University of Oslo
Department of Informatics

MEADOW - A Dataflow Language for Modelling Large and Dynamic Networks

Cand. Scient. Thesis

Fredrik Seehusen

August 2003

Abstract

We address three main problems regarding the use of the traditional dataflow language (TDL) for modelling large and dynamic networks:

- *The problem of scalability*. The concepts and notations of TDL do not scale well. Thus TDL specifications may get large (space consuming) and chaotic.
- *The problem of generality*. TDL does not have the expressibility for specifying networks consisting of n (a general number) components. We distinguish between five different network topologies consisting n components that can not be specified in TDL. For point-to-point networks these are the star, ring and tree topologies, for multipoint networks the ring and the bus topologies.
- *The problem of expressing dynamic reconfiguration*. TDL is not well suited for the specification of dynamic networks. We distinguish between three kinds of dynamic networks: object-oriented networks, ad hoc networks, and mobile code networks.

Based on an examination of three state-of-the-art modelling languages (FOCUS, SDL-2000 and UML 2.0), we propose a language, MEADOW (Mod-Elling lAnguage for DataflOW) that essentially is an extension of TDL. Our hypothesis is that MEADOW successfully solves the problems mentioned above, and we argument by small examples and case studies.

Foreword

This thesis is submitted for the fulfilment of the *Cand. Scient.* degree in Informatics at the Department of Informatics, University of Oslo (UIO). The work on this thesis has been carried out at SINTEF Telecom and Informatics under supervision of Ketil Stølen.

I would like to thank Frode, Bjørn Håvard, Marit, Ole Andre, Ole Morten and Øystein for being good friends and for doing some spell-checking.

Most of all I would like to thank my adviser, Ketil Stølen for being a source of inspiration and for his skillful guidance and help throughout the whole process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A large number of modelling languages has been proposed for the development of computerised systems in the past 20 years [27]. Different modelling languages aid the specification of different system *properties* such as system function, system behaviour and system communication. In this thesis we aim to aid the specification of *system communication* by proposing a new language, MEADOW (ModElling lAnguage for DataflOW), for modelling networks. Specifically we consider large networks and dynamic networks.

In the following section we introduce central terms and put this thesis into context by briefly describing the field of research in which we aim to contribute. In sections 1.2 and 1.3 we motivate and give an overview of the thesis.

1.1 Background

Inspired by [3], [20], [11], [19] and [9], we define a network to be a set of *components* connected by *channels* over which the components can communicate by sending and receiving *messages*. Components may themselves consist of sub-components, thus making up a *component hierarchy*. We distinguish between *static* and *dynamic* networks. A static network is a network with a fixed structure that does not change over time, whereas a dynamic network is a network that is not static. Moreover, we consider three kinds of dynamic networks: (1) *object-oriented networks* which are networks in which components act as objects in object-oriented languages; (2) *ad hoc networks* which are networks where components and channels do not constitute a fixed structure partly due to the fact that they may enter or leave the network during computation; (3) *mobile code networks* which are networks where sub-components may be sent from one component to another.

The field of research in which we aim to contribute is *system engineering* which according to [25] can be defined as "the activity of specifying, designing, implementing, validating, deploying and maintaining systems *as a whole*". Several different *system engineering methods* have been developed in order to achieve the goal of cost-effective development of quality systems. A system engineering method is a structured approach to system development, and all methods are based on the idea of developing models of a system which may be represented graphically, and using these models as a system specification or design. Indeed, modelling has been the cornerstone in many traditional software development methodologies for decades, and a large number of different languages and approaches have been developed [14].

Examples of languages that can be used to model networks are context diagrams, object communications diagrams, JSD system network diagrams [27], SDL [11], FOCUS [3] and UML [19]. All these languages can be seen as extensions of the traditional dataflow language (TDL). We address limitations of TDL when used to model the structure of large and/or dynamic networks. In particular, aiming to overcome these limitations, we propose a new language, MEADOW (ModElling lAnguage for DataflOW) which essentially is an extension of TDL. MEADOW has concepts such as component types/instances, generalisation, parameterised components for increasing scalability in a specification as well as a number of concepts for specifying relationships between components.

1.2 Motivation

Our overall motivation is the need for cost-effective development of quality systems. Specifically, as mentioned previously, we want to overcome limitations in TDL when used to model large and/or dynamic networks. There are three main problems that we wish to overcome: (1) *the problem of scalability*, which is that TDL specifications can get space consuming and chaotic when used to describe large networks; (2) *the problem of generality*, which is that TDL cannot specify network topologies consisting of an arbitrary number of components. We distinguish between five topologies that TDL cannot specify in the general case: the star, tree and ring topologies for point to point networks, and the bus and ring topology for multipoint networks; (3) *the problem of specifying dynamic reconfiguration*, which is that TDL does not include concepts for specifying dynamic properties, and therefore is unsuited as a means for specifying dynamic networks.

Our motivation is to improve and extend the concepts of current state-of-the-art modelling languages (that may be seen as extensions of TDL) with respect to the problems described above.

1.3 Overview

Chapter 2 introduces and explains terms and concepts that are central in this thesis. In the first section we introduce network terminology, then based on this introduction we define a conceptual terminology framework. This framework will be the basis for describing our language domain.

Chapter 3 describes the purpose and the target group of MEADOW. A brief discussion on language quality is presented, as well as a discussion on the limitations of TDL for describing our language domain. On the basis of these discussions we formulate four success criteria that can be used in order to assess how successfully a given dataflow language solves the three problems we address. Finally, we suggest *how* the success criteria can be assessed.

Chapter 4 presents an evaluation of three state-of-the-art modelling languages, FOCUS [3], SDL [11] and UML [19], with respect to three of the previously mentioned success criteria. In the end of the chapter we compare the evaluation results.

Chapter 5 introduces the concepts of MEADOW, and explains these by small examples.

Chapter 6 employs MEADOW in a number of small case studies.

Chapter 7 presents an evaluation of MEADOW with respect to the success criteria that are defined in chapter 3.

Chapter 8 concludes and summarises the main results of the thesis and suggests areas of future work.

Chapter 2

Terminology

In the following we introduce and explain terms and concepts that are central in this thesis. First we introduce central networking terms as they are defined in literature, then we present a conceptual terminology framework that allows us to define the language domain of MEADOW.

2.1 Network Terminology

According to [20], a *computerised network* is a collection of interconnected computers or computerised equipment (*components* or *nodes*). Network connectivity occurs at many different levels. At the lowest level, a network can consist of two or more components directly connected by some physical medium (often called a *link*). At higher levels, connectivity between two components does not necessarily imply direct physical connection between them - indirect connectivity may be achieved among a set of cooperating components. Figure 2.1 shows how indirect connection can be achieved. Here, the components that are attached to at least two links run software that forwards data received on one link out on another. The cloud in figure 2.1 distinguishes between the components on the inside that *implement* the network (commonly called *switches*) and the components on the outside of the cloud that *use* the network (commonly called *hosts*). The cloud is one of the most important icons of computer networking [20]. In general, a cloud can be used to represent any type of network.

We can also view the network as providing logical *channels* over which application-level processes can communicate with each other. “In other words, just as we use a cloud to abstractly represent connectivity among a set of computers, we now think of channels as connecting one process to another” [20]. Figure 2.2 shows a pair of application-level processes communicating over a logical channel. The channel is in turn implemen-

ted on top of a cloud that connects a set of hosts.

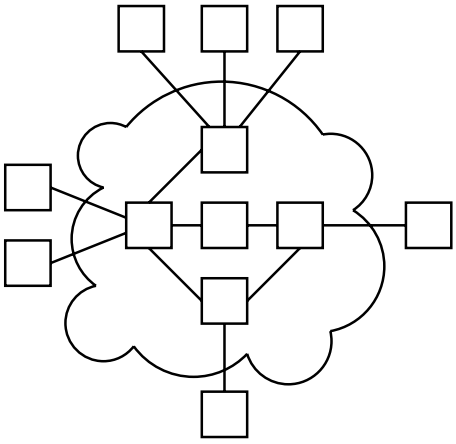

Figure 2.1: Switched network.

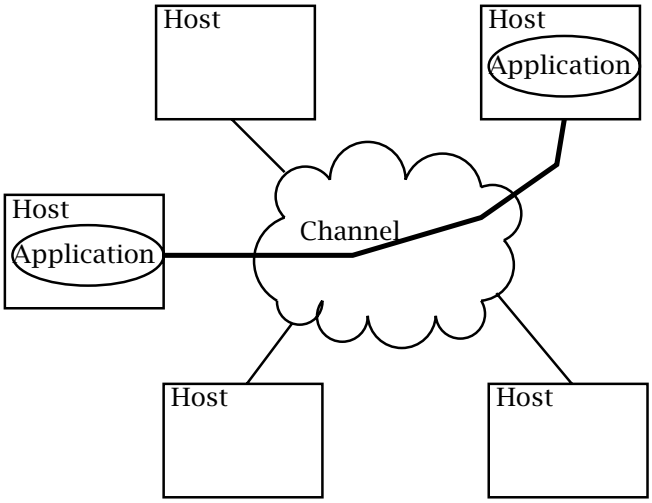

Figure 2.2: Process communication over an abstract channel.

2.1.1 Network Architectures

To help deal with the complexity that often occurs when designing networks, network designers have developed general blueprints - usually called a *network architecture* - that guide the design and implementation of networks. Network architectures often defines a partitioning of network functionality into *layers of abstraction*. The general idea of this

kind of abstraction is that you start with the services offered by the underlying hardware, then add a sequence of layers that provide a higher (more abstract) level of services. The services provided at the high layers are implemented in terms of the services provided at the lower layers. The abstract objects that make up the layers of a network system are called *protocols*. A protocol defines two interfaces. First, it defines a *service interface* to other objects (on the same computer for example) that wants to use its communication services. Second, a protocol defines a *peer interface* to its counterpart (peer). This second interface defines the form and meaning of messages exchanged between protocol peers.

Using the terms introduced thus far, we can for example define four network system layers: Hardware, host-to-host connectivity, process-to-process channels and application programs. This is illustrated in figure 2.3.

The two most widely referenced architectures are the OSI architecture [6] and the Internet architecture. For a more detailed description of these we refer to [20].

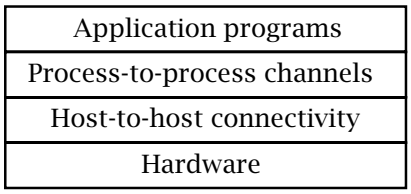

Figure 2.3: Example of a layered network system.

2.1.2 Network Topologies

The network topology refers to the way in which components are connected to each other [21]. If communication is established between two components through a direct channel, one can speak of a point-to-point channel; a network whose components communicate over point-to-point channels is called a *point-to-point network*. On the other hand, if communication is rather broadcasted from one component to several components, it is a case of a multipoint channel; a network whose components communicate via multipoint channels is called a *multipoint network* [21].

Point-to-point networks can have a star, tree, ring or mesh topology. In a star topology, all components are related by a point-to-point channel to

a common central component called the *star centre*. In a tree topology, the network is hierarchically structured with a top component called a *tree root*. In a ring topology, all components are related to form a closed ring. A mesh topology is formed by a number of channels such that each component pair of the network is connected by more than one path.

Multipoint networks can have a bus or a ring topology. In a bus topology, each component is set linearly on a channel. Messages are transmitted by any component through the entire bus in order to reach the other components of the network. In a ring configuration, all the components are set on a closed circuit formed by a series of point-to-point channels. These components form a ring.

2.1.3 Dynamic Networks

Networks may be static or dynamic. In the following we introduce terms related to dynamic networks. We separate between three kinds of dynamic networks: *object-oriented networks*, *ad hoc networks* and *mobile code networks*. We explain each of these in turn.

Object-oriented networks are networks in which components are implemented by *objects* in the sense of object-oriented programming languages. There are many object-oriented programming languages today, examples are Simula, Smalltalk, C++ and Java to name but a few. Important object-oriented concepts are *objects*, *classes* and *inheritance*. Objects are basic uniquely identifiable run-time entities that can be dynamically created or destroyed during execution. An object can invoke *methods* on other objects via references called pointers. A class defines a set of possible objects, and from the point of view of a strongly typed language, a class can be seen as a construct for implementing a user-defined type [12]. “Inheritance is a relation between classes that allows for the definition and implementation of one class to be based on that of other existing classes” [12].

“An *ad hoc network* is a dynamically reconfigurable wireless network with no fixed infrastructure or central administration” [1]. Components in these networks move arbitrarily; thus, network topology changes frequently and unpredictably. Current cellular networks rely on a wired infrastructure to connect different cells. In ad hoc wireless networks, however, a remote mobile component interconnection is achieved via a peer-level multihopping technique [17]. Moreover, each component in an ad hoc network is willing to forward packets to other components that cannot communicate directly with each other [2]. A classic example of ad hoc is a network of war fighters and their mobile platforms in battle-

fields.

Mobile code networks are networks which accommodate *code mobility*. Despite the wide-spread interest in mobile code technology and applications, the field is still quite immature. A sound terminological framework is still missing. However, according to [7], code mobility can be defined informally as the capability to dynamically change the bindings between code fragments and the location where they are executed. As an example of an application area, the research work on distributed operating systems is concerned with the ability to support the migration of active processes and objects (along with their state and associated code) at the operating system level. In particular, *process migration* concerns the transfer of an operating system process from the machine where it is running to a different one, and *object migration* makes it possible to move objects among address spaces [7].

2.2 A Conceptual Terminology Framework

In order to define a conceptual terminology framework that captures all network layers, we abstract from the distinction between physical and logical layers. Hence forth, a network is a set of *components* and a set of *channels* over which the components communicate. Components are the interconnected entities that a network consists of, channels are the connections between components that can occur at any level of communication.

Networks that consist of computers that are connected by fixed wires, or networks that consist of application processes connected by TCP/IP, are essentially represented in the same way. As we use the term *component* to generalise the exact nature of the communicating entities a network consist of, so do we use the term *channel* to generalise the exact nature of how components are connected.

2.2.1 Basic Conceptual Terms

The UML class diagram depicted in figure 2.4 specifies the basic conceptual terms. As specified in the diagram, we distinguish between two types of components: *elementary components* and *composite components*. Elementary components do not contain sub-components, while the composite components contain sub-components and channels over which they communicate. Looking at figure 2.4 again, we see that each *channel* has exactly two *ports*. Components reference ports in order to receive or transmit messages on channels. These references represent

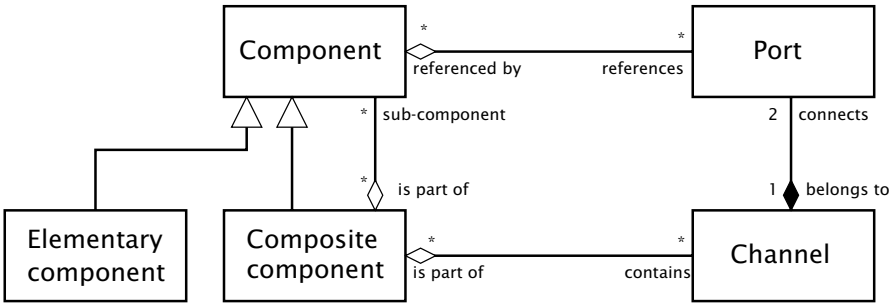

Figure 2.4: Basic conceptual terms.

interfaces between a component and its environment or between a composite component and its sub-components.

In the following we define the terms in figure 2.4. The definitions are inspired by [3], [11], [19] and [9].

Component Entity (such as for example a computer, a chip or an application process) that communicates with its environment through a set of referenced ports. A component may be created or killed during computation and may be sent from one component to another via channels. The duration of time from the creation of a component to its death is called the *lifetime* of the component. A component may have a *behaviour* which defines (1) how messages that are received by its referenced ports may be handled and (2) how messages are output on its referenced ports. A component can be elementary or composite.

Elementary Component A component that does not contain internal sub-components or channels.

Composite Component A component that contains sub-components. A composite component may contain channels over which its sub-components may communicate. The ports that are referenced by a composite component represent (1) the interface between the environment of the composite component and its sub-components or (2) the interface between the environment of the composite component and its behaviour, i.e. messages received on these referenced ports are not sent to sub-components directly. A composite component may communicate with its sub-components.

Port Provides an interface between a component and its environment or between a composite component and its sub-components. This

makes it possible to specify a component without any knowledge of the environment it will be embedded in [19]. A port is either an *input-port* or an *output-port*. The former receives messages from a channel, whereas the latter transmits messages to a channel. By convention, the name of an input-port is equal to the name of its channel prefixed by the '?'-character, and similarly the name of an output-port is the name of its channel prefixed by the '!'-character. A port is created when its channel is created, and it is killed when its channel is killed. A reference to a port may be sent from one component to another via a channel.

Channel A channel represents the forwarding of messages from an output-port to an input-port, hence a channel is *directed*. A channel is *shared* if any of the ports it connects are referenced by more than one component. A channel may or may not allow message overtaking and message disappearance. A channel is created when its containing composite component is created. A channel may also be created or killed during computation.

Example: O-O Net

In the following example we demonstrate how the terminology introduced above can be used to model a network called O-O Net. O-O Net contains three objects, A, B and C. At the beginning of computation (time 0), B and C has no references to other objects, while A has a reference $!c_1$ to B and a reference $!c_2$ to C. During computation, at time 1, object A sends a copy of $!c_2$ B. At time 2, A removes $!c_1$ from its set of port references.

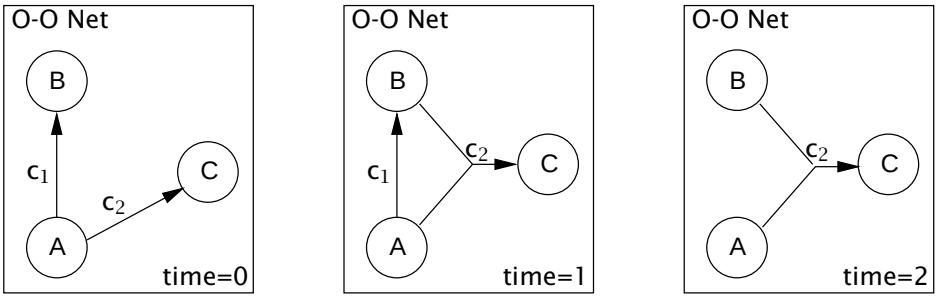

Figure 2.5: O-O Net

Figure 2.5 illustrates O-O Net at the three different points in time. In the figure, the box represents a composite component, the circles represent components and the directed edges represent channels. Ports are not

shown because they can be derived from the configuration of the channels.

Time 0. The composite component O-O Net is created along with the components and channels that are contained in O-O Net. From the configuration of the channels we can derive that object A references two output-ports, $!c_1$ and $!c_2$. Furthermore B references an input-port $?c_1$ and C references an input-port $?c_2$.

Time 1. Object A sends a copy of $!c_2$ to object B. Channel c_2 is modified accordingly. In the figure, the merging of the two lines from A and B into a single directed line going to C illustrates the fact that A and B reference the same output-port ($!c_2$). How messages sent from A and B along channel c_2 are merged is not specified.

Time 2. Object A kills its reference $!c_1$. Consequently channel c_1 is killed because its output-port is not referenced by any component. Notice that channel c_1 would have formed a feedback connection if $!c_1$ had been referenced by B.

2.2.2 Static, Dynamic and Mobile Networks

Based on the terms introduced in the previous subsection, we define static network and the different kinds of dynamic networks specified in the UML class diagram in figure 2.6. Note that the following terms are defined with respect to a *model* of a network. Consequently, whether we say that a network is dynamic or not depends on the level of abstraction we choose to model it from, and not necessarily on the physical infrastructure of the network for example.

Network A set of components and channels over which the components may communicate.

Static network A network is *static* if the sets of references to ports and sub-components of any of its components remain constant throughout any computation. Hence, in a static network, components and channels are neither created nor killed during computation.

Dynamic network A network that is not static.

Mobile network A network is mobile if it at some point in time contains two components C_1 and C_2 such that C_1 may enter or leave C_2 during the lifetime of C_1 and C_2. In other words, a mobile network is a network that contains a component that *moves/migrates* from

one composite component to another and thereby changing the set of composite components it is a part of.

Object-oriented network A dynamic network in which: (1) channels and components may be created and killed, (2) each component references a single input-port and may reference many output-ports and (3) references to output-ports may be sent along the channels. In other words, a component represents an object, and the single input-port contained by the component represents a unique object identifier. The output-ports referenced by a component represents object identifiers to other components. The fact that references to output-ports may be sent along the channels represents the fact that pointers/references may be passed on from one object to another.

Ad hoc network A mobile network in which channels and components are created or killed during computation. Ad hoc networks have no fixed infrastructure and no central administration. Ad hoc networks are mobile in the sense that components may move from one composite component to another. This allows us to model the fact that a component (f.ex. a mobile telephone) may move from the transmission radius of one component (f.ex. a base station) to the transmission radius of another component (f.ex. another base station). Ad hoc networks are not mobile code networks.

Mobile code network A mobile network where mobility is only due to components being sent from one component to another via channels. The distinction between a mobile code network and a mobile network is that a mobile network allows components to move from one composite component to another without being transported on a channel, but in a mobile code network component movement must occur via channel transportation. A component that is sent from one component to another may for example be an active process at the operating system level.

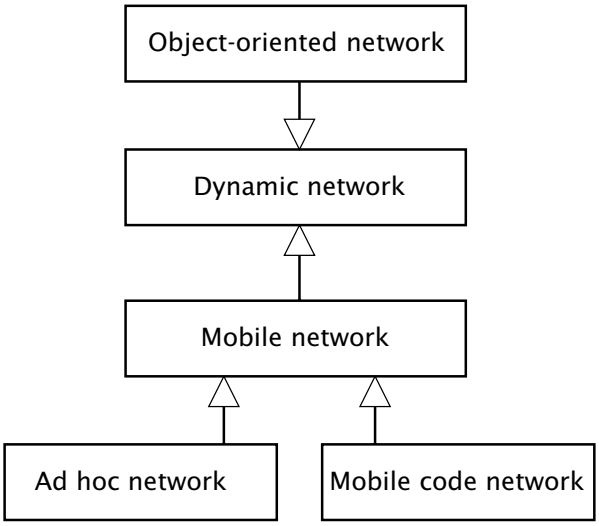

Figure 2.6: Classification of dynamic networks.

Chapter 3

Problem Analysis

In this chapter we explain the purpose of MEADOW, then on the basis of an introduction to language quality and a discussion of the problem domain of this thesis, we list four success criteria for modelling languages. Our overall hypothesis is that MEADOW fulfils these criteria, and at the end of this chapter we propose a strategy for how to provide evidence for its validity.

3.1 Purposes and Target Groups

There is a need to clarify to what purpose MEADOW should be used and what target group MEADOW is aimed at. This is necessary in order to select the right balance between the level of *understandability* and the level of *expressibility* that we want MEADOW to achieve.

MEADOW is a graphical language intended to aid the development aspects related to computerised networks by providing a way to model computerised networks. For example, much research has been devoted to the development of routing algorithms for ad hoc networks, and MEADOW (in combination with other languages) could be used to test these algorithms. Specifically however, MEADOW is intended to model:

- the infrastructure of static and dynamic networks;

Furthermore, MEADOW is intended to be used/understood by *developers* of computerised networks as well as *non-developers*. This suggests that the level of comprehensibility of MEADOW should be high.

For clarity, there are three specific aspects that MEADOW is not intended to model:

- MEADOW is not intended to model the *behaviour* of components in networks. This because many languages exist today that can

specify behaviour, and because we want to limit the scope of this thesis. Having said that, we do want MEADOW to be used in combination with other languages for specifying behaviour.

- MEADOW is not intended to model distances in physical space. This is abstracted away because of scalability issues.
- MEADOW is not intended to model communication between a composite component and its sub-components, i.e. we focus on peer-to-peer communication.

3.2 Language Quality

As mentioned in the previous section, we want MEADOW to have a high level of comprehensibility appropriateness, but there are also several other aspects of language quality that are worth taking into consideration when developing a modelling language.

The paper [14] presents a quality framework for evaluating the quality of modelling languages. Further details on the framework can also be found in [4], [13] and [15]. Five areas for language quality are identified, with aspects related to both the underlying (conceptual) basis of the language and the external (visual) representation of the language:

Domain appropriateness. This area address to what extent the conceptual basis of a language is able to express the intended language domain.

Participant language knowledge appropriateness. This area relates the participant (language user) knowledge to the language. The conceptual basis of a language should correspond as much as possible to the way individuals perceive reality.

Knowledge externalizability appropriateness. "This area relates the language to the participant knowledge. The goal is that there are no statements in the explicit knowledge of the participant that cannot be expressed in the language" [14].

Technical actor interpretation appropriateness. This area relates the language to the technical actor/developer interpretations. "For the technical actors, it is especially important that the language lend itself to automatic reasoning" [14].

Comprehensibility appropriateness. We describe this area in more detail since we want MEADOW to have a high comprehensibility. According to

[14], for the conceptual basis of a language the following aspects related to comprehensibility are important:

- The phenomena of the language should be easily distinguishable from each other.
- The phenomena must be general rather than specialised.
- The phenomena should be composable, i.e. one should be able to group statements in a natural way.
- The language must be flexible in precision.
- The use of phenomena should be uniform throughout the whole set of statements that can be expressed within the language.
- The language must be flexible in the level of detail.

The following aspects are important for the external representation of the language:

- Symbol discrimination should be easy.
- It should be easy to distinguish which of the symbols in a model any graphical mark is part of.
- The use of symbols should be uniform. This means that a symbol should not represent one phenomenon in one context and another one in a different context.
- One should strive for symbolic simplicity.
- The use of emphasis in the notation should be in accordance with the relative importance of the statements in the given model.
- Composition of symbols should be made in an aesthetically pleasing way. The language should not, for example, give rise to unnecessarily many line intersections.

3.3 The Problem Domain

We discuss the problem domain that we wish to address. The basis for this discussion is the limitation of traditional dataflow language (TDL) for describing the language domain that was described in section 2.2. We consider three problems: the problem of scalability, the problem of generality and the problem of expressing dynamic reconfiguration. We describe each of them in turn, but first we explain what we mean by TDLs.

3.3.1 The Traditional Dataflow Language (TDL)

The use of the term *dataflow diagram* differs with respect to different sources of literature. See [3] and [27] for example. The definition we will use however, is based on [3]. Here, a dataflow diagram is simply a directed graph in which the nodes represent components, and the edges represent channels, i.e. possible component communication. The concepts used in this basic diagram is what we refer to as the traditional dataflow language. TDL diagrams show a set of possible communications without indicating any sequence. Furthermore, it shows communication only, and does not refer to a particular run of a system. [27]

A simple example is given in figure 3.1. Here each network component is represented by a box. The interaction between the network components is expressed by the arrows.

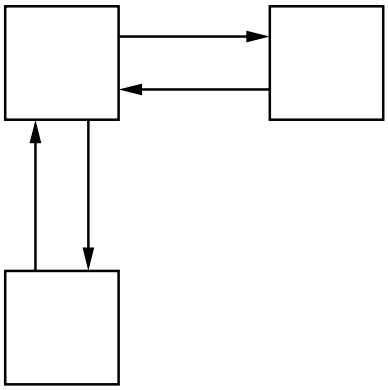

Figure 3.1: Example of a TDL diagram.

3.3.2 The Problem of Scalability

There are two scalability problems that may arise when TDL is used to model large networks:

1. The models may get large (space consuming).
2. The models may get chaotic.

These problems occur for two reasons: (a) because TDL have few concepts of *abstraction* and (b) because TDL lacks concepts of *structuring* patterns in a specification that may increase scalability.

The idea of an abstraction is to define a unifying model that can capture some important aspect of a system and hide irrelevant details. In other words, one can use abstraction as a way to handle higher complexity without being drowned in details. An example of abstraction is given in figure 3.2, where the the composite component A is seen through a *black-box view* on the left, and a *glass-box view* on the right. The internal structure is abstracted away in the specification on the left side. The terms *black-box* and *glass-box* are used in FOCUS [3], in STATECHARTS [10] they are known as *clustering* and *refinement*.

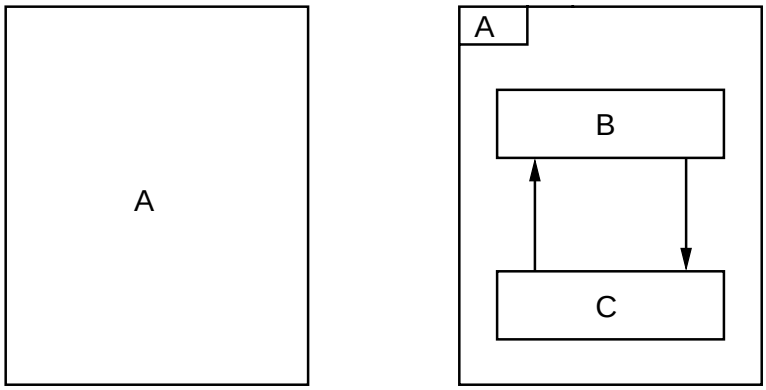

Figure 3.2: Black-box view (left). Glass-box view (right).

There are also ways of enhancing the scalability of specifications without using abstraction. As mentioned, TDL lacks concepts of structuring patterns in a specification. If one for example wants to specify 100 similar components, one has to draw up 100 boxes. As an example of principle, consider a network consisting of four similar computers that are (for simplicity) not connected. This is specified on the left side of figure 3.3 with TDL notations. On the right side of figure 3.3, we have specified the same network with SDL [11] notations. The label C(4):Computer means that four instances of Computer is specified. If we compare the two specifications in figure 3.3, it becomes obvious that the specification on the right side scales better than the one on the left side. Note that this is not an example of abstraction, since both specifications in essence are equivalent, i.e. no information is abstracted away in the SDL specification.

3.3.3 The problem of Generality

In TDL, one may only specify a fixed number of components or channels. However, it is sometimes necessary to specify networks that consist of

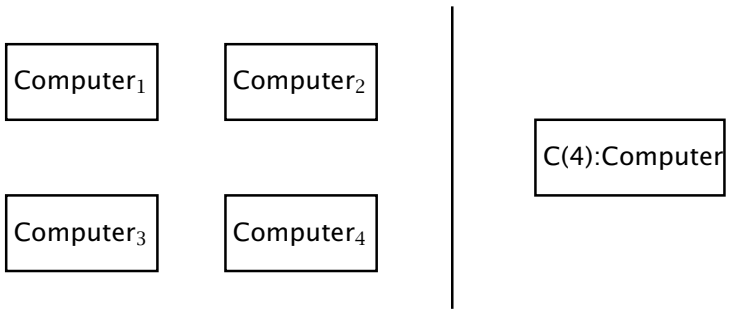

Figure 3.3: TDL specification (left). SDL specification (right).

n components or n channels where $n \in \mathbb{N}$. The problem is that it would be impossible to specify such a network in TDL without introducing additional concepts.

Some network topologies that do not consist of a fixed number of components are easier to describe than others. To see this, compare the star and the ring topology for example. On one hand, a star topology can easily be specified using the concept of a one-to-many relationship which is supported by many specification languages such as for example FOCUS [3], UML [19] and SDL **??**. On the other hand, none of these languages can specify (graphically) a ring topology with n components precisely.

As mentioned earlier in chapter two, network topologies can be classified into star, ring, tree and mesh for point-to-point networks, and ring and bus for multipoint networks [21]. In figure 3.4 we have attempted to illustrate these topologies in the general case, i.e for networks that do not have a fixed number of components. We have left out the mesh topology because the relationship pattern between the nodes of this topology is too weak for a desirable generalisation. In the general case the star, the bus and the two ring topologies all consists of n nodes that are structured in a fairly straight forward manner. The tree topology is a little different. Here the root node is connected to m sub-nodes, and each of these sub-nodes are in turn connected to a different number of sub-nodes of their own. We say that the tree is *uniform* if all nodes except the leaf-nodes are connected to an identical number of sub-nodes. Otherwise the tree is *non-uniform*. If for example the tree in figure 3.4 is uniform, then we would have that $m = n = o = p$. The depth of the tree in figure 3.4 is 3, but in the truly general case we can imagine such a tree having a depth of d.

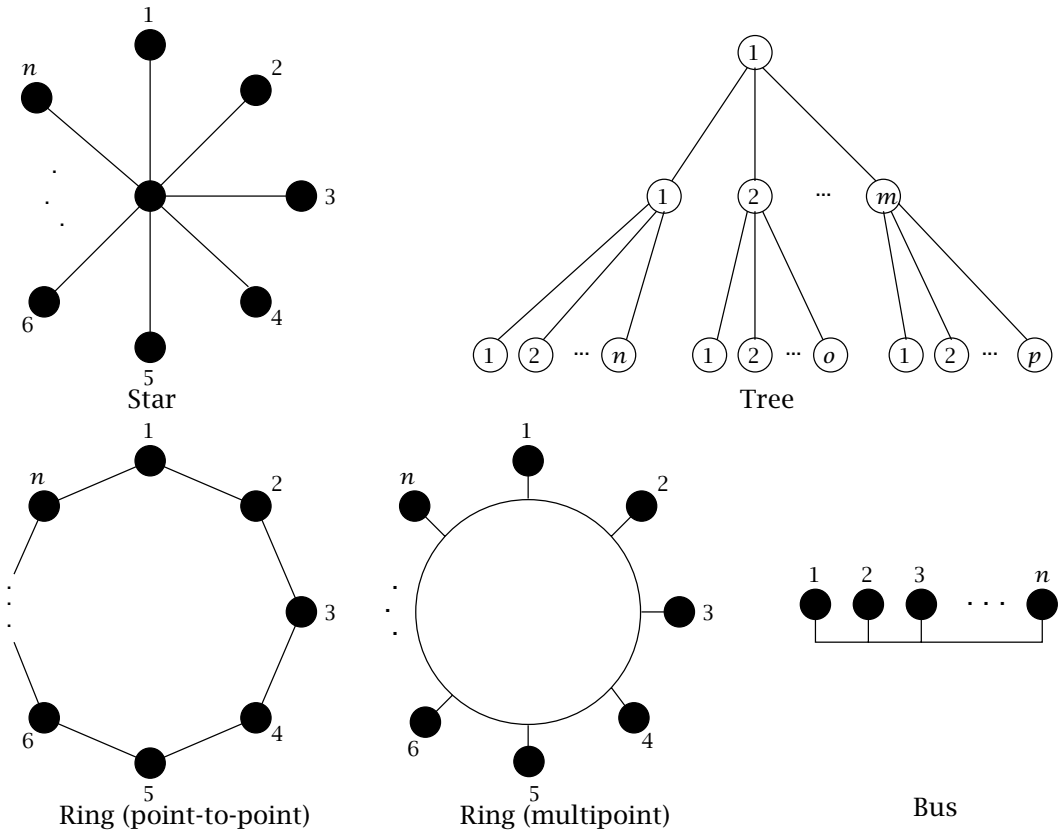

Figure 3.4: Network Topologies.

3.3.4 The Problem of Expressing Dynamic Reconfiguration

Components and channels specified in TDL are statically fixed, hence TDL is not well suited to model dynamic networks where components and channel may be created and killed. We separate between three different types of dynamic networks that TDL lacks concepts to specify.

1. *Ad hoc networks*.
2. *Mobile code networks*.
3. *Object-oriented networks*.

3.4 Overall Hypothesis

Having described (1) what we intended to model, (2) examined language quality and (3) described the domain specific problems we wish to address, we list four success criteria for a given dataflow language **DL**. Our hypothesis is that MEADOW fulfils these success criteria.

Success Criteria

The success criteria for a given dataflow language **DL** are:

- **DL** should have a high comprehensibility appropriateness;
- **DL** should handle the problem of scalability;
- **DL** should handle the problem of generality;
- **DL** should be able to specify object-oriented networks, ad hoc networks and mobile code networks.

3.5 Scientific Methods

In this section we discuss different scientific methods that might be used to argument that the success criteria are fulfilled.

According to [18], research evidence is gathered to maximise three things:
(A). The generalizability of the evidence over populations of actors.
(B). The precision of measurement of behaviours.
(C). The realism of the situation or context.

Eight research strategies that each have different strengths and weaknesses with respect to (A), (B) and (C) are summarised below. These

methods are taken from the book [18] where they are presented as a way to study groups in social science, but they may also be used in computer science.

- *Field studies* refer to efforts to make direct observations of ongoing systems. "Field studies gain realism (C) at the price of low generalizability (A) and lack of precision (B)"[18] In a computer science setting, these ongoing systems might be running software or hardware in a certain work setting. This method might for example be used to study the usability of a groupware program in a work setting by observing how users use the groupware.
- *Laboratory experiments* are attempts to create and maximise the "essence" of some general class of systems by controlling the extraneous features of the situation. This method is high on precision of measurement, and might be used in a computer science setting to assess attributes of a class of networks, algorithms, hardware components et cetera. An example of a laboratory experiment is to create a computer network and manipulate the traffic to study how new queueing algorithms in routers effects latency in the network.
- *Field experiments* are similar to field studies, but with one major difference; the deliberate manipulation of some feature whose effects are to be studied. This method might be carried out in a computer science setting for example by observing the effects of deliberately increased workload of the computer system in a certain workplace.
- *Experimental simulations* is a laboratory study in which an effort is made to create a system that is like some class of naturally occurring systems. In a computer science setting this method might be used to assess certain attributes of software, hardware, algorithms, et cetera. One might for example study the use of a particular groupware in a created work environment in order to assess attributes of the groupware.
- *Sample surveys* are efforts to get information from a broad and well devised sample of actors. This method is high on generalizability, and might be used in computer science to assess how certain attributes of something (f.ex. software, hardware, programming languages etc.) are considered by a population as a whole.
- *Judgement studies* are efforts to get responses from a selected sample of "judges" about a systematically patterned and precisely calibrated set of stimuli. Judgement studies, as opposed to sample surveys, are considered to be high on precision of measurement, but low on generalizability. This method might be used in a computer science setting, for example to study how a groupware has affected work effectivity by getting precise response from a causally selected sample of users.
- *Formal theory*. Argumentation based on general formal theory is often high on generalizability. It is not very high on realism of context. In a

computer science setting, formal theory might be used for precise argumentation, for example by using mathematics.
- *Computer simulations* are attempts to *model* a specific real life system or class of systems. This method might be used in a computer science setting by simulating a real computer network to study congestion in the network.

3.5.1 Verifying the Success Criteria

In this section we describe how we assess the fulfilment of the success criteria given previously with respect to a given dataflow language **DL**, that can be seen as an extension of the traditional dataflow language (TDL).

*"**DL** should have a high comprehensibility appropriateness."*

The fulfilment of this criteria is assessed by evaluating both the underlying concepts and the visual representation of **DL** with respect to the comprehensibility appropriateness aspects that were listed in section 3.2. This evaluation should ideally be based on empirical studies such as sample surveys and judgement studies. However, this would be too time consuming in the context of this thesis, so we argument by examples (a form of field study) and explain in natural language how **DL** relates to the comprehensibility appropriateness aspects.

*"**DL** should handle the problem of scalability."*

In order to assess how well **DL** fulfils this criteria, we examine the scalability concepts of **DL** that constitutes the extension of TDL. In particular this applies to concepts of abstraction and concepts for structuring patterns that may make specifications expressed in **DL** more scalable than specifications expressed in TDL. We then compare this examination with similar examinations of other languages.

*"**DL** should handle the problem of generality."*

To assess how well **DL** handles the problem of generality, we examine if and how **DL** can specify networks consisting of n components that have the following topologies: star, ring for point-to-point networks, and bus and ring for multipoint networks. In addition to this we examine how well **DL** can specify the tree topology in both uniform and non-uniform situations, with or without fixed depth. We then compare the results of this examination with the results of similar examinations of other modelling languages.

*"**DL** should be able to specify object-oriented, ad hoc, mobile code networks."*

In order to verify that a specification language **DL** fulfils this criteria, we examine the concepts **DL** has for specifying dynamic properties. Then we use **DL** to specify simple examples of ad hoc networks and mobile code networks. This will give an indication as to how well **DL** is able to describe these kinds of dynamic networks. We then compare the results of this examination with the results of similar examinations of other modelling languages.

Chapter 4

State-of-the-Art

We use the approach for verifying the success criteria that was described in section 3.5.1 on the parts of three state-of-the-art modelling languages that may be seen as an extension of TDL. These languages are FOCUS [3], SDL-2000 [11] and UML 2.0 [19]. We do not evaluate the criteria regarding comprehensibility appropriateness, since this is outside the scope of this thesis.

Sections 4.1 through 4.3 examine FOCUS, SDL and UML, respectively. In section 4.4 we compare the results of the examinations.

4.1 FOCUS

The FOCUS method [3] is a collection of specification techniques. Although there are many different *styles* of specification in FOCUS, we examine the *graphical style* which is the style most relevant to this thesis. In the graphical style, components and channels are described graphically in terms of dataflow diagrams. Each node in such a diagram represents a component specification.

An example of a FOCUS specification is given in figure 4.1. Here the component C is specified as having n input channels and n output channels.

4.1.1 Scalability

We found the following scalability concepts in FOCUS:

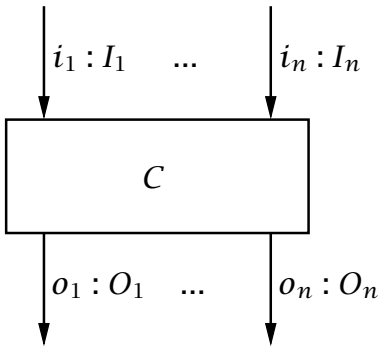

Figure 4.1: Example of a FOCUS specification.

Hierarchy

Hierarchy is achieved in FOCUS through the concept of *composition*, i.e. a component may *consist* of other components. The components that contain other components are called *composite components*, while components that do not consist of other components are called *elementary components*.

A composite component specification can be seen through a *black-box* or a *glass-box* view. When composite components are seen through a black-box view, the internal structure of a specified component is hidden (abstracted away), while a glass-box view allows us to see inside the component. These concepts provide a convenient way of abstracting away details in a specification, thus making specifications expressed in the language more scalable than specifications expressed in TDL.

Sheafs of Channels

A *sheaf of channels* can be understood as an indexed set of channels. If for example Cid is a set of identifiers, then one may specify as many channels s as there are elements in Cid by associating the label $s[Cid]$ with an arrow (which is the graphical representation of a channel/sheaf of channels). This concept can obviously improve the scalability of channel specification, since it allows a specifier to specify sets of channels instead of single channels.

Arrays of Channels

In addition to sheafs of channels, FOCUS provides another way of specifying channels. This concept, which is very similar to the concept of

sheafs of channels, is not named in FOCUS, but we name it *arrays of channels* (for lack of a better term).

An array of channels, as the name suggests, may be understood as an array of channels. Specifically, $i_1 ... i_n$ denotes the specification of n channels as illustrated in figure 4.1. An example of how sheafs of channels and arrays of channels can be used in combination is illustrated in figure 4.2.

Specification Replications

Sometimes networks contain numerous instances of the same component. *Specification replications* is a concept developed to exploit this in order to make specifications more scalable.

A specification where this concept is used is illustrated in figure 4.2. Here, C is a component specification uniquely defined by a constant, Cid is a set of component identifiers and $c \in Cid$. The specification contains exactly one instance of the component C for every identifier c. Note that the concept of sheafs of channels is also illustrated in figure 4.2.

Specification replications may increase the scalability of a specification because the concept enables a specifier to specify sets of components instead of single components.

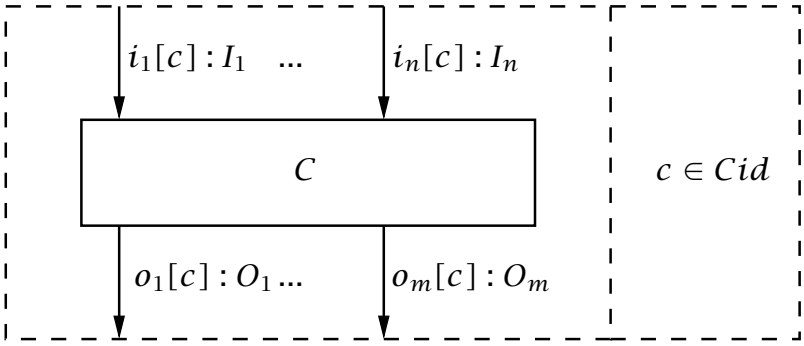

Figure 4.2: An example of specification replications.

Parameterised Specifications

Specifications can be parameterised explicitly by types and constants. This makes it possible to describe components that are schematically

similar and differ only in minor details. This concept can be used in combination with specification replications in order to overcome the limitation that all components in a specification replication must have the same internal structure. To see this, consider figure 4.3 that contains exactly one specification of the component $Server$ for every element in the set Sid. Each $Server$ component may take a different parameter, thus each $Server$ component may have a different internal structure. In this way parameterised specifications may increase the flexibility of specification replications, thus making specifications in FOCUS more scalable.

Dependent Replications

Dependent replications of specifications in FOCUS are introduced to handle nonuniform configurations. An example is given in figure 4.3, where $Server$ components are connected to Mmi components in a nonuniform manner, that is, two different $Server$ components can be connected to a different number of Mmi components. The nonuniform relationship between servers and $Mmis$ is specified by the auxiliary function f.

The concept of dependent replications increases the flexibility of sheafs of channels which, as mentioned previously, provides a scalable way of specifying connections between components.

4.1.2 Generality

It goes without saying that FOCUS has concepts that allow us to specify certain network topologies consisting of n components. However, this is not the case for all network topologies.

Point to Point Topologies

The *star* topology can be expressed using sheafs of channels and specification replications. If we look at figure 4.3 again, and ignore the Mmi components and all the channels going to and from the Mmi components, we see that a star topology is specified. Cnt would then be the *star centre*.

The *tree* topology can be specified for a fixed depth. Again we can use figure 4.3 as an example, because the network specified there may in fact have a tree topology. The depth of the tree is fixed and equal to 3. Furthermore, the tree can either be uniform or non-uniform depending on the definition of the auxiliary function f. Trees of arbitrary depths

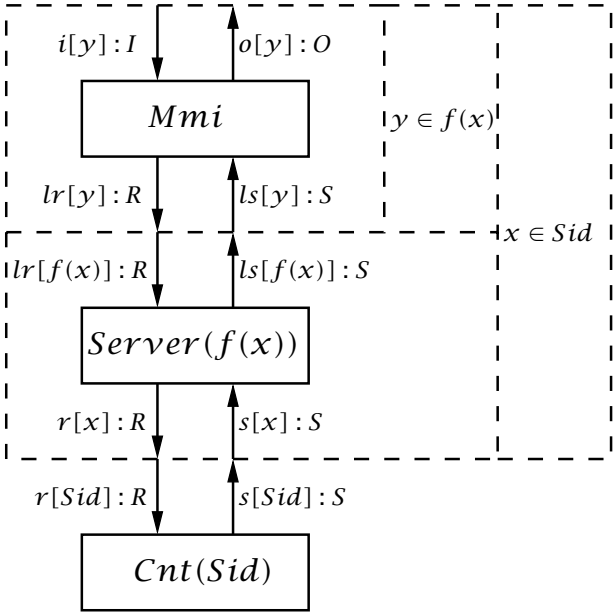

Figure 4.3: An example of dependent replications.

are not possible to specify in FOCUS.

The *ring* topology can not, in the general case, be specified in FOCUS. The reason is that such a specification would involve internal communication between specification replications, and this is not possible to specify in FOCUS.

Multipoint Topologies

It is possible to specify the bus or the ring topology for multipoint networks in FOCUS. But this can only be done for a fixed number of components. Specification of these topologies in the general case in not possible, since this would involve internal communications between specification replications.

4.1.3 Dynamic Reconfiguration

It is possible to specify a snapshot of the structure of a dynamic network, i.e the structure of a dynamic network at a given point in time. But FOCUS has no concepts for graphically specifying how the structure of a network may change over time.

4.2 SDL-2000

The Specification and Description Language (SDL) is a formal and visual modelling language standardised by the International Telecom Union (ITU), intended for unambiguous specification and description of telecom, distributed and embedded systems. The language is based on finite state machines and includes concepts for behaviour and data description and concepts for complex system structuring in addition to a visual action language and an execution model [16].

The language has been evolving since 1980 with updates in 1984, 1988, 1992, 1996 and 1999. Our examination is based on the update made in 1999, known as SDL-2000 [11].

SDL is used for the specification of systems. An SDL system consists of:

- One or more *agents*. An agent may be of the kind *system*, *block* or *process*. A block may contain other blocks or processes, while a process can only contain other processes (not other block). System is a special case of the outermost block.
- *Gates* that agents communicate through.
- *Channels* that connect gates.

An example of an SDL specification is given in figure 4.7, where three block agents named Panel, Door and Controller are represented by boxes with the appropriate labels. The channels connecting the blocks (actually the gates, but they are suppressed) are represented by the arrows.

In the following we examine in more detail how systems are are structured in SDL. We do not examine SDL concepts for specification of process behaviour or other aspects of SDL, since this is outside the scope of this thesis.

4.2.1 Scalability

The following scalability concepts were found in SDL:

Hierarchy

A block can be a container for a substructure of blocks connected by channels. Each of these blocks may in turn consist of a substructure of blocks or processes. This decomposition may be applied to any depth, thus making it possible to specify a hierarchy of blocks.

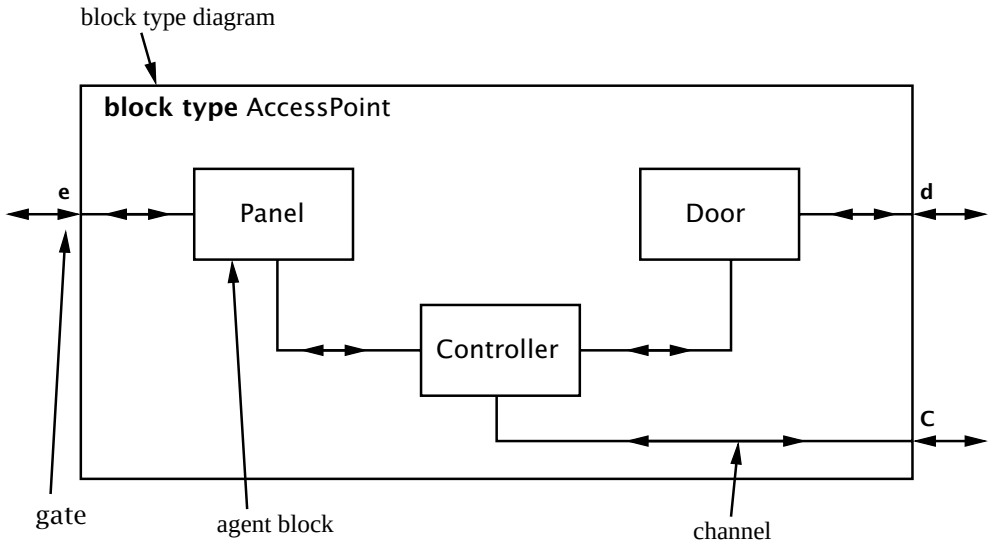

Figure 4.4: Example of an SDL specification.

Types, Instances and Sets

Agent types define the common properties of instances. It is possible to specify sets of agent instances of the same type. This concept offers the same scalability advantages as *specification replications* in FOCUS does. Figure 4.5 gives an example of how a set of 100 block agent instances of type AccessPoint may be specified in SDL.

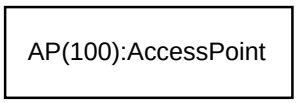

Figure 4.5: Example of a specification of a set of instances.

Subtypes

In general, a type in SDL can be defined as a *subtype* of another type (the *supertype*) and thereby inherit the properties specified for the supertype. This concept may increase scalability of a specification because it enables specification of sets of instances that are of different subtypes, but share a common supertype. For example, if the type AccessPoint in figure 4.5 was a supertype, then the 100 specified instances may not necessarily be of the same type, i.e. they can have different subtypes. Hence the concept of subtypes increases the flexibility of the use

of agent sets.

Sets of Channels Instances

A channel connected to an agent set will actually represent a set of channel instances.

An SDL specification of a set of channels is illustrated in figure 4.6. The line labelled C represents a set of 100 channel instances connecting each instance of type AccessPoint to the instance of type CentralUnit.

The concept of sets of channel instances offers much of the same scalability advantages as the concept of sheafs of channels in FOCUS.

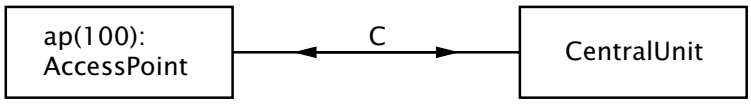

Figure 4.6: Specification of a set of channel instances.

Parameterised Agents

Agents may be parameterised. Parameters may be used in the the internal structure of agents, thus different instances of the same type may have different internal structures. This increases the scalability in a specification, because one does not necessarily have to define a new type for every agent that have a different internal structure as would be the case if parameterised agents were not in the language.

Note that all instances contained in an agent set must take the same parameter.

4.2.2 Generality

Point to Point Topologies

The general case of a star topology of processes and channels is possible to specify. This is because a set of process instances can be specified without a fixed cardinality as is illustrated in figure 4.7. The specification of a process set labelled N(0,):EndNode means that any given number of process instances of type EndNode may be created during computation. The arrow represents a set of channel instances that form a star relationship between the process instances of type EndNode and

the process instance of type StarCentre.

The general case of a tree topology for a fixed depth greater than two is not possible to specify in SDL.

The general case of a *ring* topology can not be specified in SDL because the languages lacks concepts this kind of specification.

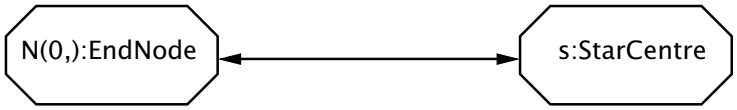

Figure 4.7: Example of an SDL specification of the star topology.

Multipoint Topologies

The general case of a *bus* or a ring topology for multipoint networks can not be specified in SDL using the concepts of agents and channels. Nor does SDL specifications containing a fixed number of instances scale better for these topologies than the corresponding TDL specifications of the same topologies.

4.2.3 Dynamic Reconfiguration

SDL has one concept for expressing dynamic configuration in a specification (remember that we only consider the structural concepts of SDL, i.e. the concepts that may be seen as an extension of TDL).

Process agents can be created dynamically during computation. Process sets in SDL are assumed to be dynamic in the sense that the number of instances contained in these sets may change over time. Omitting to specify the number of instances a process set may contain implies that initially there is one instance, and that the maximum number of instances is unbounded. One may also specify the cardinality of a process set explicitly. Such process sets define an *initial* number of process instances that are created and a *maximum* number of process instances that may be created. Thus one may specify an initial structure of a system *and* constrain the potential number of configurations that the system may exhibit at *a later point in time*.

SDL does not have special concepts for specifying mobile code networks.

In the following we suggest how the concept for specifying dynamic reconfiguration that is included in SDL can be used to specify a contrived example of an ad hoc network. We do not give an example of an object-oriented network, since such a network would be modelled in much the same way as an ad hoc network with the additional constraint that processes may only have one input gate. We do not give an example of a mobile code network either, due to the lack of concepts SDL has for describing such networks.

Example: Ad Hoc Net

Ad Hoc Net consist of three computers *c1, c2* and *c3* that may communicate with each other over a wireless network. That is, each computer has a radio transmitter and a radio receiver and they may be connected if and only if they are within transmission radius of each other. Assume:
(1) That all radio transmitters and receivers use the same radio frequency. (2) A computer is considered to be a part of Ad Hoc Net if it connected to one or more computers. Otherwise it is not part of the network.
(3) *c1* is always connected to *c2* whenever *c1* and *c2* are within transmission radius of each other.
(4) *c1* is always connected to *c3* whenever *c1* and *c3* are within transmission radius of each other.
(5) That *c2* and *c3* may be connected if they are within transmission radius of each other.

The suggested SDL specification of Ad Hoc Net is illustrated in figure 4.8. Here, three process sets are specified. Initially there no computers are part of the network, but each the process sets may contain zero or one process instance as defined by the label "(0,1)" at some later point in time. Thus the specification contains information of how the network may be reconfigured by constraining the number of potential configurations the network may exhibit over time. SDL has no concept for specifying that `c2` and `c3` may not always be connected when both instances are part of the network.

4.3 UML 2.0

The Unified Modelling Language (UML) is specified by the Object Management Group (OMG). UML is broadly used in the modelling of all kinds of systems and there are a large number of tools that support this language [16].

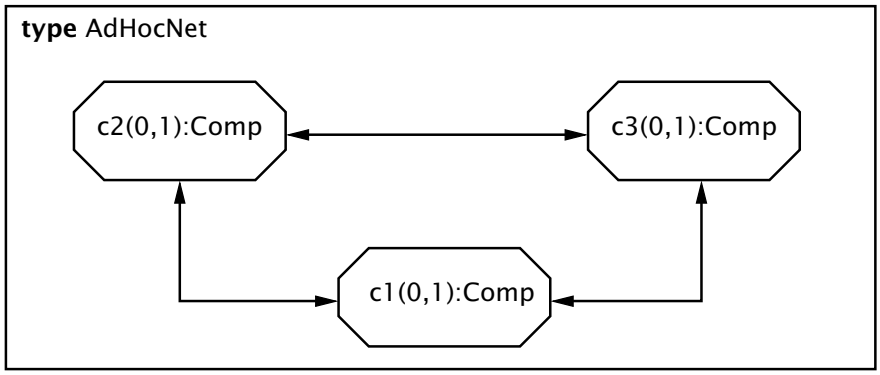

Figure 4.8: Suggested specification of Ad Hoc Net.

Our evaluation is based on UML 2.0. Since the final specification of UML 2.0 is not complete at the time of writing, the basis of our examination of UML 2.0 is the draft specification [19].

We limit our examination of UML 2.0 to the parts that are relevant for this thesis, specifically this means that we examine *composite structures* of UML 2.0 which defines the following concepts:

- *Structured classes* that support the representation of classes which are both encapsulated by ports and have an internal structure.
- *Parts* that are specified within the internal structure of a structured class, and represents sets of instances.
- *Connectors* which represent a set of links over which instances in an internal structure may communicate.

An example of a UML 2.0 specification is given in figure 4.9. Here the structured class AccessPoint is specified as having three parts that represent instances of type Door, Panel and Controller. The lines that connect the parts represent connectors. The small boxes on the boundaries of the structured class represent ports.

4.3.1 Scalability

The following scalability concepts were found in UML:

Hierarchy

Classes may have an internal structure that contain parts and connectors. These parts may represent instances of types that also have an internal structure, thus constituting a hierarchy of parts.

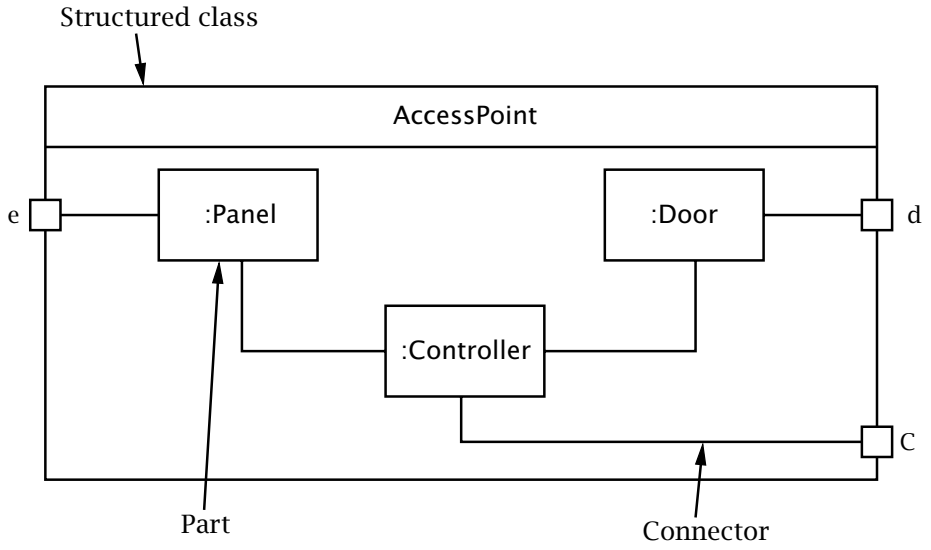

Figure 4.9: Example of a UML 2.0 specification.

Classes, Instances and Parts

UML distinguishes between classes and objects of classes. Classes define the common features of a set of objects/instances.

Instances of the same type can be represented by *parts*. This concept is similar to agent sets in SDL or specification replications in FOCUS, and offers the same scalability advantages.

The cardinality of a part, i.e. the number of instances contained in a part, can be defined using a so-called *multiplicity*. The graphical representation of a multiplicity on parts is a number written in the top right corner of a part, or in brackets next to the label of a part. See for example figure 4.16 where the number n defines the cardinality of the part labelled b:.

Generalisation

UML distinguishes between subclasses and superclasses. A subclass may inherit and specialise the properties defined in a superclass. Several different subclasses can share the same superclass. Thus a part that specifies instances of a supertype can represent instances of several subtypes. This concept offers the same scalability advantages as the concept of subtypes in SDL.

Connectors

A connector may represent a set of links. For example, a connector connecting two parts that each consists of a number of instances, represents a link between every instances in the two parts. This is illustrated in figure 4.10, where (i) illustrates a connector between two parts each containing two instances as indicated by the number in the top right corner of the parts. (ii) illustrates the same specification, only this time the parts each contain only one instance. As can be seen from the illustrations, the top specification actually represents a set of four links.

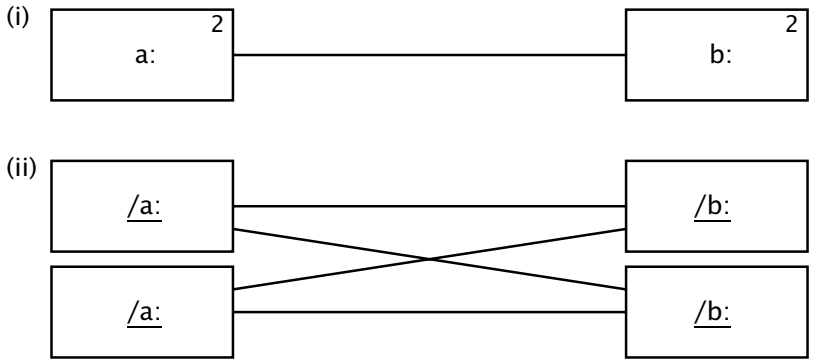

Figure 4.10: Connectors.

Multiplicity on Connector Ends

Multiplicities may be associated with connector ends. Such multiplicities serve to restrict the number of initial links created. Links will be created for each instance playing the connected roles until the connector end multiplicity is reached for both ends of the connector. This concept increases the flexibility of connectors.

Figure 4.11 illustrates the concept. Here the specification in (i), where multiplicities on the connector ends is used, results in the specification in (ii).

Multiplicity on Ports

Multiplicities on ports may define how many links that will be created from connectors that attach to these ports. This concept is similar to arrays of channels in FOCUS.

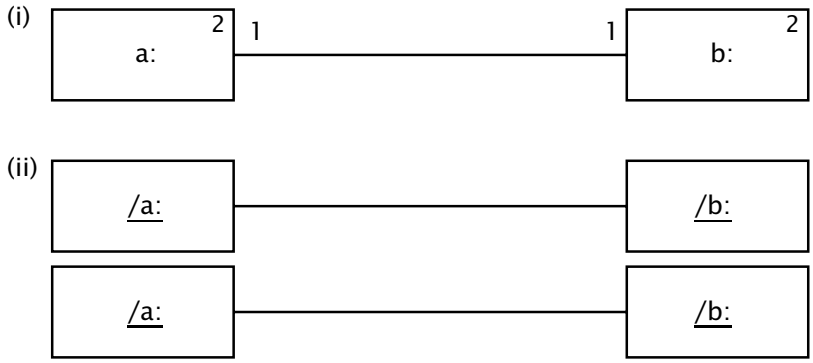

Figure 4.11: Multiplicities on connector ends.

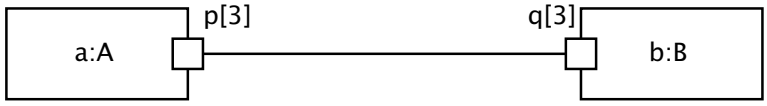

Figure 4.12: Multiplicities on ports.

An example is given in figure 4.12, where two ports, each associated with a multiplicity, are specified. The connector represents a many to many relationship of links. This means that nine links, each connecting part a and b, will be created upon instantiation.

Templates

The concept of templates is similar to the concept of parameterised components in FOCUS and parameterised agents in SDL. Figure 4.13 gives an example of a class with the template parameter CarEngine of class Engine and a so-called value template parameter n of type Integer.

Figure 4.14 shows a part of type Engine with actual parameters. Note that both instances in the part take the same actual parameters.

The concept of templates offers much of the same scalability advantages as the concept of parameterised agents in SDL.

4.3.2 Generality

Point to Point Topologies

The *star* topology may be specified in UML in a manner similar to the SDL specification of this topology.

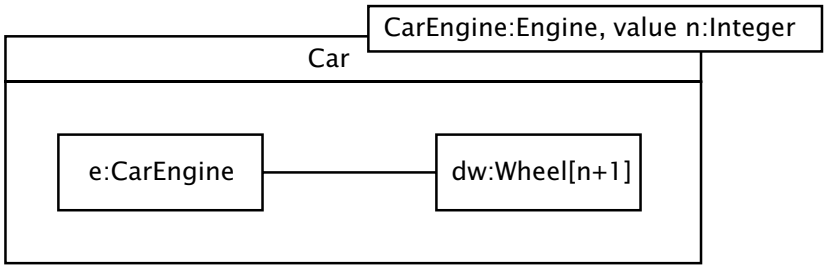

Figure 4.13: Example of template parameters.

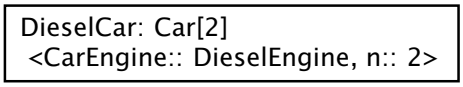

Figure 4.14: Example of actual parameters.

The general case of a *non uniform tree* topology can not be specified in UML. However, the uniform *tree* topology can be specified for a fixed depth. This is illustrated figure 4.15. Here the instance contained in the part named a is connected to every instance in the part named b, which each in turn are connected to i instances in c. Note that a uniform tree topology of depth three is specified if m = i * n.

The general case of a ring topology can not be specified in UML. It can however be underspecified as figure 4.16 illustrates. Here an unspecified (indicated by the * in the top right corner of the part) number of instances are represented by a part. The number 1 on the connector specifies that there will be one link created for each instance in the part. This gives rise to an infinite number of topologies, but one of these topologies will be the ring topology.

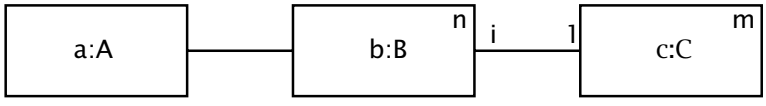

Figure 4.15: Specification of a tree topology of depth three.

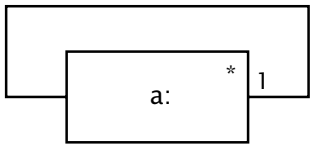

Figure 4.16: Underspecification of a ring topology.

Multipoint Topologies

The general case of a bus or a ring topology for multipoint networks can not be specified in UML. They may however be underspecified. We should also mention that there are ways in UML to refine an underspecification of a bus topology (for example) to a precise specification of a bus topology by specifying exactly how instances should be instantiated in so-called class constructors. But this may not be done in the specification *diagram*. Moreover, the latest UML 2.0 draft specification [19] available at the time of writing is unclear on how exactly this may be done.

4.3.3 Dynamic Reconfiguration

We found no concepts in UML for specifying dynamic reconfiguration.

A multiplicity can actually define a lower and an upper cardinality, so one might think that this concept can be used for expressing dynamic configuration, but this is not so because a multiplicity in UML only indicates an *initial* number of instances that may be created. For example, a part p associated with the multiplicity [0..3] (0 is the lower bound and 3 is the upper bound), will initially contain 0, 1, 2 or 3 instances. Thus the multiplicity does not express anything about how many instances the part may contain at some *later point in time*.

UML does not have any special concepts for specifying that an instance may be transported on a link. Hence, this aspect of a mobile code network cannot be specified in an UML model.

We do not give examples of UML specifications of dynamic networks, due to the lack of concepts UML has for specifying these kinds of networks.

Concept	FOCUS	SDL	UML
Hierarchy	Composite components	Agent sub-structure	Internal class structure
Component sets	Specification replications	Agent sets	Parts
Generalisation		Suptypes	Generalisation
Dependent replications	Dependent replications		
Parametrised components	Parameterised components	Parameterised agents	Templates
Channel sets	Sheafs of channels	Sets of channel instances	Connectors
Channel arrays	Arrays of channels		Multiplicities on ports
Channel constrains			Multiplicities on connector ends

Table 4.1: Classification of scalability concepts

4.4 Language Comparison

In this section we compare the three languages with respect to the problem of scalability, generality and dynamic reconfiguration.

4.4.1 Scalability

Many concepts in the three languages we have examined are similar, although the terms used to describe them are different. These concepts are classified in table 4.1.

As can be seen from the table, all three languages have the concepts of hierarchy, component sets, channel sets and parameterised components. Both UML and SDL as opposed to FOCUS, have generalisation. But the drawback of not having this concept is to some extent made up for in that FOCUS allows different actual parameters to be given to each instance contained in a component set (as was illustrated in figure 4.3). This is not possible in SDL or UML.

Only FOCUS has the concept of dependent replications which offers a scalable solution to the problem of specifying non-uniform relationships. SDL and UML do not have any corresponding concept.

Topology	FOCUS	SDL	UML
Star	Yes	Yes	Yes
Tree (uniform)	Yes*	No	Yes*
Tree (non uniform)	Yes*	No	No
Ring (p-p)	No	No	No**
Ring (mp)	No	No	No**
Bus	No	No	No**

* For a fixed depth.

** Can be underspecified.

Table 4.2: Classification of topology examination results.

Another difference is that only FOCUS and UML have the concept named channel arrays (in lack of a better name). This means for example that n channels between two component instances (agent instances in SDL) cannot be specified in SDL.

Finally, only UML has the concept we have named channel constraints. This concept may increase the flexibility of the use of channel sets.

All in all we can conclude that the scalability concepts in the three languages are very similar, and that none of these languages has concepts that offer a serious scalability advantage over the other languages. We did however find 6 scalability concepts for FOCUS, 5 for SDL and 7 for UML. This suggests that UML handles the problem of scalability slightly better than the other languages.

4.4.2 Generality

In table 4.2, we summarise the results of the examination of the three modelling languages with respect to the problem of generality.

The expressibility of the three languages with respect to specifying the topologies listed in the table are pretty much similar. The exceptions are (1) that only FOCUS and UML can specify uniform the tree topology for a fixed depth greater than two, (2) that only FOCUS can specify the non uniform tree topology for a fixed depth greater than two, and (3) that it is possible to underspecify some of the topologies in UML.

4.4.3 Dynamic Reconfiguration

To give a yes or no answer to the question of whether a language can model a dynamic network would be an oversimplification. It is possible to specify a *snapshot* of the structure of a dynamic network at *one point* in time in all three languages. However, only SDL has a concept for expressing something about the structure a system exhibits at *more than one* point in time.

We should stress (again) that our examination is based on the structural aspects of the languages, i.e. those aspects that may be seen as an extension of TDL. All three languages do in fact have concepts (state machines for example) for specifying behaviour, but these are not considered in the evaluation. Dynamic properties are traditionally not expressed in structural specifications, and this suggests why all three languages have so few “structural” concepts of specifying dynamic reconfiguration.

Chapter 5

MEADOW

5.1 Introduction

The main emphasis of this chapter is to introduce and explain, by natural language and small examples, concepts and features offered by MEADOW. Exactly how these concepts should be implemented is not the main focus of this chapter.

5.2 Components

In MEADOW, a component is an entity that communicates with its environment through its set of referenced ports. We distinguish between a *component type* and a *component instance*. Each component instance is of a specific component type. A component type describes a set of common features shared by all of its instances. Each component instance has its own identity and its own set of properties that conforms to the features defined by the type. Both component types and component instances may be:

- composite or elementary,
- parameterised or unparameterised,
- named or unnamed.

A composite component contains internal components and it may contain channels over which these internal components communicate. An elementary component contains neither internal components nor channels.

A parameterised component type defines a set of *formal parameters* that component instances of this type may take. Parameterised component

instances are component instances with *actual parameters* substituted for formal parameters.

A *named* component is identifiable within a specification, whereas the specification of an *unnamed* component only defines the existence of a component that may not necessarily be identifiable within a specification.

In the following we describe these different component aspects in further detail.

5.2.1 Elementary and Composite Components

As mentioned earlier, components can either be *elementary* or *composite*. A composite component contains (possibly) communicating internal sub-components while an elementary component does not contain internal sub-components. The components and channels that are part of a composite component constitute the *internal structure* of the composite component.

A composite component may communicate with its sub-components, but these channels of communication can not be specified explicitly in MEADOW.

Both composite and elementary components are to be understood as having a behaviour.

Exactly how sub-components within a composite component are created or killed is a semantic variation point, i.e. we do not give rules for how this may be done.

Example

Figure 5.1 presents a specification of one composite component RuntimeEnv, and two elementary components, Program1 and Program2. As shown in the figure, Program1 and Program2 belong to the *internal structure* of RuntimeEnv, thus they are sub-components of RuntimeEnv.

The components represented by RuntimeEnv, Program1 and Program2 are all to be understood as having some kind of behaviour, but (as mentioned previously) MEADOW has no constructs for specifying such behaviour.

Note that the component represented by RuntimeEnv may communicate

with its sub-components, i.e. the components represented by Program1 and Program2

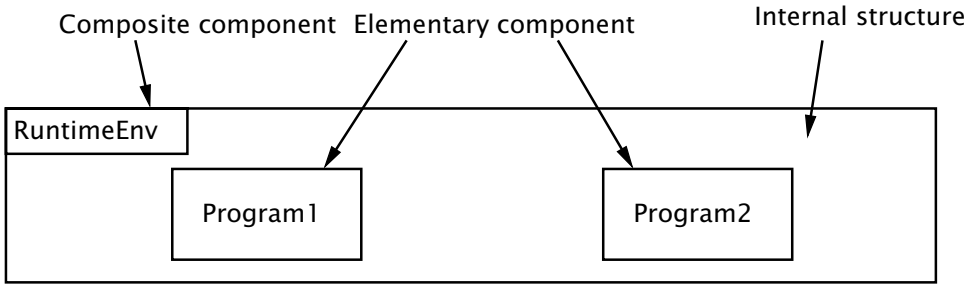

Figure 5.1: Elementary and composite component specification.

5.2.2 Types, Instances and Parts

We distinguish between *types* of components and *instances* of component types (component instances). This is similar to the distinction between the concept of *classes* and *objects* in object-oriented languages. A component type describes a set common features shared by all its instances. A component instance must have a type, and the properties of a component instance must conform to the features defined by its type.

We distinguish between the specification of:

- component types, and
- component instances.

The specification of a composite component type T involves specifying the internal structure that component instances of T have in common. The elements of this internal structure constitute the features of T. All composite component instances of the same type have the same internal structure[1].

Component instances are specified in *parts*. A part is a subset of the set of all instances of a component type. Parts are always unnamed, but the instances in a part may be named. The cardinality of a part may be explicitly defined by a multiplicity. If the multiplicity is not specified, then the part has a cardinality equal to 1.

[1]Unless the composite component is parameterised.

We say that the lifetime of a part is equal to the lifetime of the composite component instance that the part is associated with/contained in.

Example

Figure 5.2 presents a specification of a component type RuntimEnv. The fact that a component *type* is specified is indicated by the colon prefixing the label in the top left corner of the specification.

The internal structure that all component instances of RuntimeEnv have in common, consists of two internal parts labelled Program1 and Program2. The first part contains one unnamed component instance of type Program1, while the other part contains one unnamed component instance of type Program2. These parts constitute the features of the component type RuntimeEnv; all instances of component type RuntimeEnv are each associated with such parts.

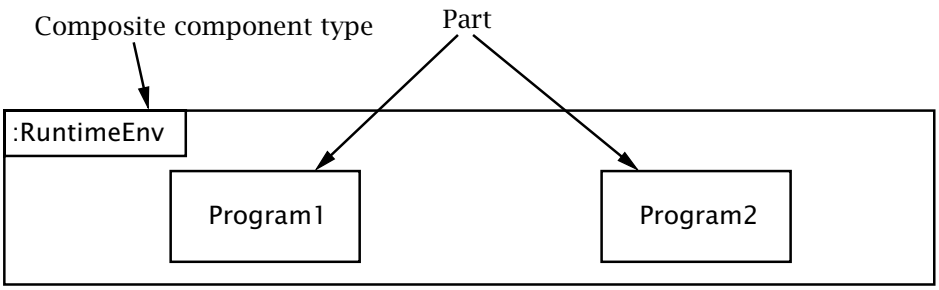

Figure 5.2: Specification of a component type and parts.

5.2.3 Multiplicity on Parts

A *multiplicity* is used in order to specify parts that contain more than one component instance. A multiplicity may consist of (1) a single value that defines the cardinality that a part must have at all times during its lifetime (unless the part is dynamic) or (2) two values l and u that define a *lower bound* and an *upper bound* for the cardinality of a part, respectively. A part (that is not dynamic) that is associated with a multiplicity of the latter kind must have a cardinality equal to exactly one number n, where $l \leq n \leq u$, at all times during its lifetime. Moreover, a part that is not dynamic must always contain the same component instances during its lifetime.

A multiplicity associated with a dynamic part has a slightly different meaning. Dynamic parts are presented in section 5.2.7.

Example

Figure 5.3 presents a specification that contains two parts. The part labelled ProgramA[2] contains two unnamed instances of component type ProgramA. All instances of RuntimeEnv are associated with such parts, i.e. they each contain two unnamed component instances of type ProgramA. The other part in the specification must contain either two, three of four instances of component type ProgramB during its lifetime. In other words, each component instance ci of type RuntimeEnv must contain either one, two or three component instances of type ProgramB at all times during the lifetime of ci.

Note the overlapping effect on the representation of the parts. This effect is always used on parts that may contain more than one component instance.

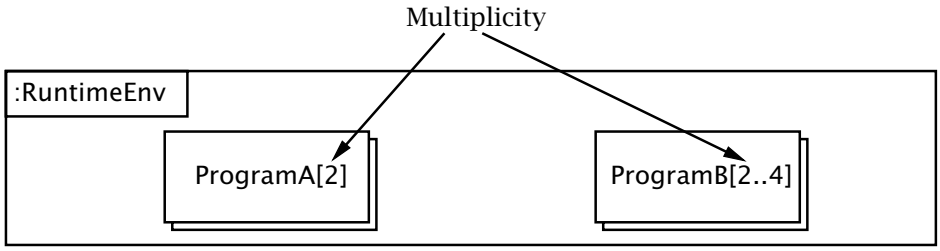

Figure 5.3: Specification of parts with multiplicity.

5.2.4 Parts of Named Component Instances

A part may be given an *identifier* that defines the names of the component instances that are contained in the part. An identifier may either be a:

- constant $i \in T$, or a
- type T.

A part whose identifier is a constant contains one component instance whose name equals that constant. A part P whose identifier is a type T contains component instances such that $\forall e \in T$: exactly one component instance named e is contained in P.

An identifier defines the cardinality of a part; a part associated with an identifier type T contains as many instances as there are elements in T.

If a part is associated with and identifier and a multiplicity, then the multiplicity defines the cardinality of the part. In such cases the upper bound of multiplicity cannot exceed the cardinality of the identifier.

If a part is associated with the an identifier T and a multiplicity that defines an upper and a lower bound such that the upper bound is not specified (denoted [I..]), then the upper bound equal the cardinality of T. (See example 2 below).

Example 1

Figure 5.4 presents a specification of a composite component type RuntimeEnv. Here, the part labelled id:ProgramA consists of one component instance named id of type ProgramA. The other part labelled {id1,id2}:ProgramB consists of two component instances named id1 and id2 of type ProgramB.

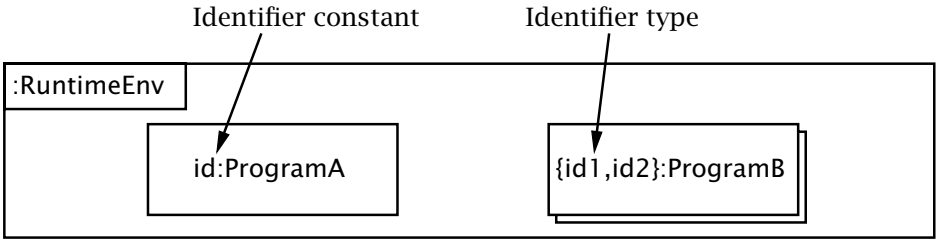

Figure 5.4: Specification of named component instances.

Example 2

Figure 5.5 presents a specification of two parts that are associated with identifiers *and* multiplicities. The part labelled {a,b,c}:ProgramA[1..] may contain either 1, 2 or three component instances during its lifetime. The other part, labelled {id1,id2}:ProgramB[1], may either contain a component instance named id1 or a component instance named id2 during its lifetime.

5.2.5 Declarations and Assignments

In a part or type specification, it is possible to specify:

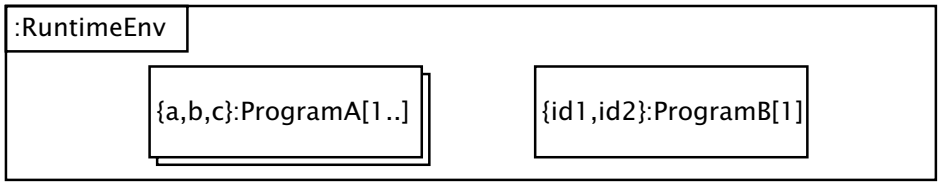

Figure 5.5: Parts with identifiers and multiplicities.

- constant, type and function declarations;
- constant, type and function definitions.

Declared types and constants may be used in the internal structure of the specification of a composite part or a composite type.

The area in which declarations, assignments and function definitions are made in a composite component is called the *header*.

In MEADOW, there are two predefined types, $\mathbb{N}$ and $\mathbb{S}$. $\mathbb{N}$ denotes the set of all natural numbers and $\mathbb{S}$ denotes the set of all strings. Also, $\mathbb{P}$ denotes a power-set. For example, $\mathbb{P}(\mathbb{N})$ denotes the power-set of $\mathbb{N}$, i.e. the set of all subsets of $\mathbb{N}$.

Example

Figure 5.6 presents a specification of the composite component type RuntimeEnv. In the header of the specification we have declared and defined two types, PidA and PidB. These types are used as identifiers for two of the parts in the internal structure. The part labelled PidA:ProgramA contains three component instances named id1, id2 and id3. Similarly, the part labelled PidB:ProgramB contains five component instances named 1, 2, 3, 4 and 5.

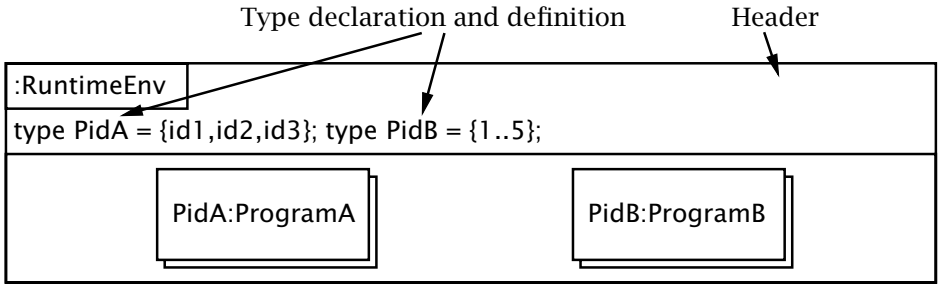

Figure 5.6: Composite component with declarations and definitions.

5.2.6 Parameterised Components

In order to model components that are similar, but differ in some small way, we allow components to be parameterised.

Parameter declarations are of two kinds:

- type T
- constant $p \in T$

The first declares a parameter of type T, the second declares a constant p of type T. We also distinguish between:

- formal parameters and
- actual parameters.

A formal parameter may occur in a component type only. Actual parameters are associated with parts only. An actual parameter is substituted for a formal parameter.

Example 1

A specification of a parameterised composite component type RuntimeEnv is presented in figure 5.7. As indicated in the figure, the specification defines two formal parameters; one type named T, and one constant e of type $\mathbb{S}$. They are used as identifiers for the instances of the two parts defined in the internal structure of RuntimeEnv.

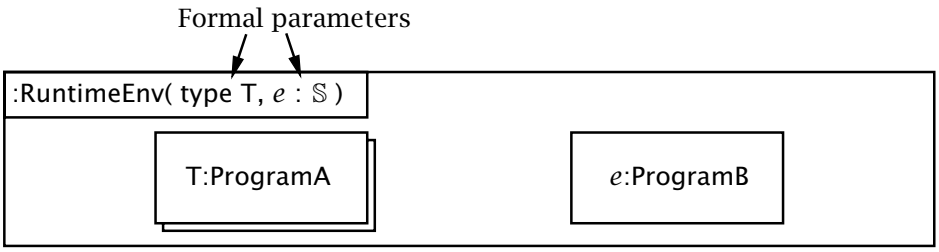

Figure 5.7: Specification of a parameterised composite component type.

Example 2

Figure 5.8 presents a specification of a composite part of type Computer (this is a specification of a part and not a type because there is no ':'-character prefixing the label in the top left corner of the specification). The (sub)part defined in its internal structure consists of one

instance of component type RuntimeEnv for each element in Rid. Each of these instances has the same internal structure as the component type RuntimeEnv (as specified in figure 5.7), except that the formal parameters T and c are substituted by the actual parameters {i1,i2} and i3, respectively.

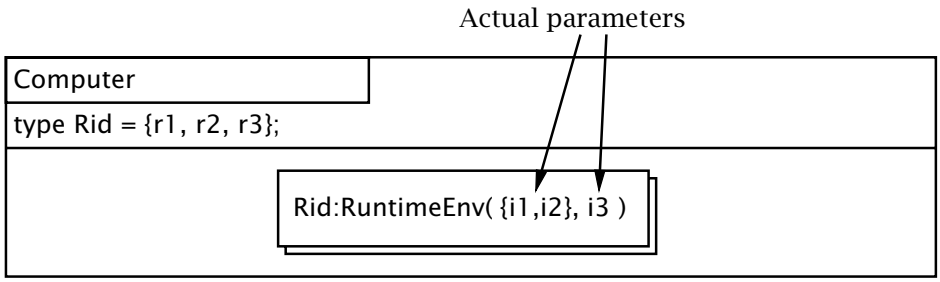

Figure 5.8: Specification containing actual parameters.

Example 3

Another specification of a part of type RuntimeEnv is presented in figure 5.9. In the header we have declared a new function, f, that maps elements of Rid to elements of $\mathbb{S}$ as defined. The part specified in the internal structure of RuntimeEnv consists of three component instances. The specification of this part defines that for each x $\in$ Rid: the relevant formal parameters of the instance named x of type RuntimeEnv are substituted by {i1,i2}, and f(x). For example, this means that the component instance named r1 of type RuntimeEnv contains a component instance of type ProgramB whose name is p1 (assuming the specification in figure 5.7).

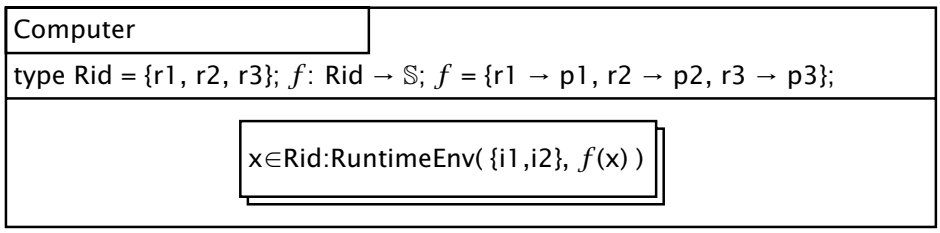

Figure 5.9: Specification of a parameterised part.

5.2.7 Dynamic Parts

A dynamic part is a part that is dynamic in the sense that all component instances that are contained in the part may be created or killed during the lifetime of the composite component instance they are a part of. Specifically, a dynamic part is part whose cardinality may change during its lifetime. This is contrary the parts we have seen so far, that must have a fixed cardinality at all times during their lifetime.

The maximum cardinality of a dynamic part denotes the cardinality value that the part cannot exceed at any given point during its lifetime. Similarly, the minimum cardinality of a dynamic part denotes the cardinality value that must be less than or equal to the actual cardinality a part has at any given time during its lifetime.

If a dynamic part is associated with a multiplicity containing an upper and a lower bound, then the lower bound defines the minimum cardinality of the part, and the upper bound defines the maximum cardinality of the part.

If a dynamic part is associated with a multiplicity consisting of a single value, then the maximum cardinality of that part is equal to this value, and the minimum cardinality of the part is equal to zero.

If a dynamic part is associated with an identifier (and not a multiplicity), then the maximum cardinality of that part is equal to the cardinality of the identifier, and the minimum cardinality that the part is equal to zero.

If a dynamic part is associated with both an identifier and a multiplicity, then the multiplicity defines the cardinality of the part. The upper bound of the multiplicity may not exceed the cardinality of the identifier.

We say that a *static part* is a part that is not dynamic.

Example

Three dynamic parts and one static part are specified in figure 5.10. As indicated, the notation of a dynamic part is a box with dashed outline. The dynamic part of type `B` contains two component instances named `a` and `b` that may or may not be part of the composite component instance of type `A` during the lifetime of this composite component. We say that the cardinality of the part of type `B` may change between 0, 1 or 2 during its lifetime.

The cardinality of the dynamic part of type C may change between zero, one or two during its lifetime. Similarly, the cardinality of the part of type D may change between one and two during its lifetime.

The static part of type E, must have a fixed cardinality of either one or two during its lifetime, i.e. its cardinality cannot change during computation as the cardinality of dynamic parts can. Furthermore, the part must either (1) contain a component instance named a at all times during its lifetime, or (2) contain a component instance named b at all times during its lifetime, or (3) contain both a and b at all times during its lifetime.

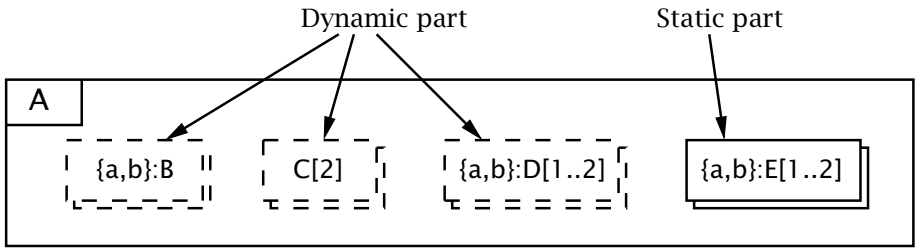

Figure 5.10: Specification of static and dynamic parts.

5.2.8 Generalisation

A generalisation is a relationship between two component types, a supertype and a subtype. The subtype of a supertype inherits all features, i.e. the declarations and definitions and internal structure of the supertype. This is similar to the concept of *inheritance* from object-oriented programming languages.

A subtype A may refine a supertype B. This means that a specification of A may include features (declarations, components and channels) in addition to the features that are inherited from B as long as the interface between A and the environment of A is equal to the interface between B and the environment of B. The interface between a component and its environment is defined by the channels with which the component communicates with its environment.

Example

Figure 5.12 presents a specification of two generalisation relationships. Here the supertype Program has two subtypes ProgramA and ProgramB.

As indicated in the figure, a line with a white arrowhead represents a generalisation relationship.

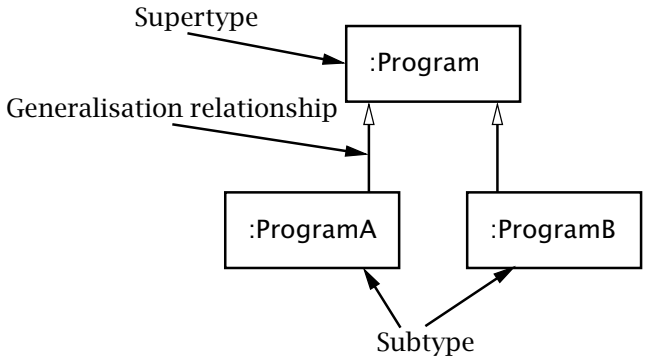

Figure 5.11: Generalisation relationships.

5.2.9 Regions

A region is a special case of a composite component. The difference between the concept of a region and the concept of a composite component is that regions cannot have a behaviour, while composite components may have a behaviour. Also, a region can not communicate with its sub-components.

Regions are specified exactly like composite components with two exceptions: (1) the notation for a region differs from the notation for a composite component and (2) a region may not be associated with input or output channels that are not connected to its sub-components or sub-regions within its internal structure. The latter constraint is due to the fact that a region cannot output messages or process incoming messages since it does not have a behaviour.

Example

Assume we want to model a network consisting of two computers. It would be inappropriate to model the network itself as a composite component, since a composite component is assumed to have a behaviour of its own. Hence we use a region instead. Figure 5.12 presents a specification of a region of type Net. The internal structure of this region consists of two component instances of type Computer. As can be seen from the figure, the notation for a region is a box with rounded edges.

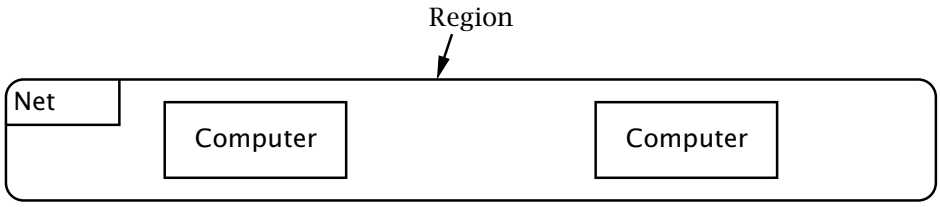

Figure 5.12: Specification of a region.

5.3 Connectors and Channels

Channels are instances of *connectors*. Channels connect *component instances* while connectors connect *parts*. In the following we describe the different kinds of channels and connectors that we use in MEADOW.

5.3.1 Classification of Channels

Every channel has exactly one input-port and one output-port. A channel represents the forwarding of messages from its output-port to its input-port and may be:

- messaged typed,
- named and
- shared.

A *channel* c connecting two component instances C_1 and C_2 means that C_1 can forward messages on c (i.e. C_1 has a reference to the output-port of c) that are received by C_2 (i.e C_2 has a reference to the input-port of c) or vice versa.

A *shared channel* is a channel that has a port that is referenced by more than one component instance.

A *messaged typed* channel is a channel that may only forward messages that are elements/instances of a specific type.

A *named* channel is uniquely identifiable within a specification, while an unnamed channel may not necessarily be identifiable in a specification.

5.3.2 Classification of Connectors

A *connector* is a set of channels that connect component instances in parts. The exact nature of the relationship of channels a connector con-

tains depends on the cardinalities of the parts the connector connects. We distinguish between three main types of relationships:

- one to one,
- one to many, and
- many to many.

A connector may also have a set of properties that may effect the relationship of its channels. A connector may be:

- named or unnamed,
- directed or bi-directed,
- split or merged,
- function constrained,
- cardinality constrained,
- identifier constrained.

A connector may also have a *multiplicity*, an *identifier type* and a *message type*.

In the following we describe these kinds of connectors in more detail.

5.3.3 Deriving Channels from Connectors

Connectors connect parts, and channels connect component instances. This is illustrated in figure 5.13.

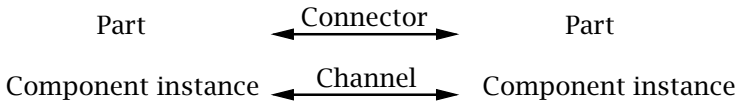

Figure 5.13: Connectors and channels.

Channels are never specified explicitly; they are always derived from connectors in a specification. Moreover, the names of channels may be derived from the labels that are associated with connectors.

A connector may be associated with a multiplicity. Intuitively the multiplicity multiplies the relationship of channels that a connector would

consist of without the multiplicity. For example, a connector named c that is associated with a multiplicity of n (we denote this c[n]) contains exactly n relationships of channels.

A multiplicity that is associated with a connector may only contain a single value.

A connector cannot be associated with both a multiplicity and an identifier.
A connector may also be associated with an identifier. In general, a connector named c that is associated with an identifier type T (we denote this T:c) means that c contains exactly one channel for each element e in T whose name is e. For example, a connector {i,j}:c means that a connector named c consists of one channel named i and one channel named j.

Throughout the rest of section 5.3, "a connector c" denotes a connector named c that is *not* associated with a multiplicity or an identifier type.

5.3.4 Directed Connectors

A *directed connector* consists of channels going in one direction only. As mentioned previously, the relationship of channels a connector consists of depends on the cardinality of the parts the connector connects. We distinguish between three main kinds of relationships of channels that a connector can consist of: *one to one*, *one to many* and *many to many*. In the following we describe these relationships in more detail.

Henceforth, "the cardinality of a relationship" denotes the number of channels that constitute a relationship.

One to One

A directed connector c that connects two parts, each of cardinality equal to one, means that c contains one channel connecting the component instances in the two parts. We say that c consists of a *one to one relationship*. This relationship is illustrated in figure 5.14.

Example

Figure 5.15 presents a specification of a region of type LNet which contains two parts labelled ole:Male and marit:Female. The arrows between the two parts represent directed connectors. The connector i consists of one unnamed channel. In other words i contains a single one to one

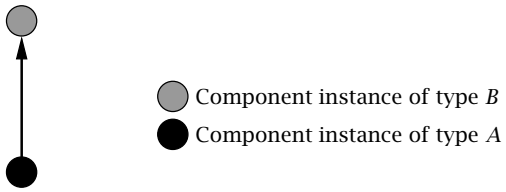

Figure 5.14: Illustration of a directed one to one relationship.

relationship. The connector j[2] contains *two* one to one relationships, i.e. two channels. The connector {a,b}:k consists of two channels named a and b.

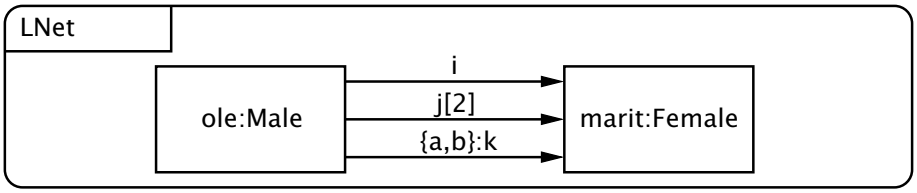

Figure 5.15: One to one relationship.

One to Many

A directed connector connecting a part P_1 of cardinality one and a part P_2 of cardinality greater than one, means that there is a one to one relationship between each component instance in P_2 and the component instance in P_1. This relationship, called a *one to many* relationship, is illustrated in figure 5.16.

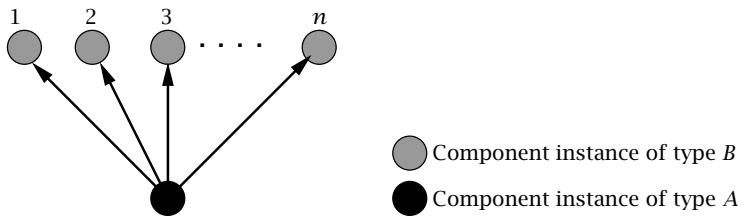

Figure 5.16: Illustration of a directed one to many relationship.

Let a connector c consist of a one to many relationship of cardinality n. If an identifier is associated with c, then the cardinality of this identifier

must be equal to n.

Example

The connectors in the lowermost specification in figure 5.17 consist of one to many relationships of channels. Specifically, the connector c contains one unnamed channel going from ole to marit and another unnamed channel going from ole to guro. The connector named l contains (1) one channel named a from ole to marit and one channel named b from ole to guro or (2) one channel named b from ole to marit and one channel named a from ole to guro. In this respect the lowermost specification in figure 5.17 is ambiguous since two different configurations can be derived from it.

The topmost specification in figure 5.17 presents an alternative specification of LNet for the situation where channel a connects ole and marit and channel b connects ole and guro.

Note that the cardinality of the identifier associated with the connector l is equal to the cardinality of the one to many relationship that l would have consisted of had it not been associated with an identifier. This is a requirement (as stated previously). Note also that if the connector c had been associated with a multiplicity of two (c:[2]), then c would have contained two one to many relationships, i.e. four unnamed channels in this case.

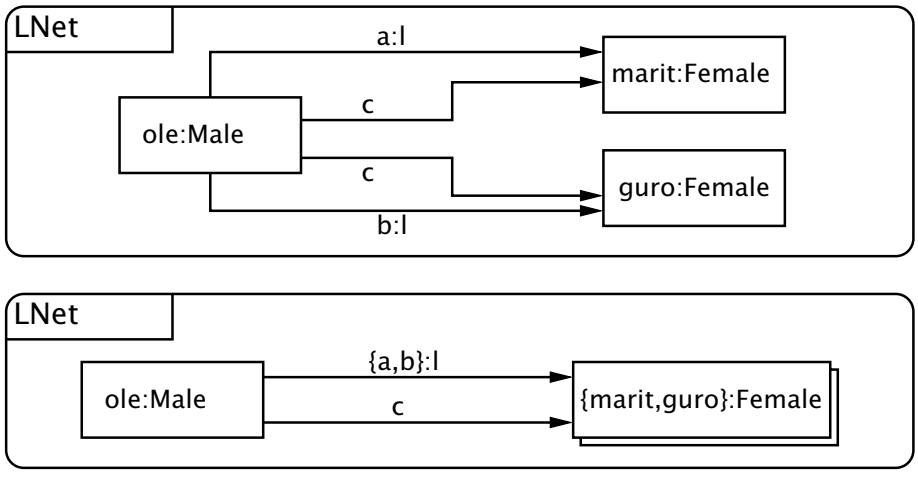

Figure 5.17: Specification of a directed one to many relationship.

Many to Many

A directed connector c going from a part P_1 to a part P_2, each of cardinality greater than one, means there is a one to many relationship between each component instance in P_1 and the component instances in P_2. We say that c consists of a *many to many* relationship. This relationship is illustrated in figure 5.18.

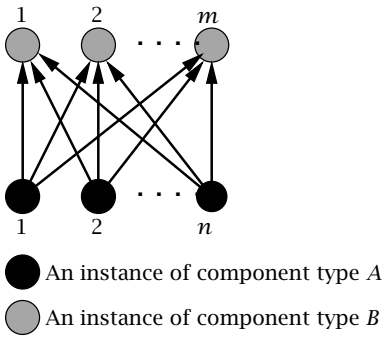

Figure 5.18: Illustration of a many to many relationship.

Let a connector c consist of a many to many relationship of cardinality n. If an identifier is associated with c, then its cardinality must be equal to n.

Example

The lowermost specification in figure 5.19 contains a connector named l that connects two parts of cardinalities greater than one. Hence l contains a many to many relationship of channels. Specifically, two channels connect ole to ine and pia and two channels connect kim to ine and pia. The names of these channels are a, b, c and d.

The topmost specification of figure 5.19 presents an alternative specification of LNet for the case where a connects ole and ine, b connects ole and pia et cetera.

5.3.5 Bi-directed Connectors

A *bi-directed connector* consists of channels going in two directions. Intuitively, we can think of a bi-directed connector as two directed connectors containing channels that forward messages in opposite directions. An identifier cannot be associated with a bi-directed connector.

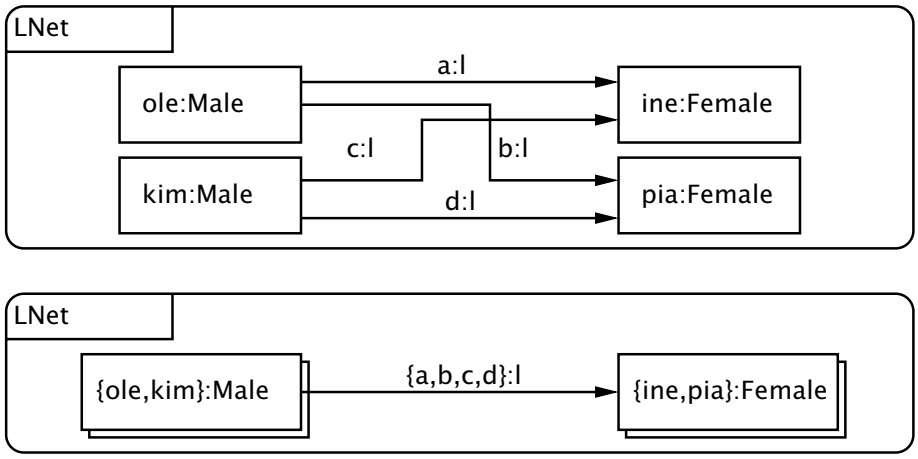

Figure 5.19: Directed many to many relationship.

Example

Figure 5.20 presents a specification containing one bi-directed connector named c that connects the two parts labelled kim:Male and {ine,pia}: Female. This bi-directed connector consists of two channels going from kim to ine and pia and two channels going from ine and pia to kim. In other words, c contains one many to one relationship and one one to many relationship. Note that the notation for a bi-directed connector is a line with two arrow heads.

If the bi-directed connector c had been associated with a multiplicity of two (denoted c[2]) for example, then c would have consisted of two bi-directed one to many relationships, i.e. eight channels in this case.

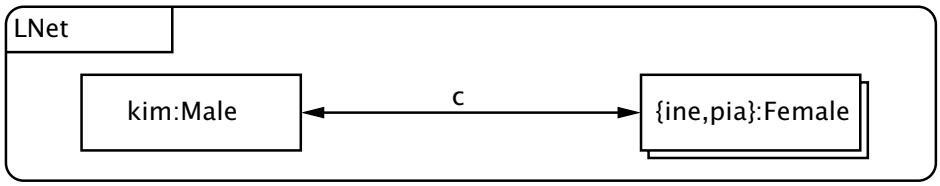

Figure 5.20: Specification of a bi-directed connector.

5.3.6 Connectors Between Parts and Environments

Thus far, we have only seen connectors between parts. It is however, also possible to specify a connector going from a part P to the environ-

ment of the containing composite component or region that P is part of.

In general, a directed connector connecting a part P and the environment of the composite component or region that P is part of contains a one to one relationship between every component instance in P and the environment. A bi-directed connector connecting P and the environment, consists of two one to one relationships (one in each direction) between every component instance in P and the environment.

Example

The specification in figure 5.21 contains three connectors named k, l and o that connect the environment of region type LNet to its internal parts. Connector k contains two channels; one channel going from kim to the environment and another going from the environment to kim. The connector l[2] contains two channels going from the environment to kim. The connector named o contains three channels, each connecting an instance of component type Female to the environment.

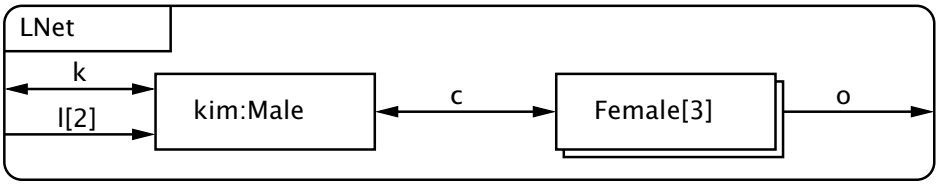

Figure 5.21: Connectors between internal parts and environment.

5.3.7 Message Typed Connectors

A connector c that has a message type M means that all messages forwarded on the channels that c consists of must be elements/instances of type M.

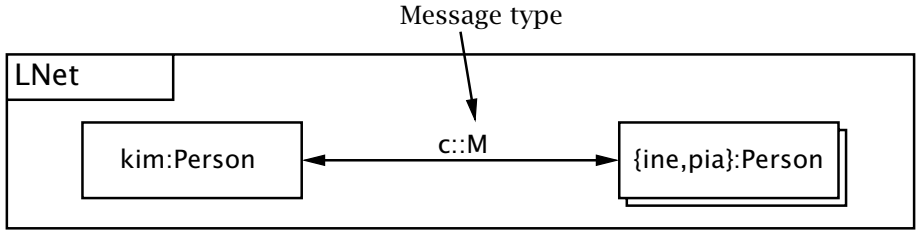

Figure 5.22: Specification of a message typed connector.

Example

Figure 5.22 presents a specification that contains a connector named c of message type M. This means that all four channels contained in c may forward messages that are elements of type M.

5.3.8 Component Types as Message Types

A connector may be associated with a message type which contains component instances. Channels derived from such a connector may forward instances of component types.

Example

Figure 5.23 presents a specification that contains the declaration of a type T. This type is defined as the component type B. The notation :B denotes that B is component typed. T is associated with the connectors in the internal structure of the specification. Hence the channels that the connectors consist of may forward instances of component type B.

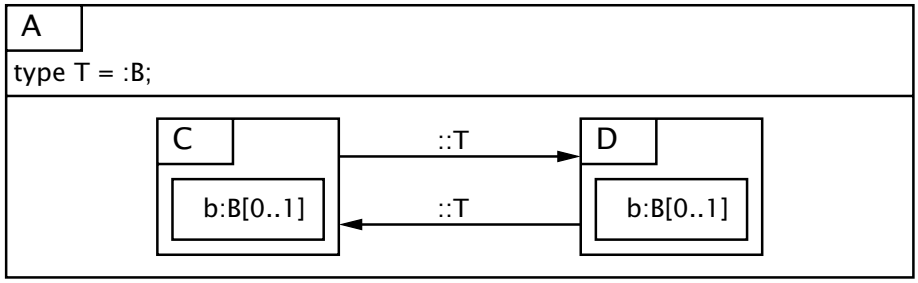

Figure 5.23: Specification of component type as message type.

5.3.9 Split Nodes

Figure 5.24 illustrates a channel that is shared in the sense that it may forward messages from exactly one component instance to more than than one component instances. We that the channel constitutes a split one to many relationship, and we use *split nodes* to model such relationships. We do not put any constraints on how a split node should be implemented. Hence messages that are output on the channel constituting a split one to many relationship may or may not be received by all component instances that reference the input-port of this channel.

A directed connector c consists of a split many to many relationship if

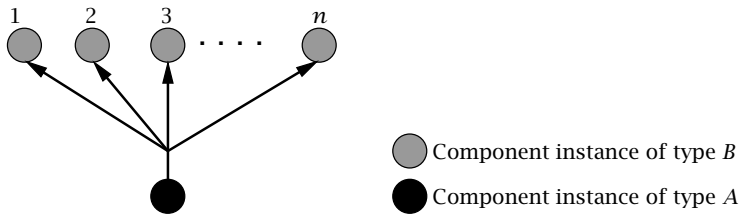

Figure 5.24: Illustration of a split relationship

(1) it is associated with a split node and (2) it connects a part P_1 (of cardinality greater than one) to a part P_2 (of cardinality greater than one). More precisely, c consists of channels such that there is a split one to many relationship between each component instance in P_1 and the component instances in P_2. If the cardinality of P_1 had been equal to one, then c would have consisted of one split one to many relationship. If c is bi-directed, then it consists of two split relationships going in opposite directions.

It is allowed to associated a split node with a connector d that connects two parts of cardinality equal to one, but d would then have contained exactly the same channel had it not been associated with a split node.

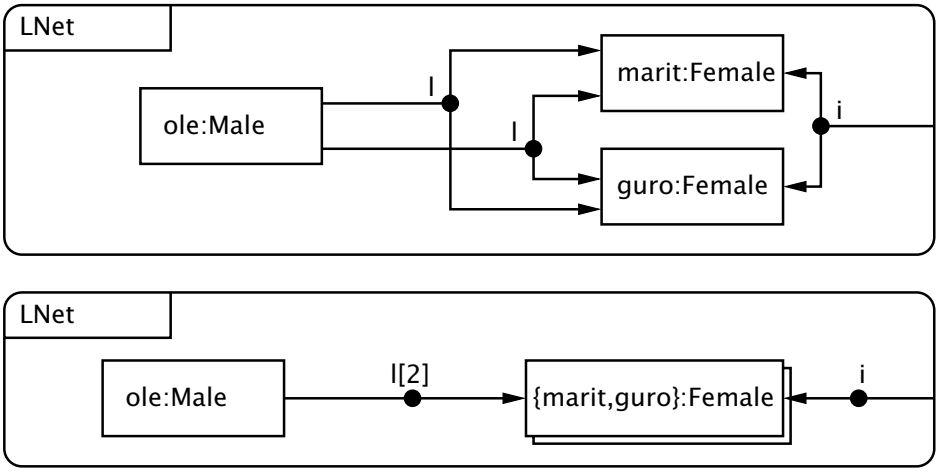

Figure 5.25: Split node associated with connectors.

Example

Figure 5.25 presents two alternative specifications of a region of type LNet. The same component instance names and channel names can be

derived from both specifications, so in that respect they are equivalent.

The lowermost specification contains two directed connectors that are associated with one split node each (as specified by the black circles). The connector l[2] consists of two split one to many relationships, i.e. in this case it contains two unnamed channels that ole may forward messages on and marit and guro may receive messages from. The other connector named i consists of one channel that forwards the messages received from the environment to marit and guro.

5.3.10 Merge Nodes

A channel that is shared in the sense that messages are forwarded from several component instances to exactly one component instance constitutes a *merged many to one* relationship. This is illustrated in figure 5.26. We do not put any constraints on how a merge node should be implemented. Hence, how messages are merged in a merged many to one relationship is always underspecified.

A directed connector c consists of a merged many to many relationship if (1) it is associated with a merge node and (2) it goes from a part P_1 (of cardinality greater than one) to a part P_2 (of cardinality greater than one). More precisely, c consists of channels such that for each component instance ci in P_2 there is a merged many to one relationship between P_1 and ci. If the cardinality of P_2 had been equal to one, then c would have contained one merged many to one relationship. If c had been bidirected, then it would have consisted of two merged relationships going in opposite directions.

It is allowed to associated a merge node with a connector d that connects two parts of cardinality equal to one, but d would then have contained exactly the same channel had it not been associated with the merge node.

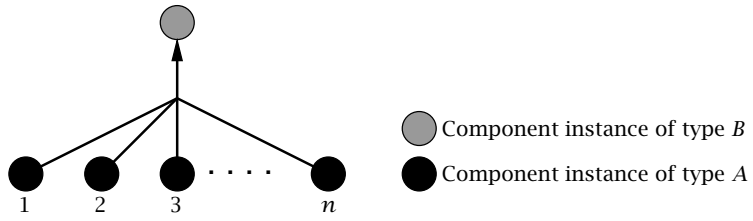

Figure 5.26: Illustration of a merge relationship.

Example

Two alternative specifications of a region of type LNet is presented in figure 5.27. The specifications are equivalent in the sense that the same component instances and channels can be derived from them.

The upper specification contains four connectors that are associated with merge nodes (represented by the white circles). The topmost connector consists of a channel that (1) ole and kim may forward messages on and (2) that ine may receive messages from. Similarly, the lowermost connector for example, consists of a channel going from ole and kim to pia.

If the bi-directed connector c in the lowermost specification had been associated with a multiplicity of two (c[2]), then c would have consisted of four (not two since the connector is bi-directed) merge many to many relationships.

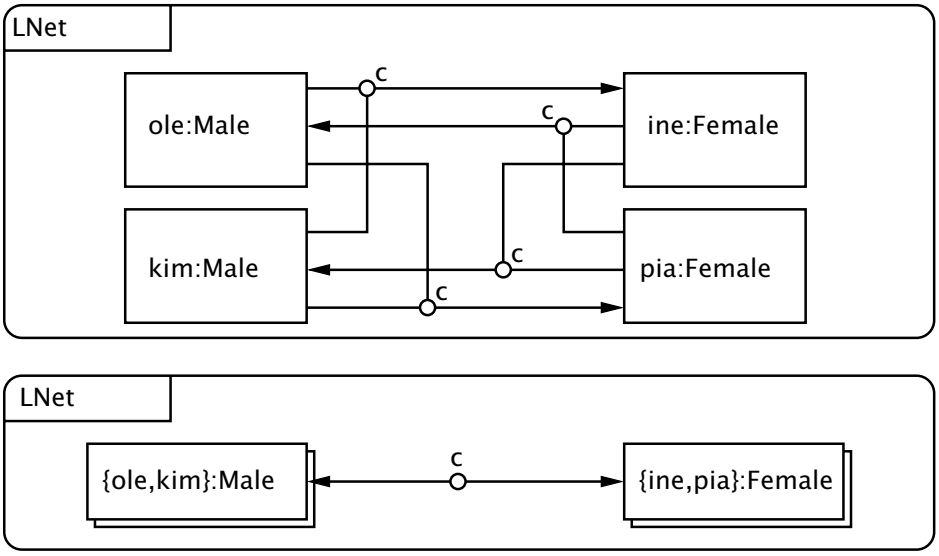

Figure 5.27: Specification with merge nodes.

5.3.11 Merge-Split Nodes

A channel that (1) one or more component instance may forward messages on and (2) one or more component instance may receive messages from, is illustrated in figure 5.28. We call this a merge-split relationship. We use a *merge-split node* to model such relationships.

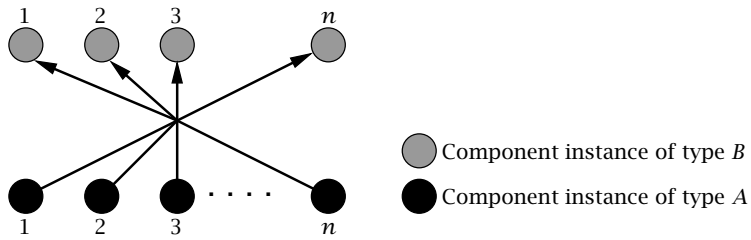

Figure 5.28: Illustration of a merge-split relationship.

Let c be a directed connector. A merge-split node associated with c means that c contains a merge-split relationship. In other words, c contains a channel that is merged and split.

A merge-split node associated with a bi-directed connector c contains two merge-split relationship going in opposite directions.

Example

The two specifications is figure 5.29 are equivalent in the sense that the same component instances and channels can be derived from them. The lowermost specification contains one connector associated with a merge-split node (represented by the grey circle). This connector contains two channels that (1) ole and kim may forward messages on and (2) ine and pia may receive messages from.

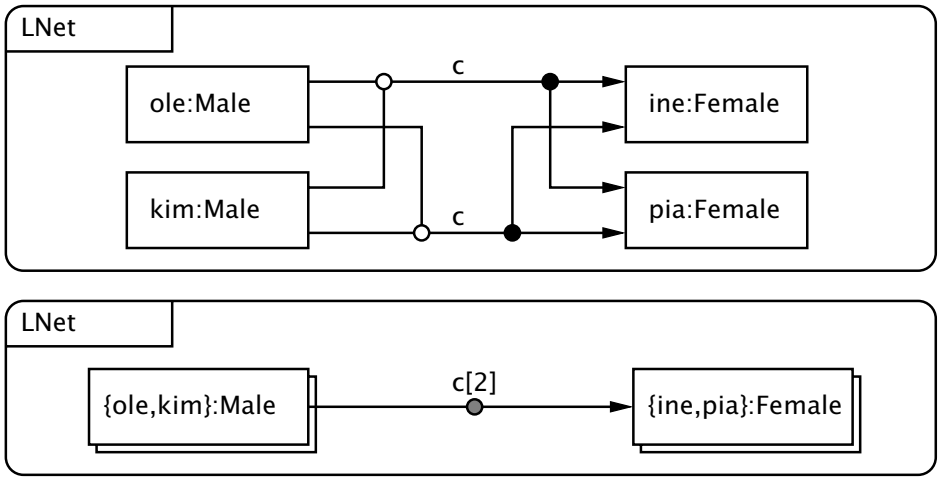

Figure 5.29: Specification containing a merge-split node.

5.3.12 Identifier Constraints

Identifiers constraints can be associated with connectors. Such identifiers constrain the number of channels that a connector contains.

Let Id and I be two identifier types where $I \subseteq Id$. Let P_1 be a part of component type CT_1 and P_2 be a part of type CT_2 that consists of exactly one component instance for each element e in Id whose name equals e.

Let c be a connector that connects P_1 and P_2. If I is associated with the endpoint of the representation of c that is attached to P_2, then c contains a relationship of channels between P_1 and every component instance in P_2 whose name equals an element in I. The exact nature of the relationship depends on the cardinalities of P_2 and I, and whether c is associated with a multiplicity.

A connector that is associated with an identifier constraint, can not be associated with a (channel) identifier.

Example

As indicated in figure 5.30, an identifier constraint is placed on the endpoint of the representation of a connector. Without the identifier constraint, the connector c would contain two channels connecting kim and ine and two channels connecting kim and pia. With the constraint however, c only connects kim and ine, thus consisting of two channels only. Similarly, without the identifier constraint, connector d[4] would contain four one to many relationships between kim and {ine,pia}, i.e. eight channels in this case. With the constraint however, d only consists of the sub-set of channels that connect kim and pia, i.e. four channels.

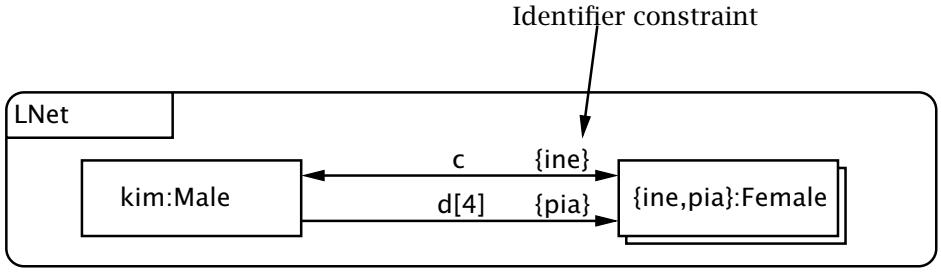

Figure 5.30: Specification containing identifier constraints.

5.3.13 Cardinality Constraints

Thus far we have only seen multiplicities that are associated with connector labels. However, multiplicities can also be associated with the endpoint of a connector.

Let c be a directed connector going from a part P_1 to a part P_2. A cardinality constraint, $[n]$, associated with c and P_1 means that c consists of channels such that each component instance in P_1 may output on n channels.

Let c be a bi-directed connector going from a part P_1 to a part P_2. A cardinality constraint, $[n]$, associated with c and P_1 means that c consists of channels such that each component instance in P_1 may output on n channels and input on n channels.

A connector c that is associated with a cardinality constraint may also be associated with a multiplicity (on the channel). The multiplicity will then multiply the relationship of channels that c consists of without the multiplicity.

A connector that is associated with a cardinality constraint can not be associated with a channel identifier.

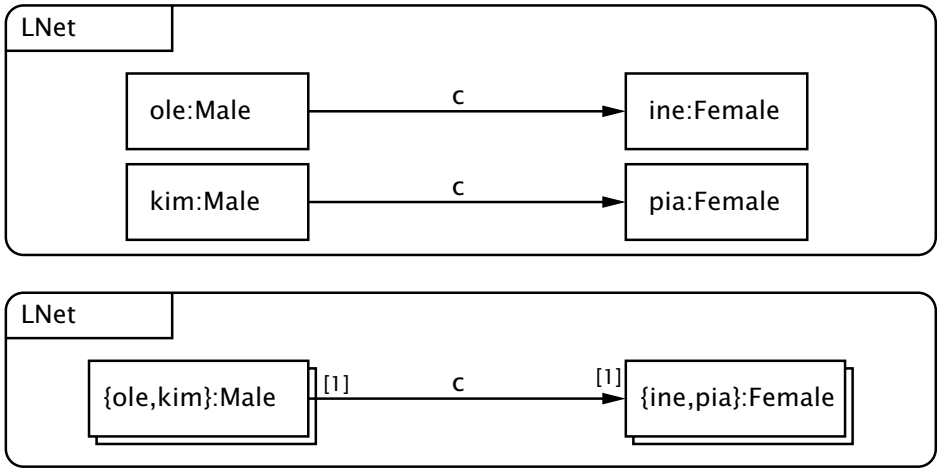

Figure 5.31: Specification containing cardinality constraints.

Example

The connector in the lowermost specification in figure 5.31 is associated with the two cardinality constraints ([1]). Hence we derive that ole and kim may output on one channel each and that ine and pia may input on one channel each. There are two possible such configurations: (1) either ole is connected to ine and kim is connected to pia or (2) ole is connected to pia and kim is connected to ine. The former configuration is specified in the topmost specification in figure 5.31.

If c had been associated with a multiplicity of two (c[2]), then c would have consisted of four channels.

5.3.14 Constraint Functions

A constraint function maps single identifiers to (1) (possible empty) subsets of identifiers. The function is declared and defined in the header of the specification of a composite part, a composite component type or a region, and it may be associated with a connector.

Let Id_1 and Id_2 be identifier types, T_1 and T_2 be component types, f be a function of the definition $\mathsf{Id}_1 \rightarrow \mathbb{P}(\mathsf{Id}_2)$, $\mathsf{Id}_1{:}\mathsf{T}_1$ and $\mathsf{Id}_2{:}\mathsf{T}_2$ be two parts, and c be a connector connecting $\mathsf{Id}_1{:}\mathsf{T}_1$ and $\mathsf{Id}_2{:}\mathsf{T}_2$.

The specification of an association between f and c means that $\forall\ \mathsf{x} \in \mathsf{Id}_1$: there is a one to many relationship of channels between the component instance named x in T_1 and each component instance in T_2 whose name equals an element in the subset $f(\mathsf{x})$.

If a function f is declared such that it maps natural numbers (or a subset of natural numbers) to natural numbers (or a subset of natural numbers), then mathematical operations such as +, -, * or the modulo function may be used in the definition of f.

A connector that is associated with a constraint function may be also be associated with a multiplicity, but it may not be associated with a channel identifier.

Example

Two alternative specifications LNet are presented in figure 5.32. Two types M and F are declared and defined in the header of the lowermost specification. Also, the function f, mapping instances of M to instances of $\mathbb{P}(\mathsf{F})$, is declared and defined in the header. This function is associated

(as denoted by the label "f|c") with the connector named c. The two specifications in figure 5.32 are equivalent in the sense that the same component instances and channels can be derived from them.

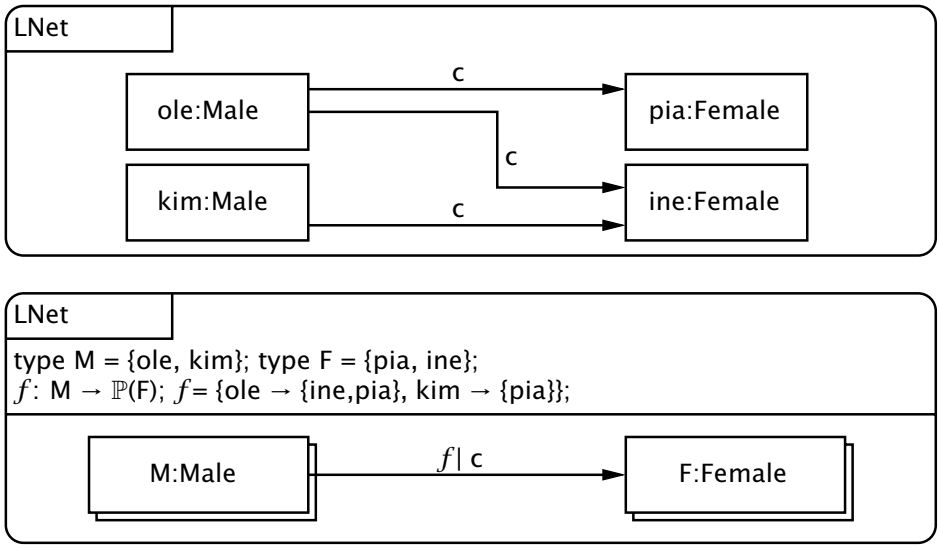

Figure 5.32: Specification containing a constraint function.

5.3.15 Dynamic Connectors

Up until now we have only dealt with connectors that contain channels that always are a part of a composite component/region as long as (1) one or more component instances may send messages on it and (2) one or more component instances may receive messages from it. A *dynamic connector*, however, is a connector that consists of channels that may be created or killed *during* the lifetime of the component instances they connect.

A *static connector* is a connector that is not dynamic.

Example

The specification in figure 5.33 contains two connectors. As indicated, one of these is dynamic. The representation of a dynamic connector is a dashed arrow. The static connector c consists of channels that will always be a part of a region type LNet as long as the component instances they connect are a part of this region. The dynamic connector d, however, consists of channels that may be created or killed *during* the lifetime of the component instances they connect.

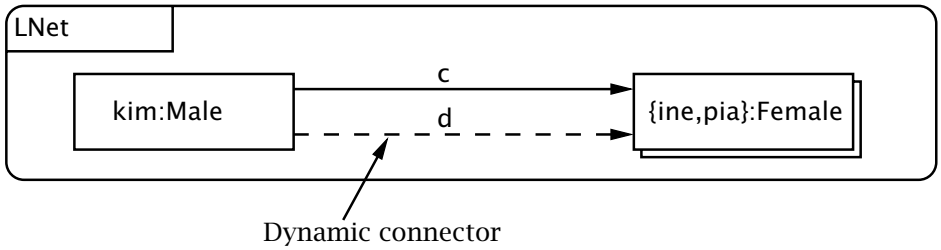

Figure 5.33: Specification containing a dynamic connector.

5.4 Diagrams and Views

In this section we (1) present concepts for classifying specifications into *views*,(2) present a concept for specifying how diagrams are related in time and (3) present the different types of diagrams that exist in MEADOW.

A network may be specified in one or more *diagrams*. In MEADOW, a diagram is named or unnamed and it must have one of the following types:

- views,
- generalisation,
- type or
- snapshot.

In the following we describe these types of diagrams in more detail.

5.4.1 Views Diagrams

A views diagram structures all the diagrams that specify a component or a region type; it declares sub-diagrams and the classification of these sub-diagrams into views. Like regions and composite components, a views diagram may have a header that contains declarations and definitions of constants, types and functions. These definitions and declarations are shared by all sub-diagrams of the views diagram. In addition to this, a views diagram may define how its sub-diagrams are related in time.

All component instance identifiers and channel identifiers that are associated with static parts and connectors must be unique within the same diagram.

If two parts P_1 and P_2 are specified in two different diagrams, and both parts are associated with the same component type and the same identifier, then P_1 and P_2 contain the same component instances. The same principle applies for connectors.

If two dynamic parts P_1 and P_2 are specified in the same diagram or in different diagrams, and both parts are associated with the same component type and the same identifier, then P_1 and P_2 contain the same component instances. The same principle applies for dynamic connectors.

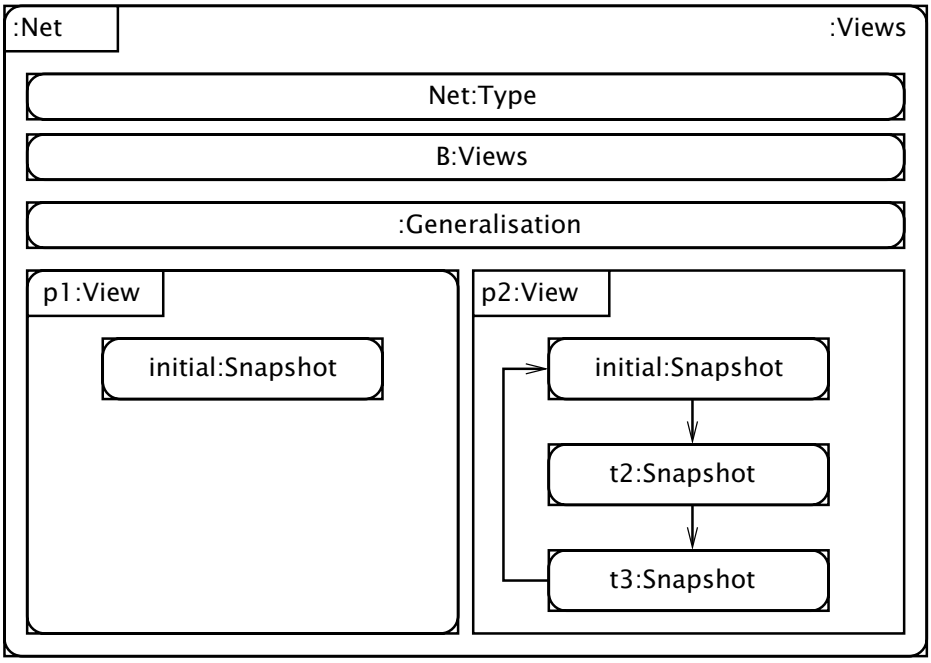

Figure 5.34: Views diagram for region type Net.

Example

Figure 5.34 presents a views diagram for a region type Net. This diagram contains the declaration of one *type* , one *generalisation* and one *views* sub-diagram. Moreover, two *views* named p1 and p2 are declared, each of which contains the declaration of diagrams of type *Snapshot*. Notice that the notation for a views diagram is a box with special edges. The same notation is used on the sub-diagrams that are declared in the views diagram.

The snapshot diagrams specify the structure of Net at given points in time with respect to the views they are contained in. The diagram name initial is reserved, as diagrams so named define the initial structure of a network. In this example, the diagrams named initial define the structure that the instance of type Net exhibits upon its creation.

The arrows in view p2 define exactly how snapshot diagrams are related in time. In this example, when Net at some point in time exhibits the structure defined in diagram initial, it may at some later point in time during computation exhibit the structure specified in t2, then it may exhibit the structure in t3, then it may exhibit the structure specified in initial and so forth. However, Net may *not* go from exhibiting the structure specified in initial to exhibiting the structure specified in t3, without first exhibiting the structure specified in t2. Similarly, Net may not go directly from t2 to initial or from t3 to t2.

Figure 5.35 presents a views diagram for region type B. Notice that this diagram is declared in figure 5.34.

In the following examples we specify most of the diagrams that are declared in figure 5.34 and 5.35.

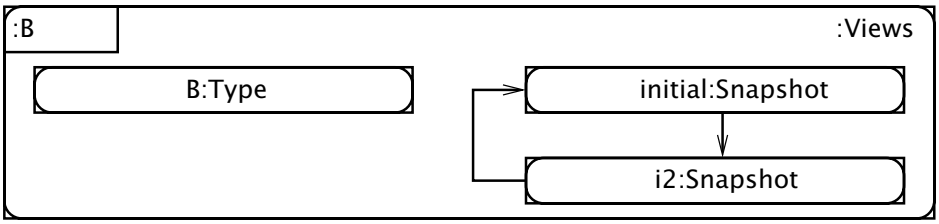

Figure 5.35: Views diagram for region type B.

5.4.2 Type Diagrams

Type diagrams contain the specification of a region or a component type. It contains the specification of all legal potential configuration that all instances of a component type or a region type may exhibit during their lifetime.

Example

The topmost diagram (which was declared in figure 5.34) in figure 5.36 contains the specification of the region type Net. Here, both instances

of component type A and D may or may not be part of the network during computation. Notice also that the connector l is dynamic, while the connectors named p are not. This means that the channels contained in the connectors named p are always part of the network as long as the component instances they connect are part of the network. Connector c, on the other hand, contains channels that may or may not be part of the network even if both the component instances (c1 and c2) they connect are part of the network.

The lowermost diagram in figure 5.36 contains the specification of a region of type B. This diagram was declared in figure 5.35.

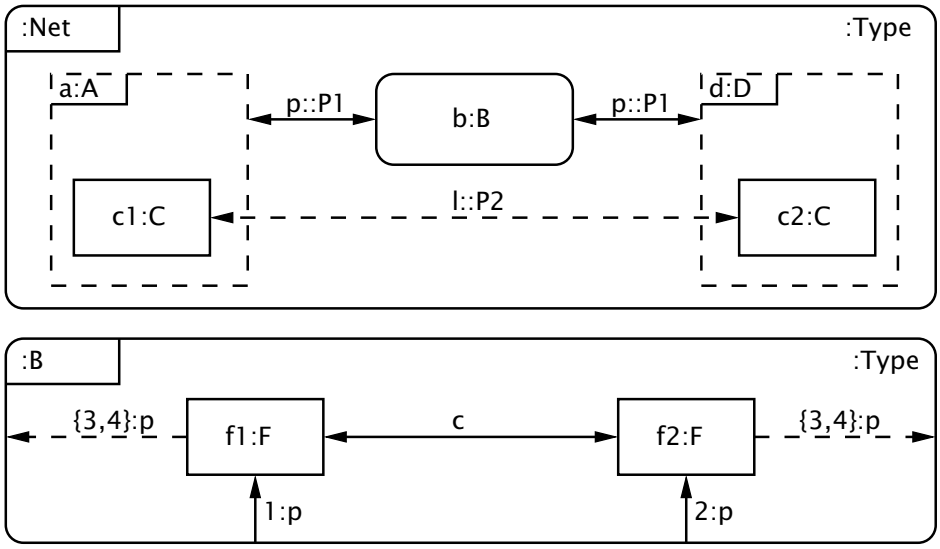

Figure 5.36: Specification of type diagrams.

5.4.3 Snapshot Diagrams

A *snapshot diagram* the configuration that instances of a component type or a region type may exhibit at *a point in time*. A snapshot diagram must be a subset of a type diagram, i.e. it cannot contain configurations that are not specified in the type diagram.

Snapshot diagrams named initial define the configuration that all instances of a component type or a region type must have upon their creation.
Dynamic parts and dynamic connectors cannot be used in snapshot diagrams.

Multiplicities that are used in snapshot diagrams define cardinalities that a connector or a part may have at a point in time.

Example

The topmost specification in figure 5.37 presents a specification of the diagram named initial that was declared in view p1 in figure 5.34. This diagram specifies the initial structure that all instances of region type Net exhibit upon their creation with respect to the view named p1. As can be derived from the diagram; both the component instances a and d are initially part of the region-. Please compare this diagram to the diagram in figure 5.36 to see that a and d may not always be part of the region.

The lowermost specification in figure 5.37 presents the diagram that specifies the structure that all instances of region type Net exhibit upon their creation with respect to view p2.

Figure 5.38 presents a specification of the snapshot diagrams that are declared in figure 5.35. Initially, component instance f1 outputs on a channel named 3, whereas the component instance named f2 outputs on a channel named 4. Then, at some later point in time, as specified in the snapshot diagram named i2, component instance f1 outputs on the channel named 4 and component instance f2 outputs on the channel named 3.

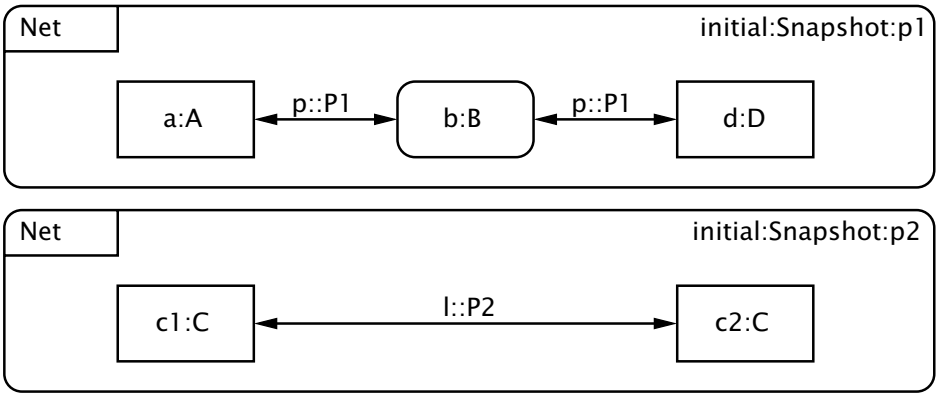

Figure 5.37: Specification of snapshot diagrams.

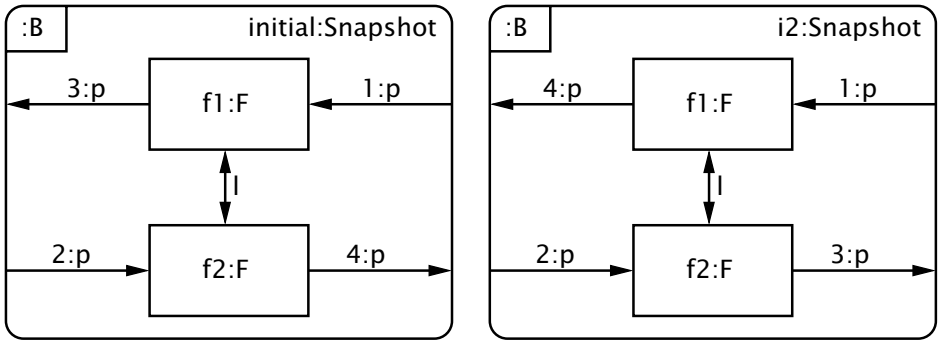

Figure 5.38: Specification of the snapshot diagrams for region type B.

5.4.4 Generalisation Diagrams

Diagrams of type *Generalisation* contain generalisation relationships between component types.

Example

Figure 5.39 presents a specification of the generalisation diagram that was declared in figure 5.34. Here, component type C is specified as supertype for component types D and E.

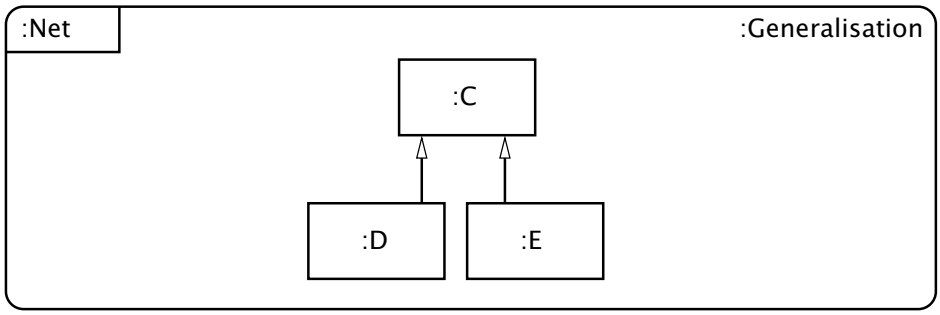

Figure 5.39: Specification of a generalisation diagram.

Chapter 6

Case Studies

The objective of this chapter is (1) to demonstrate the suitability of MEADOW with respect to the success criteria given in section 3.4 and (2) give an example-driven introduction to the central modelling constructs of MEADOW.

Section 6.1 presents a case study of a network with arbitrary, but fixed number of components. Sections 6.2 through 6.4 present case studies of networks with tree, bus and ring topologies, respectively. These case studies indicate the appropriateness of MEADOW with respect to the success criteria related to the problem of generality and the problem of scalability.

Section 6.5 gives a case study of a large network that has a topology that is not strongly patterned. This case study indicates the appropriateness of MEADOW with respect to the success criteria related to the problem of scalability.

Sections 6.6 through 6.8 give case studies of object-oriented, ad hoc, and mobile code networks, respectively. The purpose is to demonstrate the appropriateness of MEADOW with respect to the problem of expressing dynamic reconfiguration.

6.1 Configuration with N Components

In practise it is often necessary to specify networks of an arbitrary, but fixed number of components N. In that case we know that N denotes a natural number, but we do not know which one.

The task is to model the general architecture of a SIMD (single-instruction stream-multiple-data stream) machine [24]. Typically, a SIMD machine

consists of a control unit, N processors, N memory modules, and an interconnection network. The control unit broadcasts instructions to the processors, and all active processors execute the same instruction at the same time. As an example of the possible size of SIMD machines, one constructed system includes 2^{14} simple processing units [24].

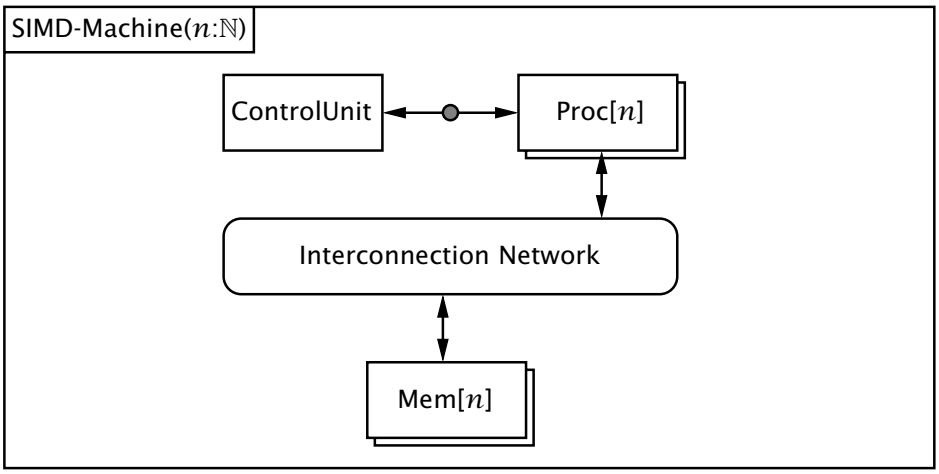

Figure 6.1: SIMD-Machine.

One way to view the physical structure of a SIMD machine is to position the interconnection network between the processors and the memories. Figure 6.1 presents a MEADOW specification of this structure. The SIMD machine itself, is modelled as a parameterised composite component of type SIMD-Machine. Notice that the a formal parameter n of type $\mathbb{N}$ is declared in the top left corner of the composite component. This parameter is used in the internal structure of SIMD-Machine to specify the number of processors and memory units the SIMD machine consists of. These processors and memory units are specified by component instances of type Proc and Mem, respectively. ControlUnit represents the control unit and Interconnection Network represents the interconnection network. All connectors in the specification are bi-directed. Notice that a merge-split node is associated with the connector that connects the part of type ControlUnit to the part of type Proc. This connector consists of two shared channels that each constitute a merge-split relationship.

For clarity, figure 6.2 presents a specification of a SIMD machine with four Proc and Mem component instances. The network specified in figure 6.1 is equivalent to the network specified in figure 6.2 for $n = 4$. Notice that two topmost connectors contain mere-split relationships of

channels. One of the relationships is actually equivalent to a split one to many relationship, and the other is equivalent to a merged many to one relationship of channels. This means that the topmost merge node and the topmost split node in the figure 6.2 can be omitted.

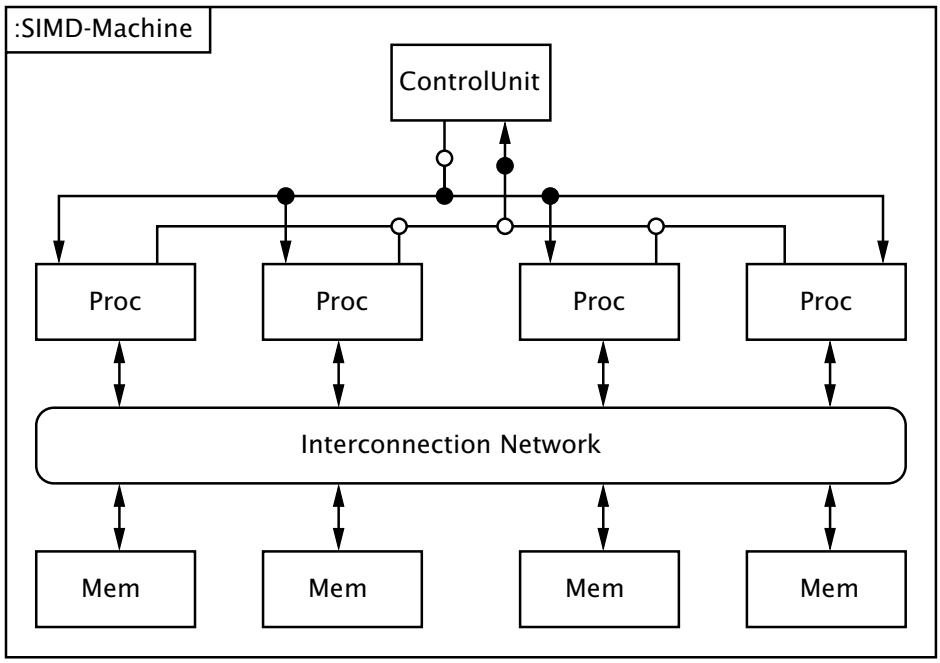

Figure 6.2: Alternative specification of SIMD-Machine for a fixed number of components.

6.2 ARDIS

ARDIS is a two-way radio service developed as a joint venture between IBM and Motorola and first implemented in 1983. The network consists of four network control centres with 32 network controllers distributed through 1250 base stations in 400 cities in the U.S. Remote users access the system from laptop radio terminals which communicate with the base stations. For a more detailed description of ARDIS, we refer to [8].

The MEADOW specification of ARDIS, which is based on a description of the ARDIS topology given in [8], is presented in figure 6.3. Here, the four control centres are specified by four component instances of type ControlCenter, the 32 network controllers are specified by the 32 com-

ponent instances of type Controller, and finally the 1250 base stations are specified by the component instances of type BS.

The connector connecting the part of type ControlCenter and the part of type Controller, contains channels that constitute a one to many and a many to one relationship that connect all instances contained in these parts.

The cardinality constraint ([1]) on the lowermost connector in figure 6.3, specifies that each component instance of type BS is connected to exactly one component instance of type Controller. The exact number of instances of type BS that are connected to each component instance of type Controller is not specified. This would however have been possible by using a constraint function. An example is presented in figure 6.4, where the relationship between component instances of type BS and Controller is defined by the constraint function f. Specifically, $f(x) = x \ mod \ 32$, where mod is a mathematical modulo function.

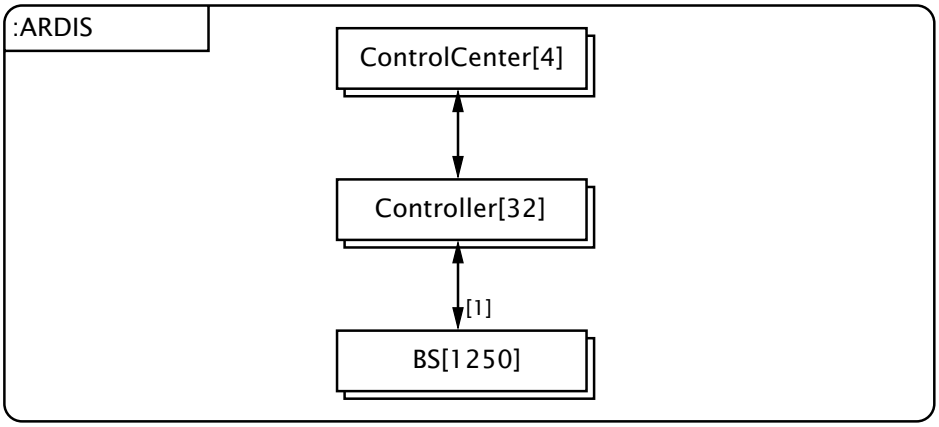

Figure 6.3: ARDIS network topology.

6.3 Ethernet

The Ethernet has been one of the most successful local area networking technologies [20]. The Ethernet is a multiple-access network, meaning that a set of components send and receive messages over a shared channel. You can therefore think of the Ethernet as being like a bus that has multiple stations plugged into it.

We model the topology of a simple Ethernet network to (1) show how

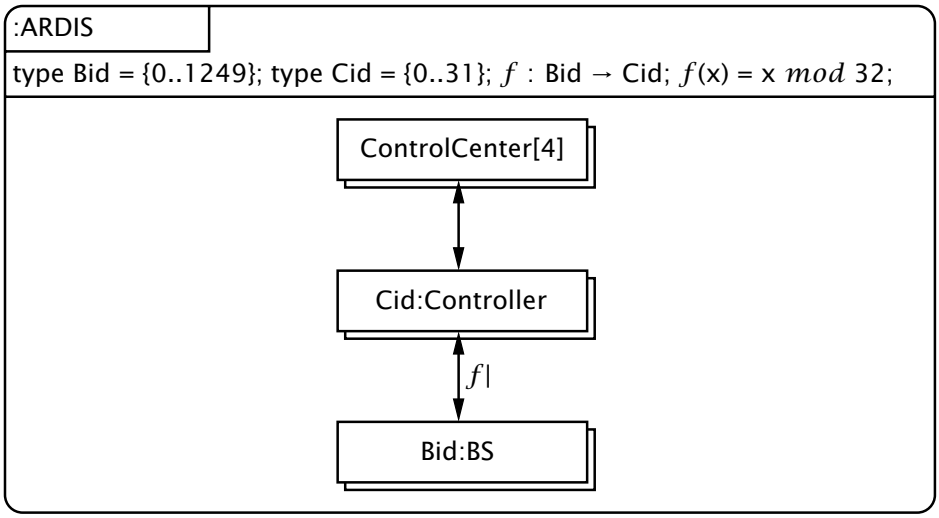

Figure 6.4: ARDIS network topology with a constraint function.

MEADOW can be used to model a general bus topology and (2) show how the concept *generalisation* may increase scalability in a specification.

As declared in the views diagram presented in figure 6.5, the region type Ethernet is specified in the two diagrams Ethernet:Type, and g:Generalisation. Diagram g, presented in figure 6.6, specifies the generalisation associations between the components that constitute Ethernet. Here, we see that component types Printer, Disc drive, Workstation and Mainframe all share the common supertype Host.

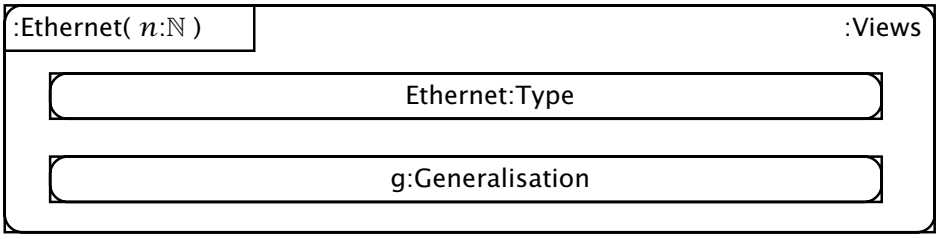

Figure 6.5: Views diagram of Ethernet.

Figure 6.7 presents the type diagram of Ethernet that specifies the potential internal structure of Ethernet. The connector in the specification contains one merge-split many to many relationship. A connector such

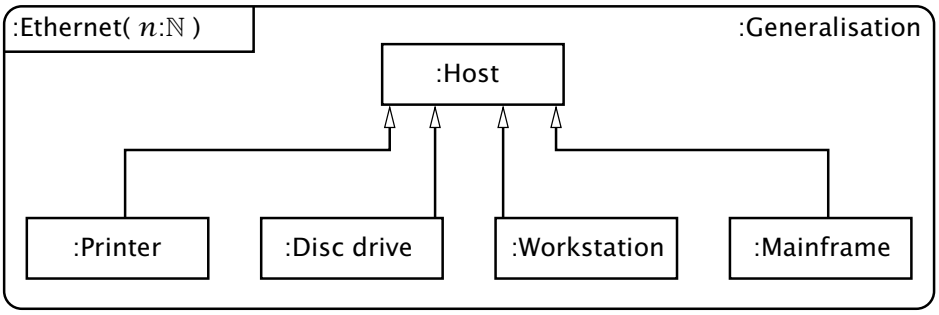

Figure 6.6: Specification of diagram g:Generalisation.

as this, that is associated with a merge-split node and that has both of its ends attached to the same part, does by definition form a bus pattern. Specifically the connector in the specification contains one channel that all component instances in the part labelled Host[n] may send messages on and receive messages from. Notice how the generalisation associations has helped increase the scalability in the specification since the subtypes of Host are abstracted away.

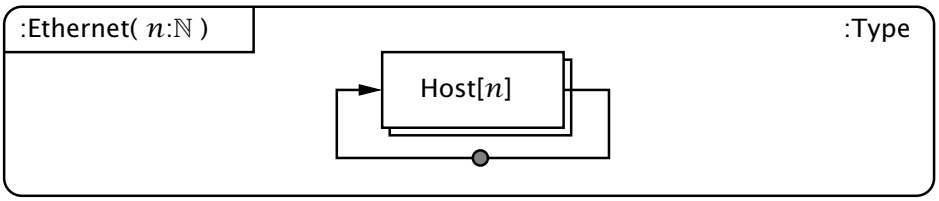

Figure 6.7: Specification of region type Ethernet.

Figure 6.8 illustrates what the specification presented in figure 6.7 looks like for a fixed number of components. Here the three topmost boxes and the three lowermost boxes represent the same three components. For example the two boxes labelled h1:Host represent the same component.

The technique of specifying a bus topology that is presented in figure 6.7 has one apparent weakness. Looking at figure 6.8, is seems that every message that is sent by a component instance on the bus, will also be received by the same component instance. This weakness would have to be addressed in the implementation of the merge and the split nodes. But as previously stated, exactly how merge and split nodes should be implemented is underspecified in MEADOW.

One way of avoiding the problem all together, is to use components that act as merge and split nodes, and then connect these components in a bus pattern with the help of a constraint function. A similar solution is used in the specification of a ring topology in the next section.

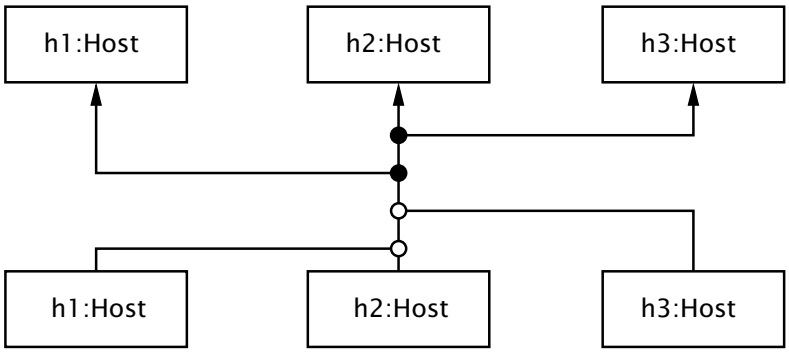

Figure 6.8: Illustration of an Ethernet network.

6.4 Token Rings

Alongside the Ethernet, token rings are the other significant class of shared-media networks [20]. As the name suggests, a token ring network consists of a set of components connected in a ring. Data always flows in a particular direction around the ring, with each component receiving messages from its upstream neighbour and then forwarding them to its downstream neighbour. Two notable token ring standards are the FDDI standard and the IEEE standard 802.5 [20].

The diagram presented in figure 6.9 specifies the topology of a token ring network. Here, the formal parameter n is used to specify the number of hosts and nodes that are in the network. Note that n must be greater than 1 in order for this specification to work.

The part labelled Nid:Node means that for each e $\in$ Nid: exactly one component instance whose name is e of type Node is specified. These component instances are connected in a ring as specified by the constraint function named $ring$ that is defined in the header. Recall that a constraint function $f(x) = y$ associated with a connector c, specifies a channel between the component instances named x and y contained in the parts connected by c. For example, in figure 6.9, a channel is specified between component instance 0:Node and 1:Node since $ring(0) =$

$(0 + 1)\ mod\ (n - 1) = 1$ (for $n > 1$).

Without its associated cardinality constraints ([1]), the connector between the part labelled Nid:Node and the part labelled Host[n], would have represented two many to many relationships between the two parts. However, the multiplicities constrains the many to many relationships such that each component instance in the parts connected by the connectors may only have exactly one output channel and exactly one input channel.

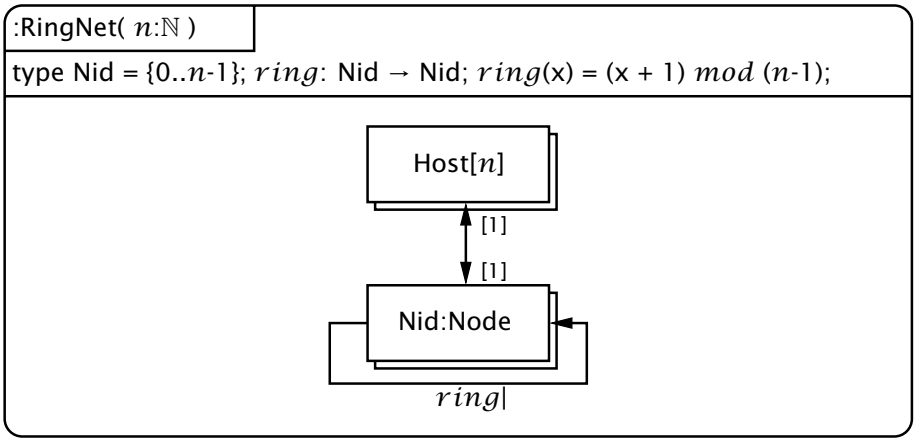

Figure 6.9: Specification of a token ring network.

For clarity, figure 6.10 presents an alternative specification of the token ring network for a fixed number of components. RingNet(4) (in figure 6.9) and RingNet2 (in figure 6.10) specify the same network.

6.5 The NTN ATM Network

The NTN network is a Canada-wide ATM National Test Network. We will use MEADOW to model this network based on a simulation model of the NTN ATM network [26]. "The simulated network topology has 54 ATM switches, and spans a geographic distance of approximately 3000 kilometres"[26].

The MEADOW specification of the NTN ATM network is presented in figure 6.11. Here, the region NTN ATM is divided into five sub-regions named Rnet, LARGnet, OCRInet, Wnet and RISQ. These sub-regions are connected by component instances of type ATM Switch. The four types

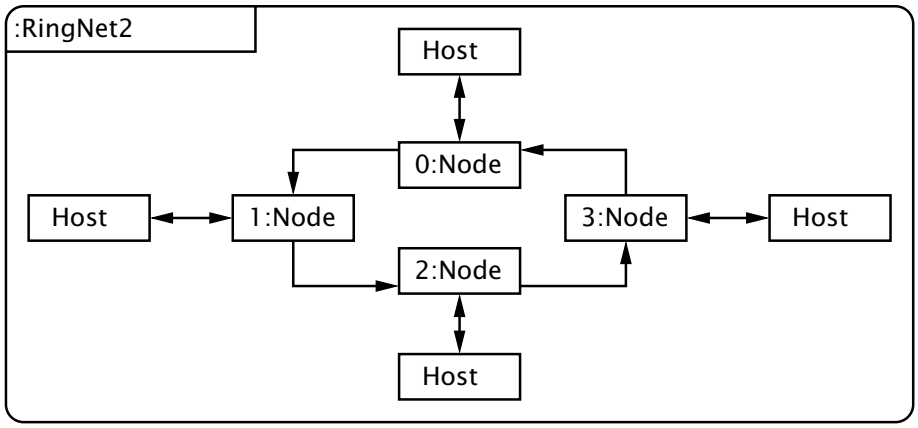

Figure 6.10: Alternative specification of a token ring network.

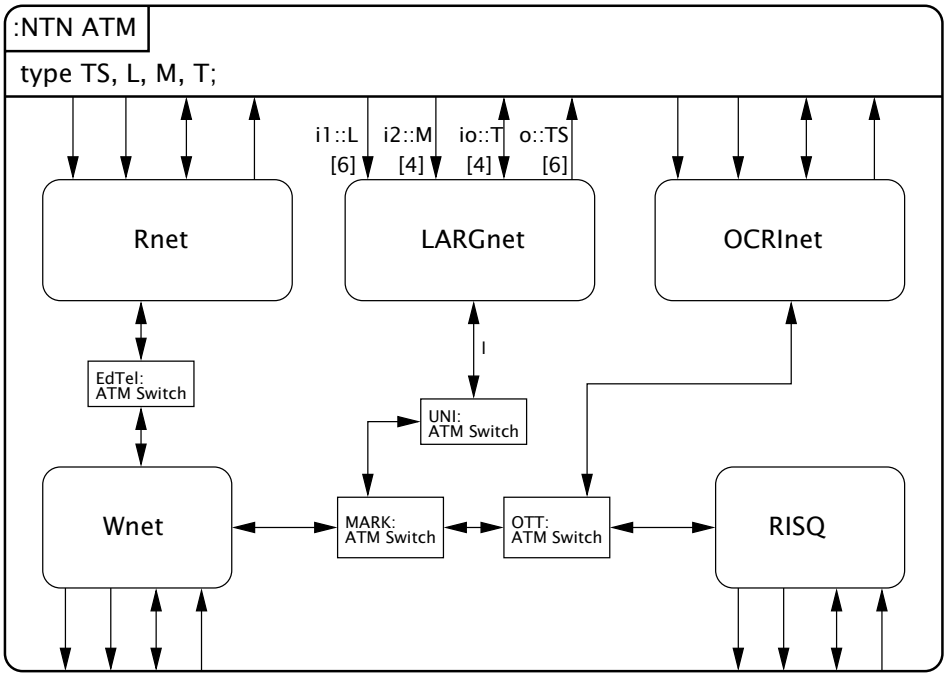

Figure 6.11: Specification of NTN ATM network.

that are declared in the header of NTN ATM define the kind of messages that can be communicated to and from the sub-region LARGnet and the environment of NTN ATM. We have only labelled the connectors going to and from the sub-region LARGnet, since LARGnet is the only sub-region we intend to model in detail.

In the following, we present three alternative specifications of LARGnet. We will not present detailed specifications of the other sub-regions that NTN ATM consist of since their internal structure is in essence very similar to the internal structure of LARGnet.

As can be seen from the specification presented in figure 6.12, the two component types that LARGnet consist of are ATM Switch and End-node. The labels associated with the connectors between component instances of type End-node and the environment of LARGnet, correspond to the connector labels in figure 6.11. The message type label is suppressed in the figure 6.12, it can however be derived from figure 6.11. For example that all channels derived from connectors named io must have a message type T.

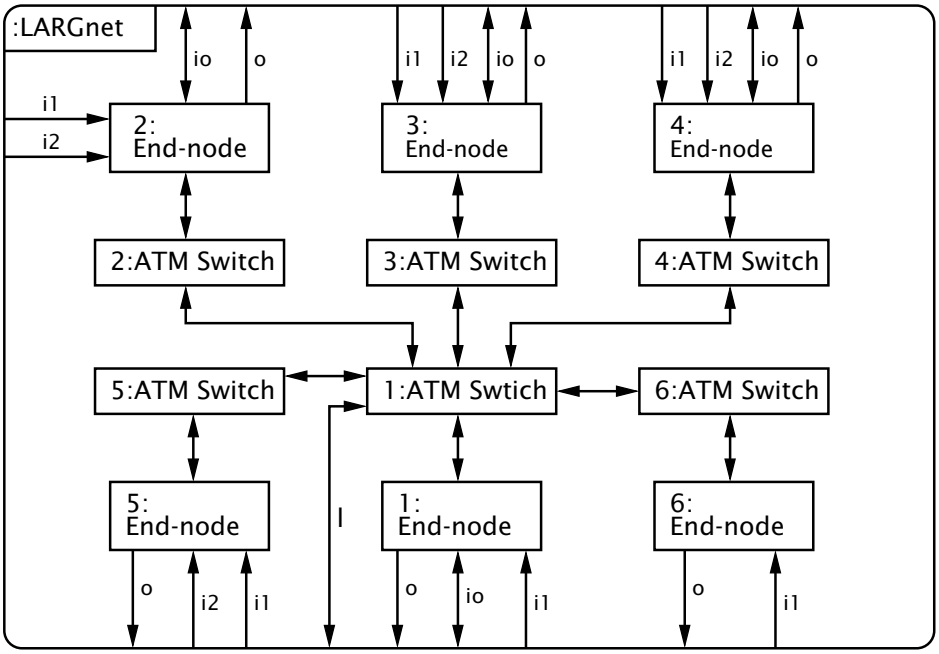

Figure 6.12: Specification of LARGnet, alternative 1.

Figure 6.13 presents an alternative specification of LARGnet. Here we

have used parts of cardinality greater than one. Specifically, we have grouped the component instances named 2, 3 and 4 of types ATM Switch and End-node in two parts, each of cardinality three. Comparing figure 6.12 and figure 6.13, we see that the number of parts used in the specification has been reduced from 12 to 8.

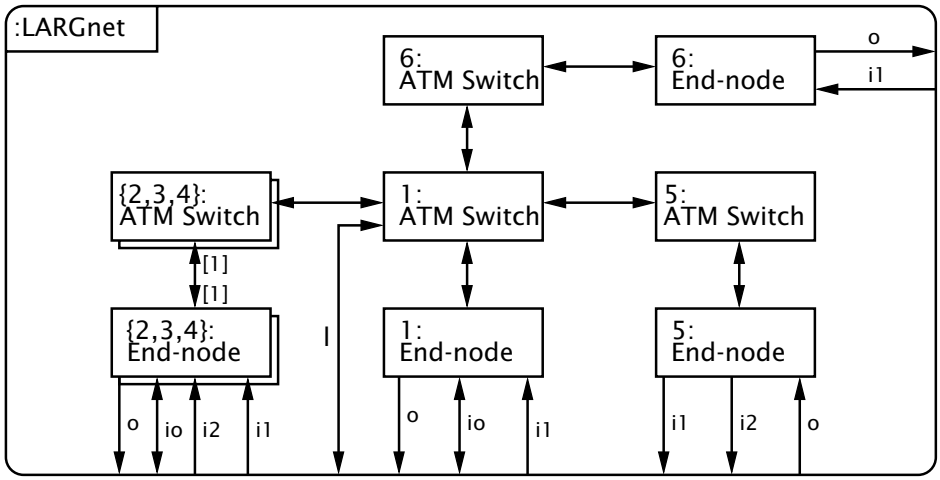

Figure 6.13: Specification of LARGnet, alternative 2.

The third alternative specification of LARGnet is presented in figure 6.14. Here we have used identifier constraints to reduce the number of parts needed in the specification further. The component instances named 2, 3, 4, 5 and 6 have been grouped into two parts. Not all of these instances are associated with the connectors named io and i2, that is why we have associated these connectors with identifier constraints. Specifically, the bi-directed connector io is associated with the identifier constraint {2..4} because only the component instances named 2, 3 and 4 in the leftmost part of type End-node are associated with the channels that may be derived from connector io. Similarly, only component instances named {2..5} in the left most part of type End-node may input messages on the channels contained in the connector i2.

If we compare figure 6.12 and figure 6.14, we see that the number of parts have been reduced from 12 to 4. Hence the latter specification is less space consuming than the former, since it is the representation of parts that take up most space. Moreover, it is easier to add/remove components in the latter specifications, since this can be done by manipulating the identifier types and the identifier constraints instead of adding/removing parts.

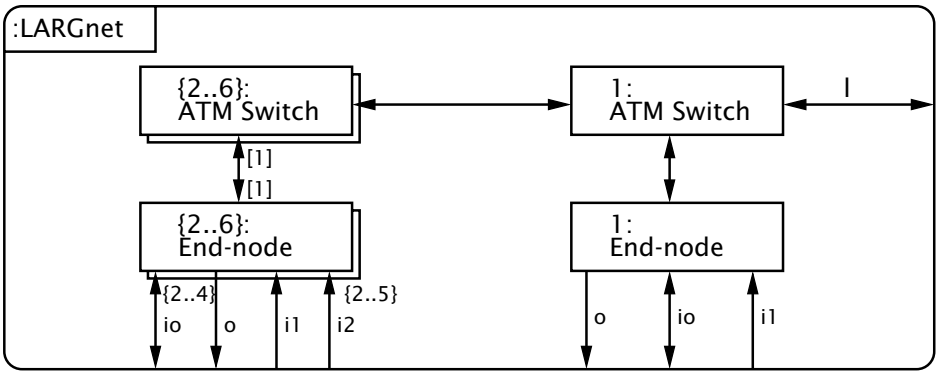

Figure 6.14: Specification of LARGnet with identifier constraints.

Note that the specification in figure 6.14 (actually also the specification in figure 6.13) is a slight underspecification figure 6.12. From the connector connector connecting the part labelled {2..6}:ATM Switch and the part labelled {2..6}:End-node, we can only derive that one component in the first part is connected to exactly one component in the second part. We cannot know if the component instance named 2 of type ATM Switch is connected to the component instance named 2 of type End-node as in the specification in figure 6.12. This problem could be solved by associating the connector with a constraint function instead of the cardinality constraints (labelled [1]). This specification is presented in figure 6.15.

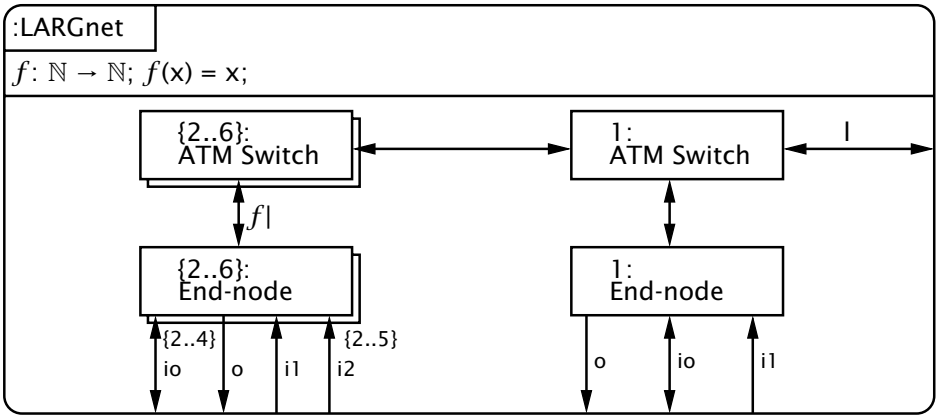

Figure 6.15: Specification of LARGnet with constraint function.

6.6 An Object-Oriented Network Example

In the following case study we model an object-oriented network called ONet. ONet consists of three objects: two objects of type Sender and one object of type Receiver. Each sender object always has a reference to the other sender object, and one and only one sender object has a reference to the receiver at any given time. The sender objects may exchange the reference to the receiver with each other in order to send messages to it.

We model ONet as a region type named ONet. The views diagram of ONet presented in figure 6.16, declares the diagrams that specify ONet. The diagram declarations of type Type and Snapshot declares diagrams that specify the internal structure of ONet. The diagram named initial specify the initial structure of ONet, whereas the diagram named step2 specifies the internal structure of ONet at some other point in time. The arrows define exactly how the snapshot diagrams are related in time. According to this definition, ONet must initially exhibit the structure specified by diagram initial, then at some later point in time it may exhibit the structure specified in step2, then it may exhibit structure specified in initial and so on.

The way in which one can specify the relationship in time between snapshot diagrams is rather basic in MEADOW. The reason is that these kinds of dataflow diagrams are not within the focus of this thesis. Also, one might use existing languages such as for example Activity Diagrams in UML [19] for this kind of specification by replacing activities in these diagrams with snapshot diagrams.

A formalism for relating snapshot diagrams in time is not only a way of increasing the understandability appropriateness of specifications, it also provides a basis for automatic model checking of specifications. This can be done by comparing the specification of how snapshot diagrams are related in time with a specification of the behaviour of components. For example, assume three snapshot diagrams initial, step2 and step3 that specify the internal structure of a region R. Assume further that the three diagrams are related in time such that initial happens before step2, step2 happens before step3 and step3 happens before initial and so on. If the network at some point during computation goes from exhibiting the structure specified in initial to exhibiting the structure specified in step3 without first exhibiting the structure specified in step2, then the behaviour of components that R consists of is inconsistent with the specification of how the snapshot diagrams are related in time.

The type diagrams may also be used as a basis for a similar kind of model checking. One can check if a network exhibits a structure during computation that is not specified in the type diagram. And if that is the case, then the behaviour of the network is inconsistent with the specification of the type diagram.

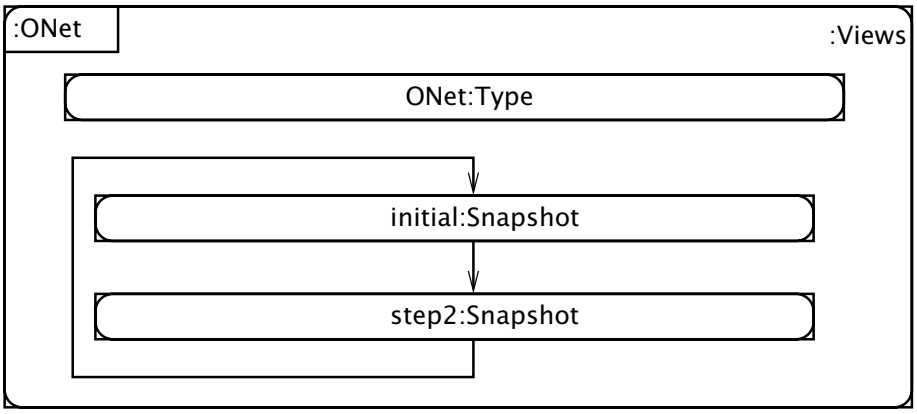

Figure 6.16: Views diagram of ONet.

The diagram presented in figure 6.17, specifies the legal potential structure of ONet. All three component instances specified in the diagram are always part of ONet. The same applies for connector s and o. Both the senders may or may not output on the channel r.

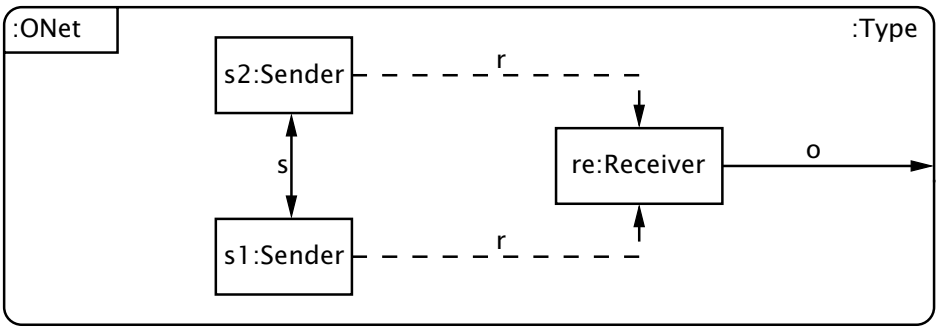

Figure 6.17: Specification of the potential configurations of ONet.

The initial structure of ONet is specified in the initial:Snapshot diagram presented in figure 6.18. Notice here that the lowermost sender in the specification has an initial output channel to the receiver.

Diagram step2 (presented in figure 6.18) is similar to diagram initial. The difference is that the topmost component instance of type Sender now has a second output channel instead of the lowermost component instance of type Sender.

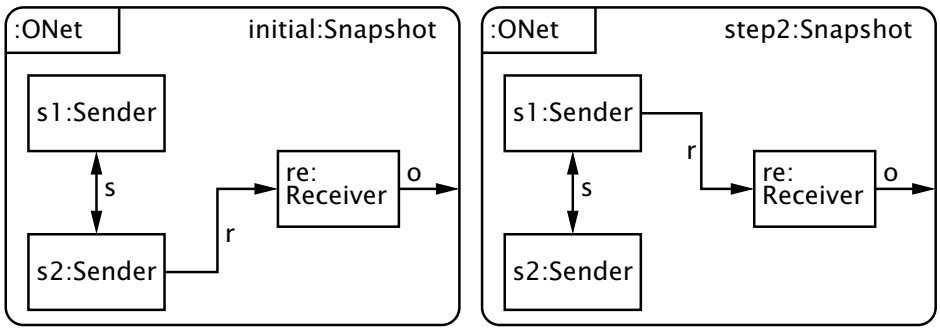

Figure 6.18: Specification of snapshot configurations of ONet.

6.7 Battlefield Control System

In the following we model a network called the Battlefield Control System (BCS) described in [5]. The system is used to control the movement, strategy and operations of troops in the battlefield.

BCS consists of commander component and a number of solider components. The commander component acts as a server and the soldiers components are its clients. One of the soldiers acts a backup. Upon failure of the commander, the backup will take over as the new commander. Communication between clients and the server is only through encrypted messages sent via a radio modem. The radio modem uses a shared communication channel; only one component can broadcast at any moment.

BCS is similar to an ad hoc network in the sense that all components and channels in the network may be created or killed during computation. However, BCS is not an ad hoc network because BCS has a so-called central administration, i.e. one component always acts as a server while the other components act as clients. An ad hoc network could be modelled in MEADOW for example by modelling BCS without a central administration. But in order to highlight the features of MEADOW, we believe that it is more interesting to model BCS with a central administration.

Our approach in this case study is not to give a full specification of BCS, but the model what the network looks like before and after the (possible) breakdown of the commander.

We model the system at a logical level (i.e. we do not consider the physical level) as a region type named BCS. The three diagrams that specify the internal structure of BCS are declared in the views diagram presented in figure 6.19. The diagram named BCS of type Type, specifies the potential internal structure of BCS. initial and breakdown are of type Snapshot and specify the internal structure of BCS at given points in time. Specifically, initial specifies the initial structure of BCS and breakdown specifies the structure of BCS upon the breakdown of the commander.

Notice that three formal parameters are defined in the diagram: One type named Rid and two constants named b and c of type $\mathbb{S}$.

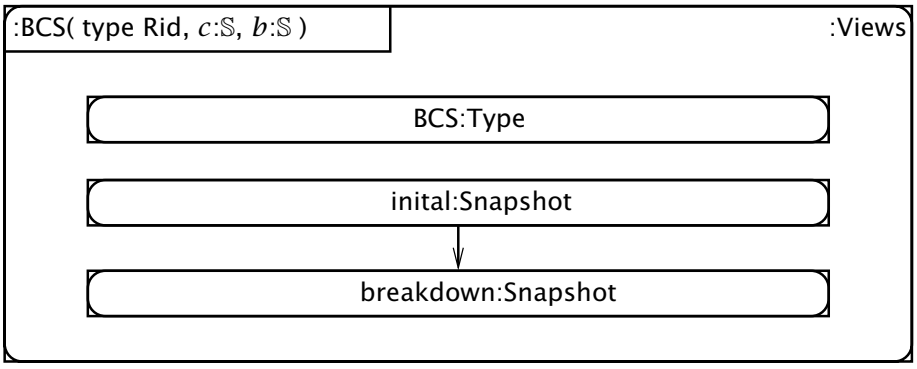

Figure 6.19: Views diagram for region type BCS.

The specification of the type diagram is presented in figure 6.20. The commander which is modelled as a component instance named c of type Commander, is contained in the part labelled c:Commander. This part is dynamic, meaning that the component instance in the part may or may not be part of the network at a given point in time. This models the fact that the commander may disappear from the network. Similarly, the part labelled Rid:Soldiers represents the soldiers of the network, and the part labelled b:Backup represents the backup. These parts also dynamic, hence the component instances of type Soldier and Backup may or may not be part of the network at a given point in time. Notice that the number of potential soldiers that can be part of the network is equal to the cardinality of the type Rid, and that an instance of region type BCS may only contain one commander and one backup.

The connector r represents the connection between the backup and the commander. The leftmost connector c represents the connection between the commander the soldiers whereas the rightmost connector c represents the connection between the component instance of type Backup to the component instances of type Solider. This latter connector is dynamic, hence the channels contained in this connector may not always be part of the network even if all the component instances that are connected by the channels are part of the network.

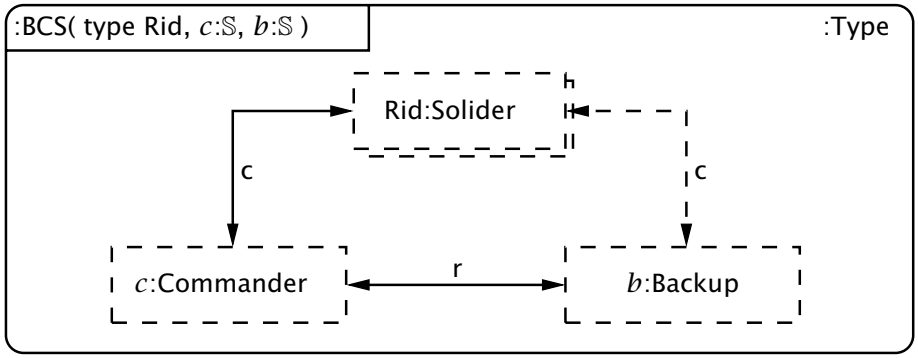

Figure 6.20: Specification of the potential structure of region type BCS.

Figure 6.21 presents the two snapshot diagrams declared in the views diagram of BCS. The diagram named initial specifies the initial internal structure of BCS. As can be seen from the diagram, there is no initial connection between the backup and the soldiers.

Diagram breakdown specifies the internal structure of BCS upon breakdown of the commander. Here, a connection is established between the soldiers and the backup as a result of the commander not being part of the network anymore. The multiplicity ([0..]) associated with the part of type Solider, specifies that part may contain zero to #Rid (where #Rid is the cardinality of Rid) component instances at the time of breakdown. Similarly, the multiplicity associated with the part of type Backup specifies that zero or one component instance of type Backup may be part of the network at the time of breakdown.

6.8 Mobile Code Network Example

In this case study we model a framework for building context-aware applications in ubiquitous and mobile computing settings. The goal of the framework is to offer a location-aware system in which spatial regions

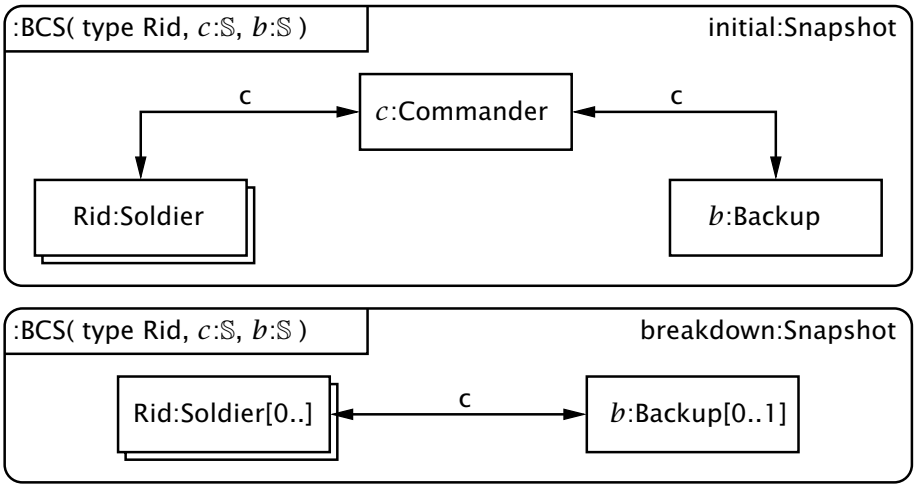

Figure 6.21: Specification of snapshot diagrams.

can be determined to within a few square feet, so that one or more portions of a room or a building can be distinguished.

The framework consists of two parts: (1) mobile agents and (2) local information servers, called LISs. The former offers application-specific services which are associated with physical entities and places. The latter provide a layer between underlying locating systems and mobile agents. Each LIS provides the agents with up-to-date information on the state of the real world, such as the locations of people, places and things, and the destinations that agents should migrate to. For a more detailed description of the framework, we refer to [23].

We model a simple network called PdaNet that is based on this framework. The physical components that constitutes PdaNet are, a sensor, a LIS, a computer associated with a tag and a personal digital assistant (PDA) also associated with a tag. The tags makes it possible for the sensor to locate the physical entities (the computer and the PDA). Each tag periodically transmits a unique identifier via infrared light that can be received by the sensor. The tag associated with the computer is always within the presence of the sensor, while the tag associated with the PDA may or may not be within the presence of the sensor. The sensor uses a radio frequency to notify the LIS of the tags that are within the presence of the sensor at a given point in time. The LIS communicates with the computer and the PDA via the same radio frequency.

At a logical level, both the computer and the PDA each have a run time

system (the Java based mobile agent system Mobile Spaces [22] for example). These run time systems can run a mobile agent we call app. Moreover, app moves from the runtime system on the computer to the runtime system on the PDA when both the previously mentioned tags are within the presence of the sensor.

A views diagram of PdaNet is presented in figure 6.22. Here, two views are defined, Physical and Logical. The type diagrams in these views specify the potential structure of PdaNet with respect to the views they are contained in. Three types are declared and defined in the header of the views diagram. These types are used as message types in the specification of the three diagrams that are declared in figure 6.22. "L = :MA" means that the type L equals all instances of the component type MA. Consequently, connectors associated with the message type L contain channels that may transport component instances of type MA.

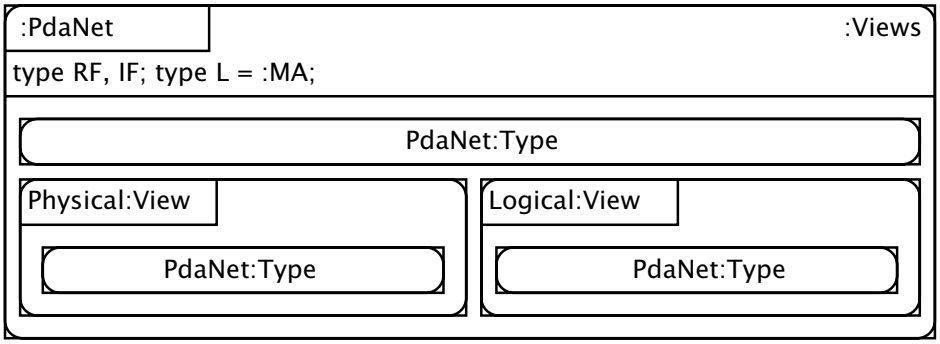

Figure 6.22: Views diagram of PdaNet

The specification of the type diagram that is not contained in any view is presented in figure 6.23. The region of type Cell models the infra red transmission radius of the sensor. The region containing comp:AH (the computer) and Tag (the tag associated with the computer) is always contained within this transmission radius. However, the region containing pda:AH and Tag (the PDA and its associated tag) may or may not be within the transmission radius at a given point in time.

The connector named if of message type IF represents the infrared communication between the tags and the sensor. Similarly, the connector named rf of message type RF represents the radio communication between the computers, the sensor and the LIS. Both these connectors represent connectivity at the physical level. This is contrary to the connector labelled ::L which represents connectivity between the two composite

component instances of type MS (the run time environments) at a logical level. Moreover, since the message type L equals the component type MA (as defined in figure 6.22), instances of type MA can be sent along the channels contained in this connector. Specifically, the component instance named app of type MA (the mobile agent) may be sent from one instance of type MS to another via the channels contained in the connector labelled ::L.

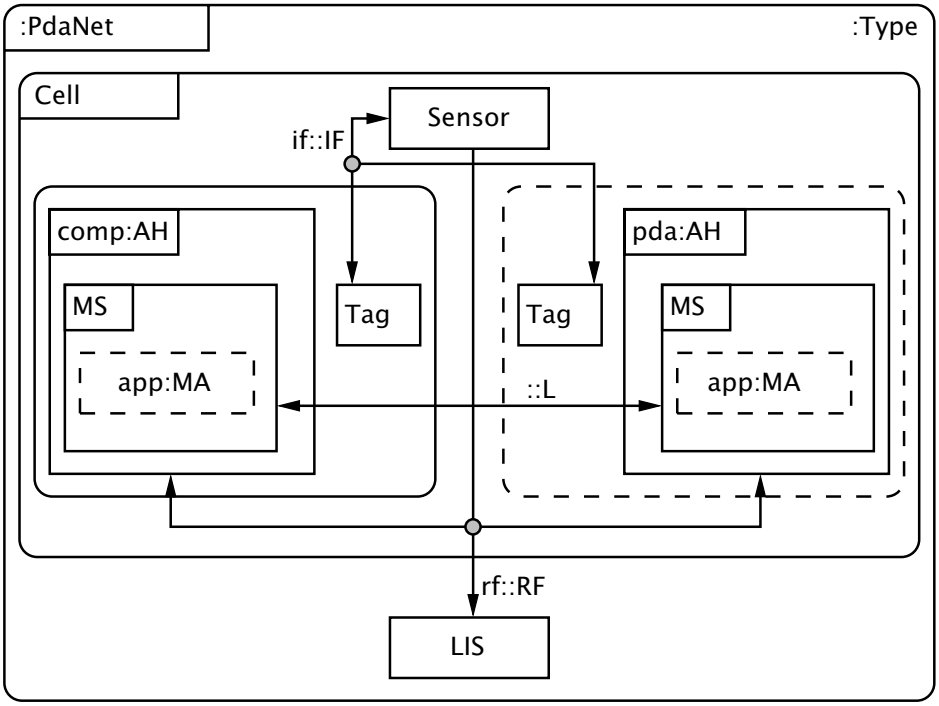

Figure 6.23: Specification of type diagram.

Figure 6.24 presents a diagram containing the specification of PdaNet with respect to the physical view. This diagram is essentially a subset of the diagram specified in figure 6.23, where only the parts connected by connectors of message types IF and RF are shown.

The diagram in figure 6.25 specifies PdaNet at a logical level, i.e. the physical level is abstracted away. Note that the diagram in figure 6.23 may be derived from the two diagrams p and l.

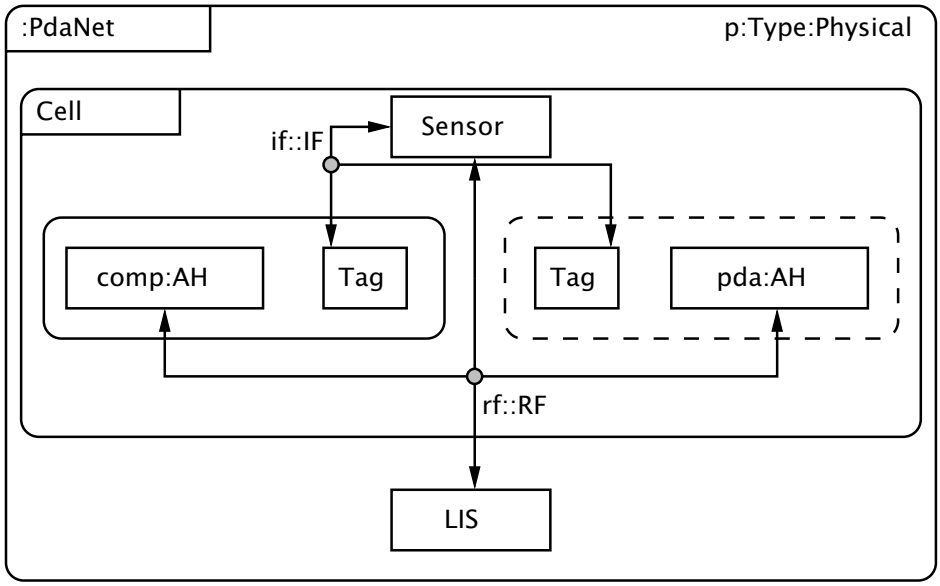

Figure 6.24: Specification of PdaNet with respect to the physical view.

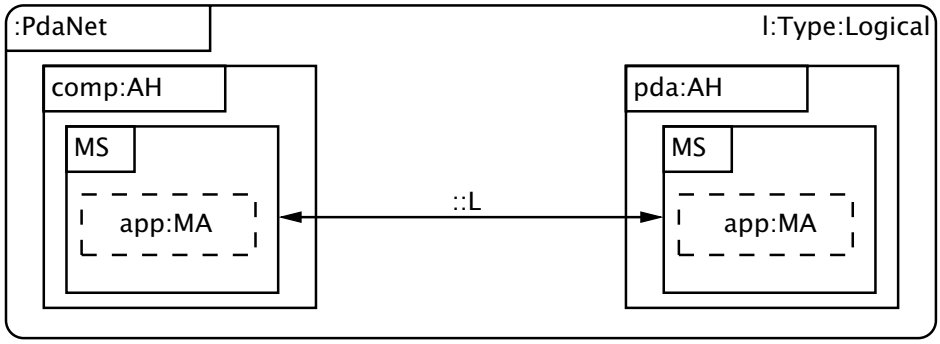

Figure 6.25: Specification of PdaNet with respect to the logical view.

Chapter 7

Evaluation of MEADOW

The objective of this chapter is to evaluate MEADOW with respect to the success criteria that are given in section 3.4.

7.1 Comprehensibility Appropriateness

In this section we argument that MEADOW fulfils the success criteria: *"MEADOW should have a high comprehensibility appropriateness"*. The aspects related to comprehensibility appropriateness that were given in section 3.2 are used as input for the evaluation.

7.1.1 Conceptual Basis

The phenomena of the language should be easily distinguishable from each other.

Almost all concepts of MEADOW are easily distinguishable from one other in the sense that they do not result in construct redundancy. The exception is the concept of a region (see section 5.2.9), which is a special case of a composite component. Although the concept of a region does not add semantics to MEADOW as such, it does *represent* a phenomena in the *real world* that is substantially different from the phenomena that a composite component represents. A composite component represents and entity, and this entity may be seen as being on a different level of abstraction than the entities that are represented by the sub-components of the composite component. This is contrary to the region, which merely is a way to group sub-components and other sub-regions at the same level of abstraction. This is why we have chosen to have the concept of regions in the language even though the region is really a special case of a composite component.

The phenomena must be general rather than specialised.

All concepts in MEADOW are general rather than specialised in the sense that none of the concepts are tailored to describe particular kind of components, networks or topologies. For example, although it is possible to specify the bus topology in MEADOW, the concepts used for this specification are not specialised for this purpose only.

The phenomena should be composable, i.e. one should be able to group statements in a natural way.

Most concepts in MEADOW are ways of grouping components and channels. In particular this applies the concept of parts, connectors, composite components and regions. In a addition, the concept of a views diagram provides a way to group diagrams.

The language must be flexible in precision and in the level of detail.

According to [14], being flexible in precision means that large numbers of phenomena should be organised hierarchically and/or in sublanguages, making it possible to approach the framework at different levels of abstraction or from different perspectives. This is to some extent realised in MEADOW through to the concepts of *hierarchy* and *views*. Also, the use of named component instances and channels versus unnamed component instances and channels provides a way of varying the level of precision in a specification. This because specifications containing named component instances and channels are in general more precise than specifications with unnamed channels and component instances. The same is true for constraint functions versus cardinality constraints, since constraint functions are in general more precise than cardinality constraints.

The use of phenomena should be uniform throughout the whole set of statements that can be expressed within the language.

This aspect has, almost from the beginning, been considered during the development of MEADOW. In particular this is reflected in the way in which instances and types are handled in MEADOW. The two most important types are component types and connectors types. Instances of these types can be specified in a uniform way, i.e. by using similar concepts of multiplicities and identifiers.

7.1.2 External Representation

Symbol discrimination should be easy.

Symbol discrimination in MEADOW is for the most part easy in the sense that different symbols are used for different phenomena. The exception that the same notation (a box) is used for both the specification of component types and parts. But we do not believe that this is a big problem, since similar languages like SDL and UML also use the same notation for type and instance specification. Also, the same notation is used for a region type and a region part.

The use of symbols should be uniform; i.e. a symbol should not represent one phenomenon in one context and another in a different context.

The symbols in MEADOW do not represent different phenomenon in different contexts. For example, MEADOW has a uniform way of expressing cardinality, instance names and type names for both parts and connectors.

One should strive for symbolic simplicity.

The symbols of MEADOW essentially consist of a small set of boxes, arrows and labels. Hence we believe that the symbols of MEADOW are fairly simple.

The use of emphasis in the notation should be in accordance with the relative importance of the statements in the model.

Emphasis on notation is not used at all in MEADOW, hence this aspect cannot be evaluated.

Composition of symbols can be made in aesthetically pleasing way.

MEADOW has several concepts for addressing this aspect. (1) Component instances can be grouped into parts, and various concepts for specifying the connection between parts are given. This reduces the number symbols needed to specify component instances and channels. (2) The concepts of composition, generalisation and parameterisation may reduce the number of symbols needed in a specification.

7.1.3 Conclusion

The fact that we found no major deficiencies in MEADOW with respect to the previously listed comprehensibility appropriateness aspects, indicates that the comprehensibility appropriateness of MEADOW is high. As for the conceptual basis of MEADOW, this is perhaps not surprising because MEADOW is for the most part is based on well established, well proved concepts that have existed for a long time. Also, in developing the external representation, we have deliberately tried to use notations that have been used in languages similar to MEADOW. This will make MEADOW easier to understand for people with experience in similar languages.

7.2 Scalability

In the following we evaluate the success criteria *"MEADOW should handle the problem of scalability"* according to the evaluation method described in section 3.5.1.

We describe briefly the concepts of MEADOW that may increase the scalability of specifications, then we relate these concepts to other languages.

7.2.1 Scalability Concepts

Hierarchy is achieved through the concept of composite components which may be decomposed to any depth. Composite components may be seen through a *glass-box* view where the internal structure of the component is shown, or a *black-box* view where the internal structure of the component is abstracted away.

Types, instances and parts. We distinguish between component types and instances of component types. A number of component instances of the same type may be specified in a part, thereby overcoming the need to specify each component instance individually.

Generalisation. The distinction between supertypes and subtypes makes it possible to use parts to specify a number of component instances that may be of different types as long as they share the same supertype.

Parameterisation. Parameterisation of components makes it possible to use parts to specify a number of component instances that may have different internal structures.

MEADOW	FOCUS	SDL	UML
Hierarchy	Composite components	Agent sub-structure	Internal class structure
Component sets	Specification replications	Agent sets	Parts
Generalisation		Suptypes	Generalisation
Parametrised components	Parameterised components	Parameterise agents	Templates
Connectors	Sheafs of channels	Sets of channels	Connectors
Multiplicity on connectors	Arrays of channels		Multiplicities on ports
Constraint functions	Dependent replications		
Cardinality constraints			Multiplicities on connector ends
Identifier constraints			

Table 7.1: Classification of scalability concepts.

Channels and connectors. The concept of connectors makes it possible to specify sets of channels, thereby overcoming the need of specifying every channel individually.

Connector constraints. Identifier constraints, cardinality constraints and function constraints provides scalable techniques of increasing the flexibility of connectors.

7.2.2 Conclusion

Table 7.1 classifies the concepts we have found that may increase scalability in a specification. FOCUS has six concepts, SDL has five, UML has seven and MEADOW has nine, one of which is embodied in MEADOW only (identifier constraints). It is also worth mentioning that MEADOW has the concept of *views*. This concept makes it possible to divide a specification into smaller more manageable parts (views) and to abstract away aspects of a specification that is not related to a view. We have not listed the *views* concept in table 7.1, because it is somewhat different from the other concepts in that it is not directly related to components

or channels.

We have not through-fully studied how well each concept increase the scalability in a specification, hence we have not weighted each concept. For example, the concept of multiplicity on connectors will probably increase scalability in specification more than the concept of identifier constraints. However, we can say that (1) we have found no scalability concept in the other languages that is not included in MEADOW and (2) that MEADOW has more scalability concepts than any of the other languages. Therefore we conclude that MEADOW successfully fulfils the success criteria regarding scalability.

7.3 Generality

In this section we evaluate MEADOW with respect to the success criteria *"MEADOW should handle the problem of generality"*.

7.3.1 Concepts

All of the five topologies that were described regarding the problem of generality are specifiable in MEADOW. In this section we describe the concepts that make this possible with respect to each topology.

The star topology can be seen as a one to many relationship between two parts. This relationship has been specified in numerous examples, see for example figure 5.23.

A specification of a non-uniform tree topology for a depth of three is presented section 6.2, figure 6.4. This topology is possible to specify by virtue of (1) the concept of associating identifiers with parts, which makes it possible to name all component instances in a part and (2) the concept of constraint functions. Neither of these concepts are part of UML or SDL.

We have not given an example of a MEADOW specification of a uniform three topology. But this topology is possible to specify either by using a constraint function, or by using the concept of cardinality constraints in a manner similar to way the concept is used in the UML specification illustrated in figure 4.15 in section 4.3.2.

A specification of a ring topology for multipoint network is presented in figure 6.9, section 6.4. Here we have used component instances of type Node to represent forwarding nodes on the ring. A ring topology

Topology	MEADOW	FOCUS	SDL	UML
Star	Yes	Yes	Yes	Yes
Tree (uniform)	Yes*	Yes*	No	Yes*
Tree (non-uniform)	Yes*	Yes*	No	No
Ring (p-p)	Yes	No	No	No**
Ring (mp)	Yes	No	No	No**
Bus	Yes	No	No	No**

* For a fixed depth.

** Can be underspecified.

Table 7.2: Classification of topology examination results.

for a point to point network could have been specified by putting the hosts on the ring itself, i.e. by removing the part labelled Host[n] and modelling component instances of type Node as hosts.

A specification of a bus topology is presented in figure 6.7, section 6.3. This topology can be specified by virtue of the concept of a merge-split node. Neither FOCUS, SDL or UML has this concept.

7.3.2 Conclusion

Table 7.2 shows a classification of the topology examination results. (The results applicable for the tree topology are for depths greater than two). As can be seen from the table, MEADOW, contrary to the other languages may specify all topologies, and consequently we conclude that MEADOW successfully fulfils the success criteria regarding generality.

7.4 Dynamic Reconfiguration

In the following we evaluate MEADOW with respect to the success criteria "*MEADOW should be able to specify object-oriented, ad hoc, and mobile code networks*".

7.4.1 Concepts

There are essentially two techniques of specifying dynamic reconfiguration in MEADOW. The first technique involves specifying legal *potential* structure of a network, thereby constraining the possible configurations a network may have during computation. In other words a specification in MEADOW may contain information of how a network may change over

MEADOW	FOCUS	SDL	UML
Static/dynamic parts		Cardinality expansion on process sets	
Static/dynamic connectors			

Table 7.3: Concepts for specifying potential reconfiguration.

time. Such specifications are contained in *type diagrams* (see section 5.4.2). Specifically, the concepts of static and dynamic parts/connectors in type diagrams enables a specifier to distinguish between parts of networks that *may not* change over time, and parts of the network that *may* change over time.

The second technique of modelling dynamic reconfiguration in MEADOW, involves specifying snapshot configurations that a network may have at different *points in time*, and defining how these specifications are related in time. See sections 5.4.1 and 5.4.3.

7.4.2 Conclusion

Table 7.3 gives a classification of the concepts we have found that may be used to specify the potential structure of a network. FOCUS and UML has no such concepts, and SDL has one which we have named "cardinality expansion of process sets". The expressibility of this concept is actually weaker than the concept of dynamic parts, because it only constrains the maximum cardinality that a process set may have (see section 4.2.3), whereas the concept of associating a multiplicity with a dynamic part constraints the minimum *and* the maximum cardinality that a part may have (see section 5.2.7). Also, the concept of a static part and a static connector in type diagrams is not included in any of the other languages.

MEADOW makes a clear distinction between *potential* specification (made in type diagrams) of a network and *snapshot* specifications (made in snapshot diagrams) of a network (FOCUS and UML may only specify the latter kind). We have not found this distinction on any of the other languages. Moreover, MEADOW has the concept of relating snapshot diagrams in time, which is not included in any of the other languages either.

As stated previously, we distinguish between three kinds of dynamic networks that we wish to be able to model. Examples of specifications

of these kinds of networks are given in chapter 6. Specifically, a specification of

- an object-oriented network is given in section 6.6,
- a mobile code network is given in section 6.8.

We have also, in section 6.7, specified a network that is similar to an ad hoc network and explained how an ad hoc network may be specified in MEADOW.

In the specification of the mobile code network we have used the concept of message typed connectors (see sections 5.3.7 and 5.3.8) to specify that component instances may be forwarded over channels. This is not in FOCUS, SDL or UML.

Having given examples of specifications of the three kinds of dynamic networks that we wanted to model, and indicated that MEADOW has more concepts for specifying the dynamic reconfiguration of these kinds of networks than FOCUS, SDL or UML, we conclude that the success criteria regarding dynamic reconfiguration is successfully fulfilled.

Chapter 8

Conclusions and Future Work

In this chapter we summarise the results of this thesis, and suggest areas of future work.

8.1 Summary

We have defined a conceptual terminology framework which can be used to describe a language domain. A *network* is defined to be a set of components communicating over channels where components (composite components) may themselves consist of other communicating components (sub components).

We have addressed three problems regarding the use of the traditional dataflow language (TDL) for modelling our language domain: (1) TDL has few scalability concepts, hence specifications of large networks may get space consuming and chaotic. (2) TDL cannot be used to specify networks consisting of a general number of components. We have distinguished between five network topologies consisting of a general components that TDL is unable to specify. (3) TDL has no concepts for specifying dynamic reconfiguration in networks. We have distinguished between three dynamic networks which TDL is unsuited to specify: Object-oriented networks, ad hoc networks and mobile code networks.

We have defined four success criteria that a language must fulfil in order to successfully solve the problems mentioned above. Then we examined and evaluated, with respect to three success criteria, the parts of three state-of-the-art modelling languages (FOCUS, SDL-2000 and UML 2.0) that may be seen an extension of TDL.

We have presented a new language MEADOW, and our hypothesis was that MEADOW fulfils the success criteria, and at the end of this thesis (in

chapter 7) we have provided arguments for its validity.

The main findings with respect to the success criteria are are:

Scalability. Of the three state-of-the-art languages we examined, no single language stood out as being particularly better equipped than the other languages with respect to scalability. Although many concepts of the languages were somewhat similar, there were also some differences. Only FOCUS has the concept of *dependent replications* for example. All the different scalability concepts we found are included in MEADOW. In addition to this we suggested one new concept (*identifier constraints*) that may increase scalability. We have also presented the concept of *views* that provides a way of structuring diagrams and abstracting away information.

Generality. Table 7.2 in section 7.3.2 summarised our topology examination results. MEADOW may specify all of the topologies, FOCUS three, SDL one and UML two[1]. One of the reasons why MEADOW can specify a larger set of topologies than the other languages, is that individual component instances in parts may be named. Relationships between components in a part may therefore be specified precisely by using the concept of function constraints. Also, MEADOW has the concepts of merge and split nodes which none of the other languages have. These concepts are used to specify the bus topology.

Dynamic Reconfiguration. With respect to specifying potential reconfigurations in a network, FOCUS an UML has zero concepts and SDL has one. Two concepts that MEADOW has for specifying potential configurations in a network are listed table 7.3. Additional concepts for specifying dynamic reconfiguration that are not included in FOCUS, SDL or UML were described in sections 7.4.1 and 7.4.2.

Regarding the comprehensibility appropriateness of MEADOW, we have argued that this is high. The conceptual basis of MEADOW is essentially based on well established, well proved concepts. We have tried to base our external representation on notation that has been used in other languages. Also, the comprehensibility appropriateness aspects have not only been used in the evaluation of MEADOW, they have also been considered during development of the external representation of MEADOW.

[1]Most of the topologies may be underspecified in UML

8.2 Future Work

In order for MEADOW to be used for simulation purposes, it must be used in combination with a language for modelling behaviour such as (certain parts of) FOCUS or STATECHARTS for example. Hence, future work on developing constructs for specifying behaviour, or alternatively on how to combine MEADOW with existing such languages, would be interesting. Such a combination would probably result in the identification of generic *operations* than can be associated with components. Examples of such operations are send/receive message, migrate component, send/receive port, create/kill/copy component et cetera.

The constructs for specifying how snapshot diagrams are related in time (section 5.4.1) are rather basic. Further work could therefore be carried out on expanding these constructs, or to combine MEADOW with an existing languages (such as activity diagrams in UML) for these kinds of specifications.

Initial work was carried out on designing a metamodel and implementing a prototype animation tool for MEADOW. However, this was considered too time consuming and focus shifted to the concepts which MEADOW is based on and on how these concepts may be used in case studies. But it would be very interesting to continue the work on the metamodel, and especially the prototype animation tool. The animation tool, as the name suggests, could be used to animate models as well as model checking by comparing MEADOW specification with behaviour specifications (as suggested in section 6.6). The animation tool would then, obviously, have to integrate MEADOW with a language for specifying behaviour. A simple way of doing this, would be to have the animation tool generate a source code shell from a MEADOW specification. A specifier could then modify the code manually (presumably by making use some generic operations) in order to specify component behaviour. This modified code could then be used by the animation tool as a basis for animation and model checking.

MEADOW does not include constructs for explicit specification of communication between a composite component and its sub-components. Further work in this area would also be interesting.

Bibliography

[1] S. H. Bae, S.-J. Lee, W. Su, and M. Gerla. The design, implementaion, and performance evaluation of the on-demand multicast rounting protocol in multihop wireless networks. *IEEE Network*, 14(1):70–77, 2000.

[2] S. Basagni. A mobility-transparent deterministic broadcast mechanism for ad hoc networks. *IEEE/ACM Transactions on Networking*, 7(6):799–807, 1999.

[3] M. Broy and K. Stølen. *Specification and developmnet of interactive systems*. Speringer-Verlag, New York, 2001.

[4] S. Carlsen, J. Krogstie, A. Sølvberg, and O. I. Lindland. Evaluating flexible workflow systems. In J.F. Nunamaker and R.H. Sprague, editors, *Proceedings of the Thirtieth Annual Hawaii International Conference on System Sciences (HICCS'97)*, Volume II Information Systems- Collaboration Systems and Technology, pages 216–232, 1997.

[5] P. Clements, R. Kazman, and M. Klein. *Evaluation software architectures: methods and case studies*. Addison-Wesley, 2001.

[6] J. D. Day and H. Zimmerman. The osi reference model. In *Proc. of IEEE*, volume 71, pages 1334–1340. IEEE Computer Society, 1983.

[7] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on software engineering*, 24(5), May 1998.

[8] J. D. Gibson. *The Mobile communications handbook*. The Electronical engineering handbook series. CRC Press, second edition, 1999.

[9] R. Grosu and K. Stølen. Stream-based specification of mobile systems. *Formal Aspects of Computing*, 13:1–31, 2001.

[10] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8:231–274, 1987.

[11] ITU-T. *Specification and description language (SDL), ITU-T Recommendation Z.100*, 1999.

[12] T. Korson and J. D. McGregor. Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(9):40–90, 1990.

[13] J. Krogstie. Using quality function deployment in software requirements specification. In & E. Dubois A. L. Opdahl, K. Phol, editor, *Proceedings of the Fifth International Workshop on Requirements Engineering: Foundations for Software Quality (REFSQ'99)*, pages 171–185. Heidelberg, Germany, 1999.

[14] J. Krogstie. Evaluating uml using a generic quality framework. SINTEF Telecom and Informatics and IDI, NTNU, 2003.

[15] J. Krogstie and A. Sølvberg. Information systems engineering: Conceptual modeling in a qualitive perspective. Draft of Book, Information Systems Groups, NTNU, Trondhiem, Norway, 2000.

[16] P. Leblanc and I. Ober. Comparative case study in sdl and uml. In *Proc. Technology of Object-Oriented Languages (TOOLS'00)*, pages 120–131. IEEE Computer Society, 2000.

[17] S.-J. Lee, M. Gerla, and C.-K Toh. A simulation study of table-driven and on-demand routing protocols for mobile ad hoc networks. *IEEE Network*, 13(4):48–54, 1999.

[18] J.E. McGrath. *Groups: Interaction and performance*. Prentice-Hall, 1984.

[19] U2 Partners. *Unified Modeling Language: Superstructure version 2.0, 2nd revised submission to OMG RFP ad/00-09-02*, 2003.

[20] L. L. Peterson and B. S. Davie. *Computer networks: a systems approach*. Morgan Kaufman Publishers, second edition, 2000.

[21] S. Pierre and I. Gharbi. A generic object-oriented model for representing computer network topologies. *Advances in engineering software*, 32(2):95–110, January 2001.

[22] I. Satoh. Mobilespaces: A framework for building adaptive distributed applications using a hierarchical mobile agentsystem. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 161–168. IEEE Computer Society, 2000.

[23] I. Satoh. Physical mobility and logical mobility in ubiquitous computing environments. In N. Suri, editor, *Proc. Mobile Agents: 6th International Conference (MA 2002)*, number 2535 in Lecture Notes in Computer Science, pages 186–201. Springer-Verlag, 2002.

[24] H. J. Siegel. *Interconnection networks for large-scale parellel processing*. McGraw-Hill Inc., 1990.

[25] I. Sommerville. *Software Engineering*. Addison-Wesley Publishers Limited, Pearson Education Limited, sixth edition, 2001.

[26] B. Unger, Z. Xiao, J. Cleary, J-J Tsai, and C. Williamson. Parallel shared-memory simulator - performance for large atm networks. *ACM Transactions on Modeling and Computer Simulation*, 10(4):358–391, 2000.

[27] R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30(4), 1998.