeg(TS) .

  eq execEvents(emptyIOSet) = emptyIOSet .
  eq execEvents(IOS & (POS, NEG) ) =
     execEvents(IOS) & (execEvents(POS), execEvents(NEG)) .

  eq execEvents(emptyTSet) = emptyTSet .
  ceq execEvents( TS + {| T, X, L |} ) =
      if TS' =/= emptyTSet then
        execEvents(TS + TS')
      else
        execEvents(TS) + T
      fi
      if TS' := getEvents( {| T, X, L |}, 0 ) .

  ceq getEvents( {| T, X, L |}, N) =
      if R3? :: ResultTriple then
        {| (if external(getEvent(getSubstitution(R3?)), L) then
               T . getEvent(getSubstitution(R3?))
            else
               T
            fi) ,
            getExec(getSubstitution(R3?)), L |} +
            getEvents( {| T, X, L |} , s N )
      else
        emptyTSet
      fi
      if R3? := metaSearch(mod, upTerm(X), ''{_'}_['F:Event,'E:Exec],
                           nil, '+, 1, N) .
endm
```

C.3.2.1 Refinement verification

```
mod MAPPING is
  including SEMANTICS .
  including LLSET .
```

```
sort Mapping .

op _->_ : Lifeline Lifeline -> Mapping [ctor prec 10] .
op emptyMapping : -> Mapping [ctor] .
op _*_ : Mapping Mapping -> Mapping
             [ctor assoc comm id: emptyMapping prec 20] .

op dom : Mapping -> LLSet .
op get : Mapping Lifeline -> Lifeline .

op subst : IOSet Mapping -> IOSet .
op subst : InteractionObligation Mapping -> InteractionObligation .
op subst : TraceSet Mapping -> TraceSet .
op subst : SDTrace Mapping -> SDTrace .
op subst : Event Mapping -> Event .
op subst : Msg Mapping -> Msg .

vars L L' : Lifeline .
vars MP MP' : Mapping .
var M : Msg .
var E : Event .
var T : SDTrace .
var S : Signal .
var K : EventKind .
var IO : InteractionObligation .
var IOS : IOSet .
vars P N TS : TraceSet .

eq dom(emptyMapping) = emptyLLSet .
eq dom(L -> L') = L .
ceq dom(MP * MP') = dom(MP) ; dom(MP')
    if MP =/= emptyMapping /\ MP' =/= emptyMapping .

eq get((L -> L') * MP , L) = L' .

eq subst(emptyIOSet, MP) = emptyIOSet .
ceq subst(IO & IOS, MP) = subst(IO, MP) & subst(IOS,MP)
    if IOS =/= emptyIOSet .
eq subst((P,N), MP) = (subst(P,MP),subst(N,MP)) .
eq subst(emptyTSet, MP) = emptyTSet .
ceq subst(T + TS, MP) = subst(T,MP) + subst(TS,MP)
    if TS =/= emptyTSet .

eq subst(emptyTrace, MP) = emptyTrace .
ceq subst(E . T, MP) = subst(E, MP) . subst(T, MP)
    if T =/= emptyTrace .
eq subst((K, M), MP) = (K, subst(M,MP)) .
eq subst((S, L, L'), MP) =
   (S, if L in dom(MP) then get(MP,L) else L fi,
       if L' in dom(MP) then get(MP,L') else L' fi) .
```

```
endm

mod REFINEMENT is
  including MAPPING .

  ops _~>r_ _~>rr_ : InteractionObligation InteractionObligation -> Bool .
  ops _~>g_ _~>rg_ _~>l_ _~>l'_ _~>rl_ _~>rl'_ : IOSet IOSet -> Bool .
  ops _~>g_ _~>rg_ : InteractionObligation IOSet -> Bool .
  ops _~>l'_ _~>rl'_ : IOSet InteractionObligation -> Bool .

  vars P1 N1 P2 N2 : TraceSet .
  vars IOS1 IOS2 : IOSet .
  vars IO1 IO2 : InteractionObligation .

  eq (P1,N1) ~>r (P2,N2) = subseteq(P1, P2 + N2) and subseteq(N1, N2) .
  eq (P1,N1) ~>rr (P2,N2) = (P1,N1) ~>r (P2,N2) and subseteq(P2,P1) .

  eq emptyIOSet ~>g IOS2 = true .
  ceq (IO1 & IOS1) ~>g IOS2 = (IO1 ~>g IOS2) and (IOS1 ~>g IOS2)
      if IOS1 =/= emptyIOSet .
  eq IO1 ~>g emptyIOSet = false .
  eq IO1 ~>g (IO2 & IOS2) = (IO1 ~>r IO2) or (IO1 ~>g IOS2) .

  eq emptyIOSet ~>rg IOS2 = true .
  ceq (IO1 & IOS1) ~>rg IOS2 = (IO1 ~>rg IOS2) and (IOS1 ~>rg IOS2)
      if IOS1 =/= emptyIOSet .
  eq IO1 ~>rg emptyIOSet = false .
  eq IO1 ~>rg (IO2 & IOS2) = (IO1 ~>rr IO2) or (IO1 ~>rg IOS2) .

  eq IOS1 ~>l IOS2 = (IOS1 ~>g IOS2) and (IOS1 ~>l' IOS2) .
  eq IOS1 ~>l' emptyIOSet = true .
  ceq IOS1 ~>l' (IO2 & IOS2) = (IOS1 ~>l' IO2) and (IOS1 ~>l' IOS2)
      if IOS2 =/= emptyIOSet .
  eq emptyIOSet ~>l' IO2 = false .
  eq (IO1 & IOS1) ~>l' IO2 = (IO1 ~>r IO2) or (IOS1 ~>l' IO2) .

  eq IOS1 ~>rl IOS2 = (IOS1 ~>rg IOS2) and (IOS1 ~>rl' IOS2) .
  eq IOS1 ~>rl' emptyIOSet = true .
  ceq IOS1 ~>rl' (IO2 & IOS2) = (IOS1 ~>rl' IO2) and (IOS1 ~>rl' IOS2)
      if IOS2 =/= emptyIOSet .
  eq emptyIOSet ~>rl' IO2 = false .
  eq (IO1 & IOS1) ~>rl' IO2 = (IO1 ~>rr IO2) or (IOS1 ~>rl' IO2) .
endm
```