UiO : University of Oslo

Farzane Karami

# Language-based Approaches for Enforcing Privacy and Security Policies

**Thesis submitted for the degree of Philosophiae Doctor**

Department of informatics
Faculty of Mathematics and Natural Science

University of Oslo


**2023**

*Series of dissertations submitted to the*
*Faculty of Mathematics and Natural Sciences, University of Oslo*
*No. 2672*

ISSN 1501-7710



Cover: UiO.
Print production: Graphic center, University of Oslo.

# Abstract

In this thesis, we experiment with customizing programming languages to enforce privacy and security policies. We enforce privacy and security requirements at the level of a programming language when a program executes. We design a language and enrich it with the essential features to enforce the requirements. Moreover, we model our language with formal methods and prove that programs written in our language do not violate the desired policies. We design the language's syntax and semantics and formalize the operational semantics with mathematical logic, which enables us to reason about the language's properties.

The structure of this thesis is described in the following. First, we introduce privacy and security policies that we want to enforce. We choose the GDPR (General Data Protection Regulation), which has strict requirements to protect the individual's privacy when processing personal data. For security, we give an overview of existing language-based techniques to preserve confidentiality and limit access to sensitive data. Second, we state the research questions that we want to handle in this thesis. Third, we introduce the tools and logic that we use for our research methods and modeling our languages. Finally, we present the research papers and relate the research questions to our contributions.

The main contributions of this thesis are presented in three research papers. In the first paper, we introduce a programming language with provable guarantees that protects privacy and enforces the GDPR's requirements. The second paper gives an overview of a category of programming languages, called active object languages, that are used to develop distributed systems. In the third paper, we introduce a security mechanism to enforce security in active object languages. We discuss and prove that our language-based approaches are exact when it comes to enforcing policies and restrictions. Moreover, our approaches can be generalized to other languages.

# Acknowledgements

Many thanks to my supervisor Olaf Owe and my co-supervisor Martin Steffen at the university of Oslo for providing guidance along my PhD and always taking time for my questions. I am very grateful to them for our many discussions, where they provided me with constructive suggestions and helpful advice.

Thanks to David Basin at the ETH Zurich University and Einar Broch Johnsen at the University of Oslo for close collaboration on developing a data protection programming language. I am very grateful to them for our many interesting discussions with creative ideas and their help with coauthoring our research paper.

Thanks to Gerardo Schneider at the Chalmers University and Christian Johansen at the University of Oslo for close collaboration during my research visit to Gothenburg in May 2018.

**Farzane Karami**
Oslo, May 2023

# Contents

Part I

# Overview

# Chapter 1

# Motivation

Compliance with the General Data Protection Regulations (the GDPR) [68] is a big challenge for enterprises. The GDPR has very strict data protection requirements and threatens enterprises with significant fines for non-compliance. If an enterprise fails to meet the GDPR requirements for its data processing systems, then the enterprise gets a fine of up to 20 million Euro or 4% of its total annual turnover, whichever is higher will be the maximum fine amount. Modern data processing systems are complex, and it is cumbersome to manually check and verify GDPR compliance for a data processing system.

We investigate this problem from a *programming language perspective*. Programming languages play an important role in the behavior of a system. A program determines the behavior of a system and the underlying language determines the capability to express a program code. Traditional programming languages are not sufficient for enforcing data protection policies. For example, the GDPR requires transparent data processing where data is collected and used for explicit purposes; however, these languages do not have an explicit representation for a purpose. Moreover, under the GDPR, data must be collected and used only for purposes that the data subject has consented. Traditional languages do not have control over data collection, data propagation, and data usage within code. The GDPR has temporal requirements for data deletion; for example, data must be deleted when the user requests for data deletion or when purposes are served. These requirements make programming tricky since data usage must stop if consent is withdrawn or when data deletion arises. Therefore, our aim is to develop a reliable programming language concept that enforces the GDPR's requirements, hence systems developed with our language are GDPR compliant.

We also conduct research to enforce confidentiality in distributed systems by means of language-based techniques. We study the programming languages that are used for developing distributed systems and the existing language-based techniques for enforcing confidentiality at the level of a programming language. Our aim is to introduce a language-based approach that is suitable for enforcing confidentiality in distributed settings. Access control mechanisms are the standard ways to control data access, but they do not fully control the propagation of sensitive data when access is granted to a system. For example, when access is granted to a component, sensitive data might be sent to other components. In a distributed setting, due to the complexity of the system, it is cumbersome to check the trustworthiness of all program codes. By means of language-based techniques, we can analyze programs automatically. In these techniques, a programming language is enriched with the necessary syntax and semantics to track the flow of information during compile time or program

execution. A compile time or runtime error arises if data flow is not compliant with security policies. This way, security policies are enforced at the level of programming languages.

In this thesis, we aim to use language-based techniques to enforce data protection under the GDPR and to enforce data confidentiality in distributed systems. We design a data protection language called DPL where the GDPR's data usage requirements are taken into account in the language design. In the following, we introduce the data usage requirements of the GDPR. Moreover, we propose a language-based approach to enforce data confidentiality in a distributed setting. We define the operational semantics of our languages in a formal and logical framework and reason about the properties of programs written in our language.

## 1.1 Introduction to the GDPR

In 2016, the EU passed the GDPR to protect the privacy of individuals when their personal data is processed by entities. The GDPR is now part of European Union law, and it came into force in May 2018. Failure to comply with the GDPR requirements leads to substantial fines. The GDPR spells out requirements that regulate data processing by enterprises and organizations. Enterprises must make their data processing systems compliant with the GDPR and provide records of their data processing activities to prove GDPR compliance.

The GDPR regulations apply to any processing of personal data. Data is personal if the information can be used directly or indirectly to identify a person; for example, finding a person's location, name, economic status, etc. Any operation on data is considered as processing, including collection, storing, erasing, etc. [82]. The GDPR has many fine-grained requirements. In this thesis, we only consider the GDPR provisions on the data usage as follows:

- **Consent**: The data subject's (the individual whose personal data is being collected and processed) consent is required prior to data collection and processing. The data subject must be aware of the identity of the enterprise that is going to use her data and the intended purposes for which her personal data is collected. If data processing has multiple purposes, consent must be obtained for all of them. For example, entity X runs a social platform and for this purpose, it collects and stores personal data from users. Entity X also sells advertising space on the platform to third parties that can use personal data for marketing. Therefore, when users sign up for X's social media, they have to consent to the use of their personal data for marketing by the third parties [82].

- **Purpose limitation**: Personal data shall be collected for legitimate and explicit purposes, and personal data must not be used for incompatible purposes. It is not allowed to process data for purposes that are incompatible with the purposes for which data is collected. Overall, the GDPR requires transparent data processing, which means that purposes

for data processing must be clear and understandable by a large group of data subjects, and data must only be used for the purposes for which data is collected [82]. For example, if data is collected for a mass-marketing purpose, it cannot be used for targeted marketing. In fact, targeted marketing is another sub-purpose of marketing that requires consent. This requirement increases the level of transparency for data usage.

- **Storage limitation**: Personal data shall be kept no longer than necessary for the processing purposes. The storage period shall have a deadline set to a strict minimum. An enterprise is responsible to delete data when the deadline for storage arrives. Moreover, personal data shall be deleted after the purposes are served [82]. For example, a credit card number is collected for the purpose of a purchase, and if the data subject consents, it can be stored for future purchases. To enforce storage limitation, an enterprise must set up retention policies, specifying how data is used for future use, how long data is needed, and the deadline for the erasure of unnecessary data.

- **Right to withdraw consent**: The data subject has the right to withdraw consent at any time, and an enterprise is no longer allowed to use the data for processing. An enterprise needs to inform the data subject of her right to withdraw before obtaining consent. When a data subject withdraws consent, an enterprise must stop using the data subject's data for the withdrawn purpose without undue delay [82].

- **Right to be forgotten**: The data subject has the right to request to delete her data, and an enterprise is obliged to delete the data without undue delay. For example, entity X wants to recruit new employees for its new business. During the process of recruitment, many applicants are rejected and they receive a rejection notification from X. In this scenario, X processes the personal data of applicants for the purpose of recruitment. Since X does not need the data of applicants who were rejected, those applicants have the right to demand the erasure of their personal data from X [82].

As mentioned before, the GDPR has many fine-grained requirements but in this thesis, we only consider the above requirements, which are named *data usage requirements*.

## 1.2 Introduction to data confidentiality

A common way to restrict access to sensitive data is via access control mechanisms, where access is granted if the request for access is authorized. The problem is that if a program code is authorized to access sensitive data, there is no guarantee that the code handles the data safely. For example, a program might send sensitive data to an unauthorized entity. In order to ensure that the confidentiality of data is preserved within a system, we need to analyze programs

and verify that they are trustworthy to deal with sensitive data. In other words, ensuring that programs are not malicious and are not sending sensitive data to unauthorized entities. In the following, we give a brief introduction to language-based techniques that are used to control the flow of information in programs.

Information-flow-control approaches [33, 71] are used to track the flow of information during compile time or program execution. In static approaches, programs are verified at compile time, and in dynamic approaches, security checks are performed at runtime. Each approach has its advantages and disadvantages and can be well-suited for enforcing a particular policy. Static approaches have zero runtime overhead but dynamic approaches come with the cost of runtime overhead. Dynamic approaches are more permissive [33, 70], i.e., programs that are accepted by a dynamic approach might be rejected by a static one. This is due to the over-approximation in static approaches, where in order to be safe, variables whose their confidentiality is unknown at the time of analysis are considered as confidential. This can lead to unnecessary rejections. In runtime approaches the confidentiality of a variable is evaluated at the time analysis. More details can be found in Section 3.2.

In information-flow-control approaches, program variables are annotated with security tags; for example, *high* and *low*, representing variables that contain sensitive data and the ones that contain nonsensitive data, respectively. Note that unauthorized entities can access low variables but they are not allowed to receive sensitive data or high variables. By annotating methods parameters, we can specify whether a method is expected to receive sensitive data. Similarly, we can annotate return values to specify whether a method returns sensitive data. During program compilation or execution, variables' security tags are propagated along the program, and security checks are performed to restrict the flow of sensitive data according to a policy. A notion of compliance is formalized based on security levels and a security policy. Compile-time or runtime errors arise if the flow of information is not compliant with a policy. For example, the low outputs of a program must not be affected by high variables since it is a leakage of information from high to low. We can design a programming language and incorporate security levels in the language's syntax and security checks in the semantics to enforce policy restrictions.

## 1.3 Introduction to active object languages

Active object languages are based on the combination of the actor model [1] and object-oriented principals [8]. They are inherently concurrent programming languages and are suitable for developing distributed systems. In the actor model, actors are concurrent processes that communicate asynchronously via message passing. An actor encapsulates its fields, procedures, and a single thread of execution. An actor starts handling a message until it completes the task and starts another message. In an actor model, actors do not return results, and this makes programming difficult [8].

In active object languages, an object encapsulates its state and has a dedicated thread for executing processes. Processes can be interleaved, which results in concurrency. Only one process can be active in an object while others are suspended and can be resumed later when the object's thread is available. In active object languages, method calls are asynchronous and objects communicate via message passing. Some active object languages use the *future* mechanism for communication, where an asynchronous call creates a message and a future that is a placeholder for the method's result [14, 28]. Therefore, a caller does not have to be blocked and wait for the callee to finish the method call. Instead the caller can continue with other tasks and fetch the result from the future when it needs the value.

Distributed systems require flexible communication between autonomous and distributed processes. Conventional object-oriented programming languages tightly couple a caller and a callee via synchronous method calls, where a caller waits for the result. In active object languages, callers and callees are loosely coupled via asynchronous message passing and asynchronous method calls. The model of asynchronous message passing is inherently concurrent and suits distributed settings.

## 1.4 Research goal and methodology

It is challenging to find a suitable way to process personal data in conformance with the GDPR. Enterprises need to reprogram their systems, or they need to consider the GDPR requirements when designing their systems. Nowadays the common approach for developing GDPR compliant systems is to consider the GDPR requirements as early as possible when designing and developing a system, which is called *privacy by design* [16]. The question is how much privacy can be achieved by manually designing and taking care of privacy. As discussed in [72], from the design point of view, many levels of abstraction exist in the design model of a system. For example, the components of a system are represented conceptually and without full details, whereas the full description of their integration into bigger components or their communication protocols are abstracted away. Moreover, it is the programmer's responsibility to take into consideration the GDPR requirements when developing a system, which requires deep knowledge in privacy. As Schneider explains [72], privacy by design has its own limitations, and by itself cannot guarantee privacy. There is a general lack of a framework or methodologies for developing privacy compliant systems. The GDPR has teeth, thus an accurate and exact approach is needed to automate the enforcement of these requirements.

Researchers have used information-flow-control approaches to check privacy compliance in programs. The state-of-the-art uses static checking to control purpose-based data usage in programs [32, 47]. For example, a programmer annotates variables with the permitted purposes, methods are also annotated with their purposes. Then at compile time, for a method call, the compliance of the method's purposes is checked against the parameter's permitted purposes. A

compile error arises if a method's purpose is not compliant with the permitted purposes of a parameter. Similarly, the authors in [81] propose a purpose-based and a static type checking approach to enforce privacy compliance in distributed systems and particularly in active object languages. In [80], a dynamic approach is proposed to control data access based on user's consent in active object languages. In this approach, an interface is defined to create a list of consented purposes, where via its methods a data subject can revoke or add consent. Data is tagged with privacy policies determining who the data belongs and program, methods are annotated with purposes. When a tagged data is used for a method, the corresponding list of consent is checked and runtime errors arise if a method uses data for purposes that do not comply with the list of consented purposes.

To enforce the GDPR requirements, we need a dynamic approach, where users' consent determines the permitted purposes. Static checking is not sufficient to enforce the GDPR temporal requirements. For example, when a user withdraws consent, data usage for the withdrawn purpose must stop without *undue delay*. Therefore, consented purposes can change over time, and a dynamic approach is required to precisely restrict data usage. Moreover, the storage limitation requirement requires that data is erased when the data deletion deadline arrives without *undue delay*. Thus a dynamic approach is needed to automatically delete personal data when the deadline arrives or when a user requests for data deletion. The term "undue delay" does not mean that a program instantly aborts and the data is instantly deleted , but rather, as soon as reasonably possible, the system will no longer process the data and it will be removed. For example, when we use the services of Google or Facebook, the expectations for data deletion is in the order of minutes or hours, not seconds. We need to design a programming language that takes into account these tricky aspects of the GDPR and still is reliable and reusable with the GDPR guarantees.

Similarly, it is challenging to preserve data confidentiality in distributed systems, where pieces of code and components can be unknown. Therefore, it is not trivial to analyze all program codes and ensure that they behave safely; i.e, they do not reveal sensitive data to unauthorized components. Information-flow-control approaches [33] are used to track the flow of sensitive data and enforce access restrictions at the level of programming. The state-of-the-art uses static type checking to enforce data confidentiality in active object languages that do not use the future mechanism [61]. Their security approach mainly focuses on other aspects of concurrency such as interleaving processes within an object's thread that might compromise the confidentiality of data. In [2], a dynamic approach is proposed to enforce data confidentiality in the ASP language [14], which is an active object language that uses the future mechanism. In [2], security levels of activities, variables, and futures are fixed and are assigned by the programmer prior to the program execution, but the compliance is checked at runtime when fetching a future value. In this thesis, we are looking for a permissive approach where security levels of variables and futures are evaluated during runtime instead of being assigned and over approximated at compile time. Futures are created upon calls at runtime and their values are calculated at runtime. A future might contain sensitive data if the corresponding method

result is sensitive.

### 1.4.1 Research Goals

Our goals are:

1. designing a programming language that enforces strict policies of the GDPR,

2. proposing a language-based approach that enforces data confidentiality in a distributed system.

These general research goals are subdivided into the following research questions.

- What are language design principles to enforce the GDPR requirements?
- How can we design and formalize a data protection programming language and verify that its semantics ensures compliance with the GDPR requirements?
- What are the communication mechanisms of active object languages? How do their communication paradigms challenge security enforcement in distributed systems?
- How can we design a language-based approach to enforce security in a distributed setting?

In the research paper I, we design a data protection language (called DPL) that enforces the GDPR requirements mentioned in Section 1.1, including consent-management, purpose-based data usage, retention policies, and the data subject's rights for withdrawing consent and data deletion. DPL is object-oriented with interface encapsulation and uses massage passing for object communication. We enrich DPL with language features that are necessary to enforce the GDPR requirements. We formalize DPL's operational semantics in rewriting logic [52] and in Maude [18], which is a logical and computational framework. The Maude implementation provides us with a prototype interpreter for executing DPL programs. DPL's formalization enables us to reason about the behavior of a system and prove that programs written in DPL satisfy the data usage GDPR properties.

In the research paper II, we study active object languages and their communication paradigms. We describe how the future mechanism can challenge the security in distributed settings. In the research paper III, we mainly focus on active object languages that use the future mechanism. We propose a permissive and dynamic approach that enforces confidentiality and is suitable for distributed settings.

### 1.4.2 Methodology

In this thesis, we conduct theoretical research to study the behavior of software systems with respect to privacy and confidentiality policies. In order to formalize a system and analyze its behavior we make use of *formal methods* (which is the main methodology for our theoretical research). In formal methods, the semantics of programming languages can be described in the following styles: 1) *operational semantics*, which describes the meaning of a programming language by means of specifying rules describing how a program executes, 2) *denotational semantics*, which describes the meaning of a programming language in the mathematical theory of domains, 3) *axiomatic semantics*, which describes the meaning of a programming language in terms of preconditions and postconditions which are true before and after the program executes [83]. In this thesis, we mainly use the operational semantics style but also the axiomatic semantics style.

Formal methods are mathematical techniques for specifying, developing, and verifying the robustness and reliability of a software system. Formal methods provide specification languages by which a system can be mathematically and unambiguously described. A specification language (like programming languages) consists of syntax and semantics. The syntax is concerned with symbols and the grammar, i.e., how symbols can be arranged. The semantics consists of rules to define well-formed sentences (built from the syntax), rules to interpret and give meaning to the sentences (semantics), and rules to infer and deduct information (proof theory).

The question is which formal method is well-suited for modeling a programming language? We are looking for an executable and computational logic to specify a language as this allows to experiment and automatically analyze or simulate programs. We formalize the operational semantics of a programming language by means of rewriting logic [52] (which we describe in the following). This results in a logical theory with derivable states and terms where program execution is logical deduction. For the specification of a language's properties, we do not need an executable model. We can specify properties in some form of logic such as first-order logic, higher-order logic, or a temporal logic. We are interested in formalizing a programming language such that it is possible to analyze programs properties. In particular, we want to verify that all programs written in our language are GDPR compliant.

Rewriting logic [52] is a computational logic that can naturally model concurrency and non-determinism in systems. In rewriting logic, a programming language can be formalized as a *rewrite theory*, describing the language's algebraic data types, and the rewrite rules defining the semantics of the language. The *rewrite rules* formalize the system's transitions (program execution) and thus its behavior. The outcome of rewrite rules can be non-deterministic, since there can be several rewrite rules that are applicable to a subterm of a system. Thus, a term can rewrite in many different ways. Based on rewriting logic, we can define an interpreter for a programming language and test and analyze programs' behavior.

Maude [18] is a framework that implements rewriting logic with a high-

performance of millions of rewrites per second. Maude is a tool that supports 1) rewriting logic specifications, 2) execution of the specification, 3) temporal logic to specify systems properties, 4) search-based analysis of a system's state space, 5) model checking, and 6) theorem proving. Maude can search through a system's state space and check if a reachable state satisfies a certain pattern and condition. Moreover, Maude's model checker can verify the temporal properties of a system. By using Maude's model checker we can reason about the behavior of a system over time.

## 1.5 Structure of this thesis

This thesis is written in the form of a cumulative dissertation, which consists of a number of research papers. The thesis has two parts. Part I provides the necessary background and preliminary context for the research papers that are presented in part II.

In Chapter 2, we introduce rewriting logic, Maude, and model checking. As mentioned in Section 1.4.2, these are the methodologies for formalizing and analyzing a programming language or a system model. In Chapter 3, we give an introduction to active object languages and information-flow-control approaches. In Section 3.2, we focus on static and dynamic approaches for tracking the flow of information in programs. Chapter 4 gives a summary of all research papers. Chapter 5 describes the contributions of the research papers, revisits the research questions, and concludes with future work.

Chapter 2

# Preliminaries on formalizing a programming language

In this thesis, we use rewriting logic to formalize our proposed programming language that enforces the GDPR requirements. We also use rewriting logic to formalize a language-based security approach. We model our languages in Maude, which is a framework for rewriting logic. Therefore, in this chapter we give an introduction to rewriting logic and Maude. Moreover, to verify that programs written in our language satisfy the GDPR properties, we make use of Maude's model checker. Thus, we discuss the process of model checking as well.

## 2.1 Rewriting logic

Rewriting logic [52] is a computational logic for modeling concurrent and distributed systems, where the system's state changes by actions taking place simultaneously. Rewriting logic is a semantic framework in which concurrent systems, programming languages, and software can be specified, executed, and analyzed as rewrite theories. Moreover, rewriting logic is a logical framework where automated deduction procedures and various logics such as equational, Horn logic, and linear logic, including quantifiers, can be expressed [54]. Rewriting logic can be used to specify dynamic aspects of computations like how a system evolves over time. Moreover, the semantics of rewriting logic is intrinsically concurrent, thus it can be used to specify concurrent programming languages.

In rewriting logic, rewrite rules are represented in the form $t \to t'$, where $t$ and $t'$ are expressions in a language or subterms of a system. The following describes the two complementary interpretations for a rewrite rule $t \to t'$ [51]:

- computational: the rewrite rule $t \to t'$ represents a transition in a concurrent system, where an instance (subterm) with the pattern $t$ changes to the corresponding instance with the pattern $t'$. The rule specifies how a system evolves over one step of execution by rewriting a term $t$ to $t'$.
- logical: the rewrite rule $t \to t'$ represents an inference rule, where we can infer formulas in the form $t'$ from formulas in the form $t$.

A rewrite rule represents a change in a concurrent system, where each rule specifies a pattern for an action that can occur concurrently with other rules being applied. This way we can reason about the possible changes in a complex system based on possible actions formalized in terms of rules. In rewriting logic, rewrite rules are applied in parallel and independently to non-overlapping

subterms of a system. Moreover, if there are more than one rule that can be applied to a subterm, the order in which they are taken is nondeterministic. This models the non-deterministic behavior of a system [86].

### 2.1.1 Foundations of rewriting logic

A *rewriting logic* specification includes an equational logic specification extended with labeled rewrite rules that describe the transitions of a system [87].

**Definition 2.1.1.** A rewrite theory $\mathcal{R}$ (which is a specification for rewriting logic) is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where $\Sigma$ is a ranked alphabet of function symbols, $E$ is the collection of equations (possibly conditional) defined on variable terms, $L$ is a set of labels, and $R$ is a collection of labeled rewrite rules (possibly conditional) [50].

#### 2.1.1.1 Equational logic

A language's syntax is defined by a signature $\Sigma$, which is a grammar for how to build up valid sentences and ground terms for that language. In rewriting logic, a signature is a pair $(\Sigma, E)$, where $\Sigma$ is a ranked alphabet of function symbols and $E$ is a set of $\Sigma$-equations [50]. The signature of a rewrite theory can be used to describe patterns of a system's states. The notation $\Sigma$ is a collection of functions (constructor operators) for building up the data values or the ground terms of a system. The notation $E$ is a set of conditional or unconditional equations defined on variables. Terms are things that are built from constants (which can be seen as nullary operators), variables, or by applying operators on ground terms. Terms without variables are called ground terms. Based on equational logic, terms $t$ and $t'$ describe the same state if and only if the equation $t = t'$ is derivable from the set of the set of equations $E$. Given a set of equations $E$, $T_{\Sigma,E}$ denotes the equivalence classes of ground $\Sigma$-terms modulo the equations in $E$. Rewriting applies to equivalence classes of terms modulo the equations in $E$. This makes rewriting rules flexible and free from the syntactic constraints of a term representation.

In equational specifications, a reduction step is the application of an equation to a term, which reduces a term $t$ to $u$ denoted as $t \rightsquigarrow u$. For example, given the equation $l = r$, where $l$ matches some subterm of $t$, then the equation is applied to the term $t$, replacing the appropriate instance of $l$ with the corresponding instance of $r$. Zero or more reduction steps are denoted as $\overset{*}{\rightsquigarrow}$.

**Definition 2.1.2** (Termination)**.** An equational logic specification $E$ is terminating if and only if there is no infinite reduction in $E$ [87].

**Definition 2.1.3** (Confluence)**.** An equational specification is confluent, if and only if for all terms $t$, $t_1$, and $t_2$, where $t \overset{*}{\rightsquigarrow} t_1$ and $t \overset{*}{\rightsquigarrow} t_2$, there exists a term $u$, where $t_1 \overset{*}{\rightsquigarrow} u$ and $t_2 \overset{*}{\rightsquigarrow} u$ [87].

The confluence property ensures that the result of a term or expression evaluation is independent of the order of rewrites being applied. In a rewrite

system, it is desirable that the equations when interpreted as rewriting (taken from left to right without reflexivity) are terminating and confluent. However, when applying rewrite rules, rules can be applied in a non-deterministic order or there can be more than one rewrite rule applicable to a (sub-)term, thus the outcome of rewrite rules can be non-confluent.

In equational logic, we can define associativity and commutativity which are two fundamental laws for defining sets and multisets. In a (multi-)set, $\{a, b\} = \{b, a\}$, which is deducted by defining a commutative binary function:

$$f(x, y) = f(y, x)$$

When a function is declared as commutative, then computations are performed on C-equivalent terms, where C refers to commutative equivalent. A binary function is associative if:

$$f(f(x, y), z) = f(x, f(y, z))$$

Defining commutativity and associativity for functions leads to loops and non-terminations. In logical frameworks, there are ways to declare a function as commutative or associative or both and avoid non-termination.

#### 2.1.1.2 Rewrite rules

In a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, $R$ is a set of rewrite rules representing the transitions of a system. A labeled rewrite rule in $R$ has the form $l : t \to t'$, where the label $l$ is the rule's name and $t$ and $t'$ are algebraic expressions in the syntax of $\Sigma$. The left-hand-side $t$ is a firing pattern and the right-hand-side $t'$ is the corresponding replacement pattern. If a subterm of a system state matches with the pattern $t$, then in one step of execution, it is replaced by the corresponding instance of $t'$.

Given a rewrite theory $\mathcal{R}$, the notation $\mathcal{R} \vdash t \to t'$ means that the transition $t \to t'$ is provable in the theory $\mathcal{R}$ using the following inference rules [50, 55] (which are also called deduction rules). Note that to indicate the existing set of variables $X = \{x_1, \ldots, x_n, \ldots\}$ in the terms $t$ and $t'$, we write the abbreviate notation $l : t(\bar{x}) \to t'(\bar{x})$, where $\bar{x}$ denotes a sequence of variables $x_1, \ldots, x_n$. The notation $t(w_1/x_1, \ldots, w_n/x_n)$ denotes that for the term $t$, each occurrence of $x_i$ is replaced by the term $w_i$.

- **Reflexivity**: For each term $t \in T_{\Sigma,E}(X)$, there is an arrow to itself. In fact, for any term $t$ there is a transition where nothing changes.

$$\frac{}{t \to t}$$

- **Equality**: The notation $E \vdash t = u$ means that the equality $t = u$ can be proven by equational logic and the equations in $E$. This rule specifies that states are equivalence classes modulo the equations $E$.

$$\frac{u \to v \quad E \vdash u = u' \quad E \vdash v = v'}{u' \to v'}$$

- **Congruence**: for each function symbol $f$ in $\Sigma$, if $t_1 \to t'_1, \ldots, t_n \to t'_n$ holds, then $f(t_1, \ldots, t_n) \to f(t'_1, \ldots, t'_n)$ also holds. Congruence helps to model concurrency meaning that if $a \to b$ and $c \to d$ hold and can occur in one step, then $f(a, c) \to f(b, d)$ holds and can occur in one concurrent step.

$$\frac{t_1 \to t'_1 \quad \ldots \quad t_n \to t'_n}{f(t_1, \ldots, t_n) \to f(t'_1, \ldots, t'_n)}$$

- **Replacement**: for each rewrite rule in $R$, where $l : t(x_1, \ldots, x_n) \to t'(x_1, \ldots, x_n)$, then we can substitute $\bar{x}$ with $\bar{w}$ where:

$$\frac{w_1 \to w'_1 \quad \ldots \quad w_n \to w'_n}{t(\bar{w}/\bar{x}) \to t'(\bar{w'}/\bar{x})}$$

- **Transitivity**:

$$\frac{t_1 \to t_2 \quad t_2 \to t_3}{t_1 \to t_3}$$

Equational logic (modulo a set of axioms $E$) consists of the above inference rules and one additional rule called the symmetry property in the following.

$$\frac{t_1 \to t_2}{t_2 \to t_1}$$

A rewriting logic specification is terminating if the underlying equational specification is terminating, and there is no term as starting point for an infinite sequence of rewrite steps, where a rewrite rule is applied in an infinite sequence. A rewriting logic is confluent if and only if $t \to t_1$ and $t \to t_2$, then there exists a term $u$ where both $t_1 \to u$ and $t_2 \to u$ hold. In contrast to equational specifications, rewrite rules specifications can be non-terminating or non-confluent, this reflects a non-terminating or non-deterministic dynamic system.

## 2.2 Introduction to Maude

Maude [18] is an implementation for the logical framework of rewriting logic and is used to specify a distributed system or a programming language. In Maude, the properties of a system can be formalized in *linear temporal logic* LTL formulas (which we discuss in Section 2.3.4). Moreover, Maude's model checker can be used to verify that a system's model satisfies the desired properties.

In Maude, a system is specified as a rewrite theory in a modular way. Rewrite rules are defined in system modules, and data types of a language are defined by means of equations in functional modules [10, 52]. Recall the definition of a rewrite theory in Section 2.1.1, a system specification in Maude consists of a functional module for specifying the equational specifications $(\Sigma, E)$, and a system module for specifying rewriting logic rules $(L, R)$. In Maude, data types such as natural numbers, integers, lists, and binary multisets and the

functions on these data types can be defined as equational specifications. The subset relationships between data types are captured in equational specifications by subsorts. The operators are used to create values of the data types. The equations are used to define operations on data types [86].

In Maude, an equation reduces a term or a language expression from the left-hand-side to the right-hand-side of the equation. A term is repeatedly reduced until no further equation can be applied. For example, given an equation $l = r$, a term $t$ reduces by this equation if a subterm of $t$ (or $t$ itself) matches $l$, then the subterm is substituted with the appropriate instance of $r$. Matching is modulo associativity and commutativity (AC-matching) for those operators and equations that are declared as AC. In other words, a term $t$ is considered to be equivalent modulo AC to a term $u$, if we can reduce $t$ to $u$ in zero or more steps by using the associativity and commutativity operations.

Rewrite rules capture the dynamic behavior of a system and describe how a part of the state can change in one step [86]. For a rewrite rule $t \to t'$, if a sub-configuration matches the pattern $t$, then the rule is applied, which changes the sub-configuration to the pattern of $t'$. A system is repeatedly reduced until no further rule can be applied. In Maude, equations have priority over the rules. Thus a sub-configuration's terms are first simplified by equations, then rules can be applied to the sub-configuration.

In the following, we briefly describe the syntax of Maude. Functional modules are introduced by the keyword `fmod` and system modules by the keyword `mod`. The end of modules is introduced by `endfm` and `endm`, respectively. In functional modules, equations are declared with `eq` and `ceq` (for conditional equations). Equations are written with syntax `eq` $t = t'$, and conditional equations are written with `ceq` $t = t'$ `if` $u_1 = v_1 \wedge u_2 = v_2 \ldots$. Equations can be defined with any combination of associativity, commutativity, and identity with the keywords `assoc`, `comm`, and `id` respectively. Rules are defined by the keywords `rl` and `crl` (for conditional rules) in system modules. A labeled rewrite rule is specified as: $l : t \to t'$, where $l$ is the label of the rule, and the rule rewrites an instance with the pattern $t$ to $t'$ in one step. Rewrite rules can be conditional in the form of $l : t \to t'$ **if** *cond*, where the rule is applied if the condition *cond* is true.

Maude supports membership equational logic [53], which allows us to define sorts for terms. For example, $t : s$ states that the term $t$ has the sort $s$. The notation $<$ represents sub-sorting. Maude supports equational attributes, such as associativity, commutativity, and identity, which allow us to define sets, multisets, and lists. In Maude, a system is modeled as a configuration consisting of its components or sub-configurations. The standard way is that a system has the sort *Configuration*, which is a multiset of objects, messages, and other dynamic components [86], and the sub-configurations are sub-sorts of *Configuration*.

In the following, we give a brief description of the Maude definition for a configuration, consisting of objects, classes, and messages. The constructor operators (**`op`**) are used to define the terms of the sort *Configuration*. For example, the first operator *noConf* does not take any argument, and it is a constant, declaring an empty configuration. The second operator, the white

space, takes two terms of sort *Configuration* and creates a multiset. Note that *noConf* is the identity term of the multiset. In Maude, *ctor* denotes a constructor operator.

**sorts** *Configuration Object Class Msg* .
**subsorts** *Object Class Msg* < *Configuration* .
**op** *noConf* : → *Configuration* . [*ctor*].
**op** _ _ : *Configuration Configuration* → *Configuration*
[*ctor assoc comm identity* : *noConf*] .

In Maude, a class *Cl* can be defined with attributes $Att_1$ of sort $A_1$,..., $Att_n$ of sort $A_n$. For example, a class *Cl* is represented as:

$$class\ Cl \mid Att_1 : A_1, \ldots, Att_n : A_n$$

Moreover, an object can be defined as a term with attributes, representing the object's fields, local variables, statements that the object is executing, etc. For example, an object *O* from the class *Cl* has the form structure:

$$< O : Cl \mid Att_1 : Val_1, \ldots,\ Att_n : Val_n >$$

where the object has different attributes $Att_i$ with values $Val_i$ [86]. The constructor operator for an object is represented in the following, where *Oid*, *Cid*, and $A_i$ are the sorts for the object identity, the class identity, and attributes, respectively.

**op** < _ : _ | $Att_1$ : _, ..., $Att_n$ : _ > : *Oid Cid A1 ... An* → *Object* [*ctor*] .

The following example describes an initial configuration for a sender and a receiver object that communicate with messages to send a sequence of values (say natural numbers) [55]. This system has two classes *Sender* and *Receiver*. A sender object has a corresponding receiver that it sends messages to. One can define a message by declaring operators of sort *Msg*. In the following, we define the constructor operations for the classes and messages. Note that sort *Nat* represents natural number and the sort *Oid* is used for object identifiers.

**op** *class Sender* | *cell* : *Nat*, *cnt* : *Nat*, *receiver* : *Oid* [*ctor*] .
**op** *class Receiver* | *cell* : *Nat*, *cnt* : *Nat* [*ctor*] .
**op** *to_* :: *_from_cnt_* : *Oid Nat Oid Nat* → *Msg* [*ctor*] .

For example, *to d* :: *3 from b cnt 1* means that the object *b* sends data item 3 to the object *d*, where the count is 1, indicating the first element that is sent.

Here we define the rewrite rules for sending and receiving a message. In the *send* rule, each time an object of class *Sender* has a data item in its cell, it can send a message to the corresponding receiver, and the cell becomes empty. In this rule, a message is created in the right-hand-side of the rule. A receiver that

has an empty cell can get a message, and the message's data item is stored in the receiver's cell.

$$vars\ Y\ Z\ :\ Oid\ .\ \ vars\ N\ E\ :\ Nat\ .$$

$$\begin{aligned} rl\ [send]\ :\ & < Y : Sender \mid cell : E,\ cnt : N,\ receiver : Z > \\ & => \\ & < Y : Sender \mid cell : empty > \quad (to\ Z :: E\ from\ Y\ cnt\ N)\ . \end{aligned}$$

$$\begin{aligned} rl\ [receive]\ :\ & < Z : Receiver \mid cell : empty,\ cnt : N > \\ & (to\ Z :: E\ from\ Y\ cnt\ N) \\ & => \\ & < Z : Receiver \mid cell : E,\ cnt : N + 1 > \ . \end{aligned}$$

Note that the rules *send* and *receive* represent an asynchronous message passing, which is a communication paradigm in concurrent programming languages. These rules specifications determine how a system that consists of sender and receiver objects can evolve and communicate.

**Maude's model checking.** In addition to a specification and modeling framework, Maude provides LTL model checking, which can be used to automatically verify LTL properties in a system specification. Given that a system's state space is finite, i.e., the set of reachable states from an initial state is finite, Maude can verify whether all possible behavior starting from the initial state satisfies a given property. The result of model checking can be: 1) verification (the given formula holds for all states), 2) refutation (the given formula does not hold and a counter example is produced, 3) the state space, while finite is too big to be explored (or it takes too long time), and a given formula holds for all states up to a given bound [3].

## 2.3 Model checking

In this thesis, we model our designed programming language for the GDPR in Maude. We formalize the operational semantics that gives rise to a transition system, where states are configurations and transitions correspond to rewrite rules application. Moreover we map the GDPR requirements to LTL formulas, which are properties of paths and traces of a transition system. We verify that programs written in our language are GDPR compliant. In the following sections, we discuss transition systems, paths, traces, and LTL formulas.

Model checking [17, 66] is a automatic way to verify that a system satisfies a certain property. System properties specify what a system has to do and what not. For example, a system should never reach a state where no progress can be made (a deadlock state), or a response must be received within a certain time. A system is correct when it satisfies all its specified properties.

In model checking, properties are often specified in propositional temporal logic, and a system is modeled by finite state machines. A model checker examines all (or a portion of) the reachable states and verifies whether a system satisfies a certain property. Verification is the process of searching the state space and examining all possible system scenarios in a systematic manner. Nowadays model checking is able to handle huge states space (due to advances in computer, in algorithms, and technologies) [4]. Model checking complements testing and simulation. Given a model checker tool, the process of model checking has the following phases [4]:

- model the system using the specification language of the model checker ;
- formalize the desired properties in the specification language;
- run the model checking algorithm to verify the properties;
- analysis phase:
  - is the property satisfied? Check the next property;
  - is the property violated? The model checker returns the counter example, then we can correct the model or design, and run the model checker again.

A system can be modeled by using finite-state automata, consisting of a finite set of states and a set of transitions. In a transition system, a state consists of the current values for variables, and transitions describe how the system evolves from one state to another. To precisely specify the properties of a system, *temporal logic* is used. Temporal logic is an extension of an underlying (non-temporal) logic with temporal operators that describe the behavior of a system over time. With temporal logic, we can specify a broad range of properties. For example, correctness (does the system fulfill what it is supposed to do), safety ("something bad never happens"), reachability (does the system reach a certain state?), liveness ("something good will eventually happen"), and fairness (do enabled actions get their turn to execute and not be neglected in favor of other actions that are likewise enabled?).

Model checking has its own strengths and weaknesses. Some of the strengths are that model checking is a general verification approach that can be applied to many ICT systems. It provides counter examples or diagnostic information that can be very useful for debugging. It is run based on a sound mathematical theory such as the theory of graph algorithms, data structures, and logic. The weakness of model checking is that it suffers from *disability issues*, where it cannot be applied to infinite-state systems, which require undecidable logic. It verifies the model of a system not the actual system, and complementary techniques such as testing and simulating are needed to find coding errors or hardware faults. It suffers from the *state-space explosion* issue. It happens that the number of states for modeling a system exceeds the memory capacity. Therefore, realistic systems can be too large for model checking [4].

As mentioned in Section 2.2 Maude supports LTL model checking. In Maude, if the set of initial states is finite and the set of reachable states is also finite, then Maude's model checker can automatically decide where an LTL (linear temporal logic) [65] formula is satisfied in all traces of a transition system.

As mentioned, the operational semantics of our designed language in paper I gives rise to a transition system with infinitely many initial states reflecting infinitely many programs. Therefore, we cannot perform model checking and verify properties for all these programs. A pen and paper proof is required to prove that all programs written in our language satisfies the desired GDPR compliance properties. However, Maude's Model checking can be used to verify properties for a terminating program with a fixed initial state. Note that in paper I, we provide the pen and paper proof.

### 2.3.1 Transition system

The first step for model checking is to model a system with transition systems. Transition systems are used to describe the behavior of systems with graphs consisting of nodes, representing states, and edges, representing transitions. A state gives information about the current behavior of a system. For example, in sequential programming, a state consists of information about the current values of variables and the program counter value that points to the next statement to be executed. Transitions represent how the system evolves by a certain action or execution. For example, in sequential programming, by executing a statement, the state of a system changes, i.e., the value of variables and the program counter.

In transition systems, *action names* such as $\alpha$, $\beta$ are used for transitions, and *atomic propositions* are used for states. Atomic propositions such as $a$, $b$, $c$ are used to formalize the properties of states; for example, an atomic proposition is "$x$ equals 0" for a given variable $x$.

**Definition 2.3.1.** (Transition System) [4]. A transition system is a tuple $(S, \mathit{Actions}, \longrightarrow, I, AP, L)$ where

- $S$ is a set of states
- $\mathit{Actions}$ is a set of actions
- $\longrightarrow \subseteq S \times \mathit{Actions} \times S$ represents transitions
- $I \subseteq S$ is a set of initial states
- $AP$ is a set of atomic propositions
- $L : S \to 2^{AP}$ is a labeling function for transitions

A transition system is finite if $S$, $\mathit{Actions}$, and $AP$ are finite. A transition system starts from an initial state $s_0 \in I$ and evolves based on the transition relations $\longrightarrow$. For example, starting from a state $s$ and non-deterministically choosing the transition $s \xrightarrow{\alpha} s'$, the action $\alpha$ is taken and the state $s$ evolves to the state $s'$. The evolving continues for $s'$ until no further transitions are left for

a state. Note that if a state has more than one transition, the next transition is chosen non-deterministically.

### 2.3.2 Paths and traces

Before explaining temporal logic, we give a brief introduction to the paths and traces of a transition system. A finite path $\hat{\pi}$ consists of a finite state sequence $s_0, s_1, \ldots, s_n$ such that for each $s_i$, where $0 < i \leq n$, there is a reachable state $s_{i-1}$ and a transition from $s_{i-1}$ to $s_i$. An infinite path $\pi$ consists of infinite state sequence $s_0, s_1, \ldots$ such that there is a reachable state $s_{i-1}$ and a transition from $s_{i-1}$ to $s_i$ for $i > 0$. The initial state of a path is denoted as $\mathit{first}(\pi)$, the $j$th state of a path is denoted as $\pi[j] = s_j$, and $\pi[..j]$ denotes the prefix for $j$th state, i.e., $\pi[..j] = s_0, s_1, \ldots, s_j$. Similarly, the notation $\pi[j..]$ denotes the $j$th suffix such that $\pi[j..] = s_j, s_{j+1}, \ldots$.

A trace consists of the sequence of states that are visited and actions that occur, shown as $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \ldots$. For a trace, we can also consider the set of atomic propositions instead of states and actions, i.e., $L(s_0)L(s_1)L(s_2)\ldots$. This way, a trace consists of words over the alphabet $2^{AP}$.

Let $(S, \mathit{Actions}, \longrightarrow, I, AP, L)$ be a transition system. The trace of an infinite path $\pi = s_0\ s_1\ \ldots$ is defined as $\mathit{trace}(\pi) = L(s_0)L(s_1)\ldots$, and for a finite path $\hat{\pi}$, the trace is $\mathit{trace}(\hat{\pi}) = L(s_0)\ L(s_1)\ \ldots\ L(s_n)$. Therefore, a trace consists of infinite or finite words over the alphabet $2^{AP}$, i.e., the atomic propositions that are valid in the states of the path. The function *trace* can be defined over a set of paths $\Pi$ as follows:

$$\mathit{trace}(\Pi) = \{\mathit{trace}(\pi) \quad | \quad \pi \in \Pi\}$$

Let $\mathit{Paths}(s)$ denote the set of paths starting from the state $s$. A trace of a state $s$ is denoted as $\mathit{Traces}(s)$ and is the trace of infinite paths that are starting with the state $s$. Moreover, $\mathit{Traces}(TS)$ is the set of traces of the initial states of the transition system $TS$.

$$\mathit{Traces}(s) = \mathit{trace}(\mathit{Paths}(s)) \qquad \mathit{Traces}(TS) = \bigcup_{s \in I} \mathit{Traces}(s)$$

Figure 2.3.2 is a simple transition system modeling a beverage system. For simplicity, the actions are ignored, and the names of states are equal to atomic propositions, i.e., $L(s) = s$. The total state space is $S = \{\mathit{pay}, \mathit{select}, \mathit{coffee}, \mathit{soda}\}$, and the initial state $I = \{\mathit{pay}\}$. An infinite path $\pi$ can be like $\pi = \mathit{pay}, \mathit{select}, \mathit{coffee}, \mathit{pay}, \mathit{select}, \ldots$.

### 2.3.3 Temporal logic

Temporal logic allows us to reason about the behavior of a system in the future. Figure 2.3.3 shows the semantics of temporal logic. Given atomic propositions $p$ and $q$, basic temporal operators are:

- $\Diamond p$: $p$ holds some time in the future

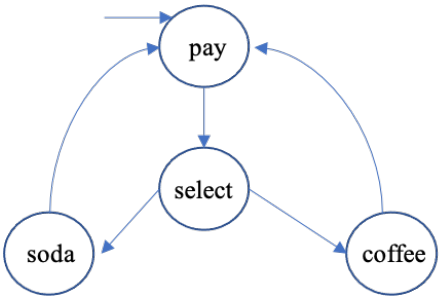


Figure 2.1: Transition system of a beverage machine [4].

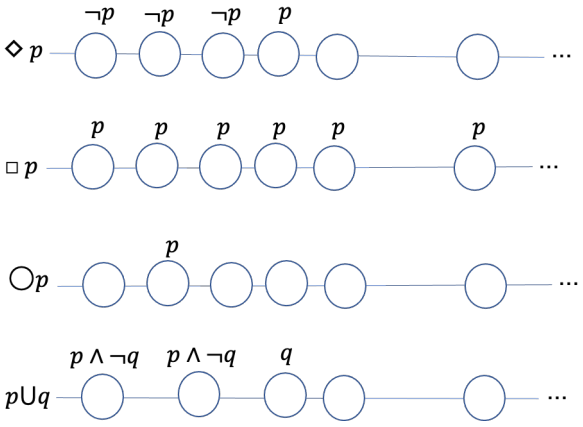


Figure 2.2: Semantics of temporal logic.

- $\Box p$: $p$ holds now and forever in the future
- $\bigcirc p$: $p$ holds in the next moment in time
- $p\ U\ q$: $p$ holds until $q$ holds

Linear-time (LT) properties are defined over traces of a transition system and specify the behavior of a transition system. With LT properties we can specify the set of admissible behavior for a transition system. Given the set of words resulting from the infinite concatenation of words in $AP$, denoted $(2^{AP})^{\omega}$, an LT property is a subset of $(2^{AP})^{\omega}$. We say that a transition system satisfies an LT property $P$, denoted $TS \models P$, iff $Traces(TS) \subseteq P$, and a state $s \models P$ iff $Traces(s) \subseteq P$. In other words, a transition system satisfies an LT property $P$ if all its traces satisfy $P$. A state satisfies $P$ if all traces starting from this state respect $P$.

In this thesis, most of the properties are safety properties. Safety properties are categorized as properties that demand "nothing bad happens". For example,

sensitive data must not be sent to an unauthorized party, or data must not be used without consent from the user. These safety properties are considered *invariants*. Invariants are LT properties that must hold for all reachable states. Some of the GDPR properties (such as when data must no longer be used or must be deleted) are mapped to predicates that must eventually hold and for this to be the case we require a fair transition system.

### 2.3.4 Linear Temporal Logic

In this thesis, the GDPR properties are formulated in the form of LTL properties. In the following, we define LTL syntax and semantics.

LTL is a temporal logic to specify LT properties [4]. We briefly describe the syntax and the semantics of LTL formulas. The following is the syntax for LTL formulas [4, 49]:

$$\phi, \psi ::= \neg\phi \mid \phi \ \wedge \ \psi \mid \phi \ \vee \ \psi \mid \Box\phi \mid \Diamond\phi \mid \ \bigcirc \phi \mid \phi \ U \ \psi \mid p \mid q \mid \ \dots$$

Let $AP = \{p, q, \dots\}$ be a finite set of atomic propositions. LTL formulas are interpreted over linear paths at some position, which means along infinite words over the alphabet $2^{AP}$. For a given path $\pi = A_0, A_1, A_2, \dots \in 2^{AP}$, an LTL formula $\phi$ is true, written as $\pi \models \phi$. The following is the semantics of LTL formulas:

$$\begin{array}{lll}
\pi \models p & \text{iff} & A_0 \models p \quad (\text{i.e., } p \in A_0) \\
\pi \models \Box\phi & \text{iff} & \forall j \geq 0.\ \pi[j \dots] \models \phi \\
\pi \models \Diamond\phi & \text{iff} & \exists j \geq 0.\ \pi[j \dots] \models \phi \\
\pi \models \bigcirc\phi & \text{iff} & \pi[1 \dots] = [A_1, A_2, A_3, \dots] \models \phi \\
\pi \models \phi \ U \ \psi & \text{iff} & \exists j \geq 0.\ \pi[j \dots] \models \psi \quad \text{and} \\
 & & \pi[i \dots] \models \phi, \text{ for every i such that } 0 \leq i < j \\
\pi \models \phi \ W \ \psi & \text{iff} & \pi \models \phi \ U \ \psi, \text{ or } \pi \models \Box\phi \\
\dots & &
\end{array}$$

Here the intuitive meaning of the LTL operators is explained. $\Diamond p$ says that $p$ is eventually true. $\Box p$ is true if $p$ holds forever. By mixing temporal modalities $\Diamond$ and $\Box$, new modalities are derived. For example, $\Box\Diamond p$ ("always eventually $p$") says that it is always the case that at some time in the future $p$ is true. In other words, in a path, at any time $j$, there is a time $i \geq j$ where the property $p$ is true. This ensures that $p$ is true infinitely often. The dual modality $\Diamond\Box p$ ("eventually forever $p$") says that at some time $j$, only $p$ becomes true forever. $\phi \ U \ \psi$ means that $\phi$ is continuously true until $\psi$ becomes true. $\phi \ W \ \psi$ is similar to the $U$ operation except that $\phi$ can always remain true.

#### 2.3.4.1 Semantics of LTL formulas over paths and states

LTL formulas are properties of paths and traces of a transition system. A given path either satisfies a property or not [4]. For simplicity and generality of defining

the LTL semantics, we assume that a transition does not have a terminal state. Thus we can assume that all paths are infinite. Similarly, the LTL semantics can be defined for finite paths. Let $TS = (S, Actions, \longrightarrow, I, AP, L)$ be a transition system and $\phi$ an LTL formula over $AP$. Note that $Words(\phi)$ return all infinite words over the alphabet $2^{AP}$ that satisfies $\phi$.

- For an infinite path $\pi$ of $TS$, the satisfaction relation is defined as:

$$\pi \models \phi \qquad \textit{iff} \qquad trace(\pi) \models \phi$$

- For a state $s \in S$, the satisfaction relation is defined as, where $Paths(s)$ returns a set of paths starting from $s$:

$$s \models \phi \qquad \textit{iff} \qquad \forall \pi \in Paths(s).\ \pi \models \phi$$

- $TS$ satisfies $\phi$ ($TS \models \phi$), if $Traces(TS) \subseteq Words(\phi)$

The last definition is extended as:

$$\begin{aligned} & TS \models \phi \\ \textit{iff} \quad & \\ & Traces(TS) \subseteq Words(\phi) \\ \textit{iff} \quad & \\ & \pi \models \phi \quad \forall \pi \in Paths(TS) \\ \textit{iff} \quad & \\ & s_0 \models \phi \quad \forall s_0 \in I \end{aligned}$$

Therefore, a transition system satisfies an LTL formula $\phi$ if all its initial states satisfy the formula, i.e., $s_0 \models \phi$ for all $s_0 \in I$. This requires that all paths starting from the initial states also satisfy the formula. Correspondingly, it requires that all traces of those paths satisfy the formula.

#### 2.3.4.2 Fairness in LTL

In this thesis, some of the GDPR properties are formalized in the from of predicates that must eventually hold. For this to be the case, we require a fair transition system. In DPL's rewrite rules specification, there are rules that can fire at anytime, representing the non-deterministic behavior of a user. These rules can starve other enabled rules from being fired. Thus we assume strong fairness for DPL's transition system.

Fairness assumptions are necessary to ensure that all enabled transitions are executed, and that there is not an infinite transition that makes other transitions starve. In model checking a transition system that involves non-determinism, it is vital to assume fairness. For example, if transitions can be interleaved and the choice of the next transition is arbitrary, then fairness ensures that a transition is not consistently ignored. With fairness assumptions, we can be sure that all possible transitions are given the chance to actually occur. In general, fairness is characterized by the following definitions [4]:

- Unconditional fairness: "Every transition gets its turn infinitely often to execute."
- Strong fairness: "Every transition that is enabled infinitely often gets its turn infinitely often to execute."
- Weak fairness: "Every transition that is continuously enabled from a certain time gets its turn infinitely often to execute."

Note that the term "is enabled" means that a transition is ready to execute, and the term "gets its turn" stands for the execution of a transition. Fairness for a transition system is formalized by LTL formulas $\phi$ and $\psi$ as follows [4].

- Unconditional fairness:
$$ufair = \Box\Diamond\phi$$
- Strong fairness:
$$sfair = \Box\Diamond\phi \longrightarrow \Box\Diamond\psi$$
- Weak fairness:
$$wfair = \Diamond\Box\phi \longrightarrow \Box\Diamond\psi$$

In paper I, in order to verify that programs written in DPL satisfy the GDPR data usage properties, we assume strong fairness for DPL's transition system that results from the operational semantics.

Chapter 3

# Distributed systems and information-flow-control approaches

## 3.1 Active object languages

Active object languages are suitable programming languages for developing distributed systems. They extend the actor model [1, 5] with asynchronous method calls and the future mechanism for synchronization and scheduling the retrieval of methods' results. An active object is a single-threaded distributed entity that communicates with other active objects via asynchronous message passing. Asynchronous method calls decouple the sender and receiver objects and reduce the risk of deadlock, where the control in one object is not blocked while waiting for a method result. However, the asynchronous message passing leads to the non-deterministic behavior of a system since an object can receive messages in different orders. Non-determinism is a natural behavior in distributed systems.

An asynchronous call in an object creates a future and a message that is sent to the callee. The caller object continues its process and do not wait for the method result, and the callee starts a new process when it receives the message. When the caller needs the method's result, it synchronizes with the created future and performs a get operation on the future. When the callee finishes the process, it returns the result in the future, and the future becomes resolved.

To explain the future mechanism we use ABS [42], which is an actor-based and object-oriented language, as our core language. For example, $f = o!m()$ is an asynchronous call where method $m$ of object $o$ is called, and the future reference is assigned to $f$. The caller can suspend a process while waiting for the callee to finish the call by **`await`** $f?$. Suspension means that the object can suspend the current process in its queue, and pull another process from the queue or stay idle while waiting for the future. Moreover, an object can read the future by $f$.**`get`** which blocks the object until the future contains the result.

Most active object languages use the future mechanism to avoid blocking calls and have synchronization at the same time. A future reference can be passed to active objects, and any object that has a reference to a future can get the result. This raises the question of how to enforce the confidentiality of data when the future mechanism is used. In a distributed system, a future can contain the method's result from a component that is unknown at the time of analysis. In this thesis, we are looking for a precise and permissive approach to control access to futures in distributed systems.

## 3.2 Information-flow-control approaches

Information-flow-control is a technique to track the flow of information and prevent leakage of sensitive information during program execution. In this technique, one defines a set of rules that programs must conform to, and these rules can be enforced at compile time or at runtime during program execution. Based on restriction rules, programs only allow acceptable flows of information, otherwise, they throw compile-time or runtime errors. Information-flow-control approaches are divided into two categories: static and dynamic approaches. A static approach is used to certify programs for confidentiality at compile time. A dynamic approach is used to enforce confidentiality during program execution.

### 3.2.1 Static information-flow-control

In static information-flow-control, we extend type systems with security labels and typing rules to prevent illegal flows leading to the leakage of sensitive information. Program variables are annotated with security labels, where a variable's label determines how information stored in the variable can be propagated. If the content of a variable affects another variable it is called a flow of information. By annotating variables, every expression has a label that consists of two parts: a basic type such as `int` and a security label which can be high ($H$) or low ($L$). By means of type checking, it is ensured that security labels of variables are at least as restrictive as the labels of values that they might contain at runtime. Moreover, the outputs of a program can be high or low, where low outputs are observable by an attacker or any unauthorized entity, and high outputs are only observable by authorized entities. In terms of active object languages, objects due to object encapsulations, objects fields are not accessible from outside but methods can be accessed. Therefore, the result of a method is an output, and objects can send sensitive data to each other through method calls. For example an object can access sensitive data though method calls parameters or getting a return value. Moreover, objects can be assigned high or low security levels indicating if they are authorized or not.

We define a partial order relation $\sqsubseteq$ between security labels, where $L \sqsubseteq H$. Security labels form a lattice [21], where the least restrictive label $\bot$ can flow anywhere, and the most restrictive label $\top$ cannot flow anywhere less restrictive. For simplicity, we consider only two security labels $H$ and $L$. Thus, we have a two-point lattice $L \sqsubseteq H$. In computing security labels of variables, we use the join operator $\sqcup$, which returns the least upper bound between two labels.

In a strict approach (called flow-insensitive), we do not allow a flow from high to low. For example, in the assignment $l := h$, $h$ is a variable with high security label and $l$ has a low label, thus the assignment is an illegal flow, which rises a compile time error. In a more permissive approach (called flow-sensitive), instead of rejecting a program due to an illegal flow, we update the security labels of variables if they are affected by variables with different security labels. For example, in the assignment $l := h$, after the assignment, the security label of $l$ becomes $H$. However, to avoid sensitive information ending up at the hands of

an illegal agent, we set the rules to check the security labels when variables are sent out at the output points; for example, when returning a method's result. In fact, we do not allow sensitive information with a high security label to go to low outputs, which are observable to unauthorized entities or an attacker.

Sensitive information can also leak through implicit flows in conditional structures. For example, in the code $x := 0$; **if** $b = 1$ **then** $x := 1$;, given that $x$ is low and $b$ is high, there is an implicit leakage of information from $b$ to $x$. Therefore, the conditional construct leads to an illegal flow from high to low. Again, in a strict approach, one can reject the program due to the implicit flow. In a more permissive approach, we update the security label of $x$ after $x := 1$. In order to track implicit flows, we use a program counter label *pc*, which becomes high when there is a conditional construct with a high condition. For example, in the code, since $b$ is high *pc* becomes high. Then any expression's security label is computed with respect to *pc*. For example, in the conditional code, *pc* is high since the condition $b$ is high. In the assignment $x := 1$ since 1 is a constant with a low security label and *pc* is high, the security label of $x$ becomes $L \sqcup pc = H$.

We can extend a type system with typing rules that check the security label of variables in each expression. The typing rules can be defined in flow-insensitive way such as the one proposed in [71]. Typing rules can be more permissive in flow-sensitive way as the ones proposed in [61, 70].

### 3.2.2 Dynamic vs. static approach

All static checks can be done at runtime with the expense of runtime overhead. However, a static approach does not have any runtime overhead. A dynamic approach is more permissive than a static one as discussed in [33, 70]. In order to be safe, a static analysis over-approximates security levels of variables that are unknown at the time of analysis to high, which can be low at run-time when the program executes. Therefore, due to overapproximation, a static analysis might reject a program, which can be safe. While, a dynamic approach deals with the run-time and real security levels of variables, which makes it more precise. Here is another example, showing that a dynamic approach is more permissive: **if** $h$ **then return**$(l)$; **else** $h := 0$;, a static analysis rejects the program because there is a leakage in the first branch, while a dynamic approach accepts the program if $h$ is false.

### 3.2.3 Noninterference

**Attacker model** We assume that an attacker is a low level object that has access to public and observable outputs of a program. For example, an attackers has access to a low level return value. Moreover, the attacker can change the value of low inputs.

The goal of information-flow-control is to ensure noninterference where public outputs of a program are independent of secret inputs. Therefore, the attacker is not able to infer the secret inputs from the public outputs. A deterministic program satisfies noninterference, if in two executions of the program, by changing

secret inputs and the same public inputs, the public outputs remain the same. Note that low equivalence captures the fact that an attackers is only able to observe things that have low security levels, hence changes in high security things are not observable. Low equivalent runs means program runs that have equal values for low variables but variables with high security levels can be different. For example, in active object languages, in two program runs, by changing the values of high parameters of a method, while low variables have the same values, the method returns low equivalent results or if the return value is high, then a low-level object is not allowed to access high variables.

In non-deterministic systems, the definition of non-interference is more complex, for example, due to the fact that messages can arrive in different order or processes can be non-deterministically interleaved within an object. Therefore, when comparing two runs for non-interference, one can consider the sequence of input and output (such as method calls and return values), where the histories are low equal [61].

There are 4 variants of noninterference: 1) termination-insensitive, 2) termination-sensitive, 3) progress-insensitive, and 4) progress-sensitive. We briefly explain these variants. The definitions are defined over two low equivalent runs of a program. In termination-insensitive noninterference (TINI), we assume that the divergence of a program is indistinguishable to an attacker. Thus, termination-insensitive noninterference makes security guarantees under the assumption that a program always terminates. If either of program runs diverges, nothing is demanded from the non-interference guarantee, but if both runs terminate, then public outputs must be low equivalent. In termination-sensitive noninterference (TSNI), if one run terminates, then the other run must also terminate and they must be low equivalent [33, 46]. In this work, we mainly focus on termination-insensitive and progress insensitive since it is the simplest form of noninterference.

If an attacker is able to inspect the intermediate steps of an execution, then TINI and TSNI are not enough to guarantee security since the leakage of information can be hidden due to non-termination. For example, a program that sends secret data to a low output prior to a non-terminating loop is verified as secure by TINI or TSNI since the program does not terminate. In this case, progress-insensitive or progress-sensitive noninterference is needed to guarantee security when an attacker is able to inspect the intermediate steps of execution. In progress-insensitive noninterference (PINI), both equivalent runs remain low equivalent, or they have low equivalent outputs prior to the step where one of them diverges. In progress-sensitive noninterference (PSNI), both low equivalent runs terminate and remain low equivalent in every step or they both diverge [33].

# Chapter 4

# Summary of Papers

The research contributions of this thesis are presented in three papers, which are shortly summarized in this chapter. The full version of the papers can be found in part II, where the content is mostly similar to the published version. The papers are presented from the most recent to the oldest. The first paper is independent of the others, while the second paper gives an introduction and opens a research question that is addressed in the third one. The authors of each paper are listed in the same order as the published papers. Note that the papers' format is changed to fit the structure of this thesis.

In the first paper, we design a programming language, called DPL, with formal guarantees for developing GDPR compliant systems. An introduction to the GDPR requirements and the fact that enterprises are obliged to comply with these requirements are described in Section 1.1. The rewriting logic that is the basis of our formal definitions is described in Section 2.1. We simulate DPL in Maude and use Maude's model checker to prove our claims that GDPR properties hold in DPL programs. An introduction to Maude, model checking, and Maude's model checker is provided in Section 2.2 and 2.3.

The second paper gives an overview of active object languages and the future mechanism that they use for asynchronous communication. The paper discusses the challenges to enforce confidentiality when futures are used in a distributed setting. A brief introduction to active object languages and the future mechanism are given in Section 3.1. The third paper addresses the security issues that arise when futures are used in active object languages. In the third paper, we design a language-based mechanism that protects the confidentiality of futures. To define the formal semantics of our security mechanism we use the rewriting logic. Our security mechanism uses information-flow-control approaches, which are described in Section 3.2.

## 4.1 Paper 1: DPL A Language for GDPR Enforcement

**Authors** Farzane Karami, David Basin, Einar Broch Johnsen

**Publication** 35th IEEE Computer Security Foundations Symposium (CSF 2022), pp 112–129

**Summary** The strict requirements of the GDPR raise the question of how to develop a GDPR compliant system that enforces these requirements. In this paper, we tackle this problem from a programming language perspective and design a language that takes care of the GDPR requirements. We call our language DPL, a data protection language. To the best of our knowledge, DPL is the first programming language designed for developing programs that comply with the GDPR requirements such as consent, purpose-limitation, storage-limitation, and the right to withdraw consent, and the right to be forgotten. The state-of-the-art mainly focuses on purpose-based access control, where runtime errors arise if a method accesses data that is not compliant with consented purposes by a data subject [80]. DPL enforces more strict policies of the GDPR such as data consent (where consent is required prior to data collection), data storage, and the right to be forgotten. Moreover, DPL syntax is customized for the GDPR and is enriched with privacy-relevant constructs that help a programmer to write error free codes. We also provide formal proof that DPL programs satisfy the GDPR properties.

In this paper, we investigate language design principles that are essential to enforce the GDPR requirements. DPL has an explicit representation for purpose, where a purpose is declared as an interface and its methods are used to achieve the purpose. In DPL, privacy policies are runtime elements that are attached to collected data and restrict the data usage. Due to the temporal requirements of the GDPR, privacy policies can change over time. For example, data usage for a purpose becomes non-compliant if the user withdraws consent, the user requests for data deletion, or the deadline for data deletion arrives. Therefore, we design a dynamic approach where policies change over time based on users' actions and also the passage of time. DPL's design involves 1) the syntax for creating objects' databases for data storage and retrieval, 2) interface encapsulation where interfaces represent purposes and their methods are used to achieve the purposes, 3) the syntax for data collection, opting-in for granting consent, opting-out for revoking consent, 4) privacy policies that determine to whom data belongs, for which purposes data can be used, and when data must be deleted, and 5) runtime checking to enforce that processes can only access and use data if their purpose is compliant with the data's policy. In DPL, failure to comply does not lead to privacy violations instead it raises runtime errors. We enrich DPL with conditional constructs to perform privacy-relevant checks and avoid actions that lead to runtime errors. This enables a programmer to create error-free and GDPR compliant codes.

We formalize DPL's operational semantics in rewriting logic and also specify the semantics in Maude, which provides us with a prototype interpreter for

executing DPL programs. We formalize the GDPR data usage requirements in LTL formulas and use Maude's model checker for a program example. Maude's model checker verifies that our case study satisfies the GDPR data usage requirements. DPL's operational semantics gives rise to a transition system with infinitely many initial states reflecting infinitely many programs. In order to prove our GDPR claims for infinitely many programs, we provide a pen-and-paper proof and prove that DPL properties hold for all traces of DPL's transition system.

## 4.2 Paper 2: An Evaluation of Interaction Paradigms for Active Objects

**Authors** Farzane Karami, Olaf Owe, Toktam Ramezanifarkhani

**Publication** Journal of Logical and Algebraic Methods in Programming Volume 103, pp 154-183, 2019

**Summary** Active object languages are programming languages that are used to develop distributed systems. These languages have adopted the actor model and object-oriented concepts. In the actor model, concurrent entities communicate via asynchronous message passing, where no data structure is shared among the entities. In active object languages, the autonomous entities are active objects and communication is via asynchronous method calls.

In this paper, we give an overview of the communication mechanisms in active object languages. We mainly focus on a number of active object languages, including ABCL, Rebeca, Creol, ABS, Encore, and ASP/ProActive. The future mechanism is a central communication mechanism in these languages. A future is created as a result of an asynchronous method call, and it eventually holds the method's return value. It is a flexible and non-blocking way of sharing methods' results. Each language has adopted a different strategy for creating, scheduling, and fetching a future. We compare the various future mechanisms with respect to the following criteria:

- expressiveness
- efficiency
- syntactic and semantic complexity
- simplicity of program reasoning and static analysis
- information security aspects.

We discuss the pros and cons of the various future mechanisms. Moreover, we suggest some language improvements in the setting of asynchronous call/return without the use of futures. We provide a unified syntax and semantics for languages' future mechanisms. We compare the program reasoning in languages with and without the future mechanism. We conclude that specifications and invariants verification is more indirect in languages with futures than in future-free ones. We show that verification conditions become more complex for languages with futures when a condition depends on a future get. Moreover, we briefly explain the security issues for programs with futures and discuss the challanges for enforcing confidentiality in these programs.

## 4.3 Paper 3: Information-Flow-Control by means of Security Wrappers for Active Object Languages with Futures

**Authors** Farzane Karami, Olaf Owe, Gerardo Schneider

**Publication** Nordic Conference on Secure IT Systems (NordSec 2020), pp 74–91

The following summary is similar to the version of our paper's overview that was presented at Nordic Workshop on Programming (NWPT 2018).

**Summary** As described in paper 2, future variables give a level of indirectness, where the retrieval of a method's result is no longer syntactically connected to the method call, compared to future free languages. For instance, when a future is received as a parameter, it may not statically correspond to a unique call statement, and different call statements may have different security levels. One may overestimate the security level of a future by considering the set of call statements that correspond to the future, but it requires access to the whole program. In this case, a static approach becomes imprecise and causes unnecessary program rejections, especially when the complete program is not statically known, which can be the case in distributed systems. In addition, the future concept comes with a notion of future identity, but not a notion of the associated caller, callee. These identities could in principle be incorporated into the future identity, but only at run-time. At static time there is no information about the caller and the creator of a future.

The state-of-the-art proposes a dynamic approach to enforce data confidentiality in the ASP [14] language which uses the future mechanism [2]. In [2], security levels of activities and variables are fixed and are assigned by the programmer, but the compliance is checked at runtime when requesting a future value. Our approach is flow-sensitive, where the security level of variables can change, which makes our approach more precise and permissive. We use "security wrappers", where futures and objects are wrapped, and wrappers perform dynamic checking. A wrapper controls an object's communication and the future access. For example, when a low level object attempts to access a high sensitive future, the access is rejected because of incompatible security levels. The idea of wrappers is a permissive and precise approach due to runtime checking and flow-sensitive information-flow-control. We provide the operational semantics for a basic active object language with futures and enrich it with runtime wrappers and flow-sensitivity. We guarantee that an object will be given access only to the information that it is allowed to access.

Our approach comes with the price of runtime overhead. In order to limit this drawback, we combine it with static analysis proposed in [61]. We statically identify objects that deal with sensitive data and only wrap those objects to control their communication. Moreover, the runtime system only wraps futures that contain sensitive data.

# Chapter 5

# Discussion and Conclusion

In this chapter, we present the contributions of this thesis and relate them to the research questions stated in Chapter 1.4.

## 5.1 Summary of Contributions

In this thesis, we experiment with designing custom languages to enforce strict policies such as the GDPR and confidentiality in distributed systems. In paper I, we design an object-oriented language to enforce the GDPR's data usage requirements correctly and exactly. We call our language DPL, a data protection language, and to the best of our knowledge it is the first language designed for developing GDPR compliant programs. We formalize DPL's operational semantics in rewriting logic and prove that DPL programs are GDPR compliant. DPL has the essential features for enforcing the GDPR requirements. GDPR has temporal requirements such as the right to withdraw consent, request for data deletion, or retention deadlines. In DPL, we use a runtime approach to enforce these temporal requirements. Our runtime approach allows us to enforce these requirements exactly. For example, if a user withdraws consent, then data is no longer used for that purpose, which is the GDPR's purpose limitation requirement. DPL provides conditional constructs to perform privacy-related checks prior to data-usage actions such as data collection, method calls, assignments, etc. These privacy checks help to write hygienic programs and avoid runtime errors. Note that a non-hygienic program does not lead to privacy violations but rather runtime errors. We specify DPL's operational semantics in Maude, which provides us with an executable formal model for DPL. We made a case study of an online retailer in Maude, and perform Maude's model checker to verify the GDPR's properties. Maude verifies that GDPR violations cannot occur in DPL programs. Moreover, we provide pen and paper proof for our claims.

In paper II, we give a review of active object languages and their communication mechanisms. We discuss the asynchronous method calls and the future mechanism, which is a placeholder for the result of an asynchronous call. Asynchronous calls and the future mechanism are widely used in active object languages to avoid blocking calls. For example, an object calls a method, creates a future for the result, and continues with other processes while waiting for the result. An object that has a reference to a future can easily access the value when the future is resolved. This gives rise to a security issue when a future contains confidential data and unauthorized objects access the future. We discuss the dynamic nature of the future mechanism and the security issues when futures exist in a distributed system.

In paper III, we introduce a security mechanism to enforce security for futures in distributed systems. We enforce information-flow security in active object languages that support the future mechanism. We track the flow of confidential data by dynamic approach. We define a language primitive called wrapper to monitor the flow of data at runtime. A wrapper wraps an object or a future and controls the security levels of communicated messages. A wrapper prevents sending secret data to low level objects or unauthorized objects. We extend the operational semantics of Creol, which is an active object language, with wrappers and information flow security. The operational semantics is defined by means of rewriting logic. Since runtime checking has runtime overhead, we combine our approach with a pre-existing static checking for Creol, where if an object is statically checked and does not deal with sensitive data, then it does not need a wrapper at runtime.

## 5.2 Discussion of research questions

The goals of this thesis are:

1. designing a custom-based programming language to enforce strict policies of the GDPR,

2. proposing a security approach to enforce data confidentiality in a distributed setting.

We approach the first goal by formalizing our designed language for the GDPR and proving that programs written in our language satisfy the GDPR requirements. For the second goal, for enforcing data confidentiality, we make use of information-flow-control approaches and design a security approach that is well-suited for distributed systems. In order to precisely state the purpose of this thesis we define four research questions. Each question is addressed by the research papers, presented in part II.

- What are language design principles to enforce the GDPR requirements?

In paper I, we describe the GDPR requirements from a programming language perspective. We design a data protection language called DPL, where for each requirement, DPL is enriched with the necessary syntax and semantics. We use a runtime approach to enforce temporal requirements of the GDPR, where the behavior of a system must remain GDPR compliant while a user can request for data deletion or withdraw consent at any time.

In DPL, each interface represents one specific purpose, and the methods of such an interface are used ot achieve that particular purpose. We define constructs to create and manipulate privacy policies including **`policy`** to create privacy policies, **`opt-in`** for granting consent, and **`collect`** for data collection. In DPL, users actions for withdrawing consent and data deletion are captured by rewrite rules without any premise, thus they can fire at any time. A policy describes to whom data belongs, for which purposes data can be used, and

when data must be deleted. In DPL, policy compliance is checked prior to data usage, and if data usage is not compliant with the policy, a runtime error arises. Moreover, to enable a programmer to write error-free DPL programs, we define conditional constructs for privacy-relevant checks and compliance scopes, where compliance is checked prior to data usage and remains compliant within the scope.

- How can we design and formalize a data protection programming language and verify that its semantics ensures compliance with the GDPR requirements?

Maude is a common framework to model and formalize a programming language. In paper I, we design a data protection language, called DPL, that enforces GDPR data usage requirements. We formalize the operational semantics of DPL in the form of Maude rewrite rules. Maude naturally captures non-determinism, which enables us to model users' non-deterministic actions for data deletion and withdrawing consent. We capture these users' actions with rules that do not have axioms, thus they can fire at any time. We also map the GDPR data usage requirements to LTL formulas and use Maude's model checker to verify that programs written in DPL satisfy the GDPR properties.

We make a case study in DPL, which is an online retailing example, where users' personal data are used for the purposes of purchasing, marketing, and targeted-marketing. We simulate and analyze this case study program in DPL's Maude model. We formalize the GDPR requirements with LTL formulas and use Maude's model checker to verify that our case study program satisfies the GDPR properties. Moreover, we define an executable formal model for DPL, given by SOS-style operational semantics. This enables us to provide a pen and paper proof that all programs written in DPL satisfy the GDPR requirements.

Overall, DPL is a programming language designed for developing programs that comply to GDPR data usage requirements. DPL is user-centric, where users' consent and requests are reflected in policies, and it provides exact enforcement of privacy policies. Moreover, DPL enforces richer policies (in particular, temporal requirements of the GDPR) than the static approaches. DPL is an object-oriented programming language, and its core is inspired by ABS, an active object language. We believe that our approach can be carried out for other Java-like languages. We formalize DPL's operational semantics in rewriting logic and provide a pen and paper proof that GDPR violations cannot occur in programs written in DPL. We also formalize DPL's operational semantics in Maude and use Maude's model checker with model-checked examples to support our claims.

- What are the communication mechanisms of active object languages? How do their communication paradigms challenge security enforcement in distributed systems?

In paper II, we give an introduction to active object languages which are used for developing distributed systems. We mainly focus on their communication paradigms including the future mechanism. We discuss how the future mechanism

can compromise the confidentially of data when a future contains sensitive data, and objects with the future reference can access the sensitive data. The future mechanism challenges the enforcement of security in active object languages. We also discuss what type of a language-based security approach can be suitable for active object languages with futures.

- How can we design a language-based approach to enforce security in a distributed setting?

In paper III, we introduce a security mechanism based on wrappers, where futures and objects are wrapped and their interactions are controlled by their wrappers. We use a basic active object language and enrich the operational semantics with wrappers and information-flow tracking. We guarantee that objects can only have access to the information that they are allowed to access. This compliance is checked at runtime precisely based on the security levels of variables, futures, and objects.

## 5.3 Limitation and future work

Here, we discuss the topics that can be further investigated in this research thesis. The GDPR has many requirements, and in this thesis, we only consider the data usage requirements described in Section 1.1. One can investigate other aspects and requirements of the GDPR from a language-based perspective. For example, handling the GDPR requirements when data belongs to multiple data subjects. This can be tricky since if one data subject decides to delete her data while the others do not. Moreover, there are few research on enforcing the GDPR requirements in distributed settings or investigating how can parallel and concurrent paradigms can challenge the GDPR enforcement in language based approaches? Therefore, we expect it to be an interesting line to extend our research. Moreover, the research paper I can use a larger case study to better assess DPL's usability and runtime overhead. For a stronger correctness result, the pen and paper proof in paper I can be formalized in a theorem prover like Coq or Isabelle. In DPL compliance scopes are defined by conditional constructs that perform privacy-relevant checks to avoid runtime errors. DPL can be enriched with a type checking system to effectively enforce the correct use of compliance scopes. For example, runtime errors can arise due to non-compliant data usage in method calls, and in order to avoid runtime errors, method calls must be in appropriate conditional constructs. Conditional constructs could make programming in DPL easier, where well-typed programs are error free. Other possibilities for future work could be extending DPL with binary and general operations, where an operation is carried out on data with two different policies from two different data subjects. In this case, one needs to consider the GDPR compliance in scenarios where two policies from two different data subjects are involved. This allows further investigation for a permissive approach to keep the program error free and GDPR compliant when one data subject

revokes consent or data deletion deadline arrives for one policy. Moreover, DPL is an experimental language modelled in Maude, it can be incorporated in Java.

For paper III, again a larger case study is required to better assess our approach's runtime overhead. The wrapper mechanism could be integrated in a runtime system in a way that protects the wrapped components. Other possibilities for future work could be to extend our language with different concurrency paradigms and investigate their affect on noninterference. Moreover, our security approach with wrappers can be further investigated to satisfy progress-sensitive and termination-sensitive noninterference properties. For a stronger correctness, our proof sketch for the noninterference property can be thoroughly formalized.

Part II

# Research papers

# Papers

Paper I

# DPL: A Language for GDPR Enforcement

I

**Farzane Karami, David Basin, Einar Broch Johnsen**

Published in *IEEE 35th Computer Security Foundations Symposium (CSF)*, 2022, pp. 112–129. DOI: 10.1109/CSF54842.2022.9919687.

**Abstract**

The General Data Protection Regulation (GDPR) regulates the handling of personal data, including that personal data may be collected and stored only with the data subject's consent, that data is used only for the explicit purposes for which it is collected, and that is deleted after the purposes are served. We propose a programming language called DPL (Data Protection Language) with constructs for enforcing these central GDPR requirements and provide the language's runtime operational semantics. DPL is designed so that GDPR violations cannot occur: potential violations instead result in runtime errors. Moreover, DPL provides constructs to perform privacy-relevant checks, which enable programmers to avoid these errors. Finally, we formalize DPL in Maude, yielding an environment for program simulation, and verify our claims that DPL programs cannot result in privacy violations.

## I.1 Introduction

The General Data Protection Regulation (the GDPR) [69] regulates the processing of personal data and is now part of European Union law. The GDPR mandates transparent data processing, where data is collected with the data subject's consent and used only for the purposes for which it was collected. Moreover, the GDPR requires the right to be forgotten, where data must be deleted on request or after its purposes are served. It is an open question how systems processing sensitive data should be built to satisfy these requirements.

We approach this problem from a programming language perspective: how can one design a language to prevent data-protection violations? Conventional programming languages do not support the features essential to enforce GDPR compliance. For example, they lack an explicit representation of purpose,

The authors were partially funded by the Research Council of Norway through IoTSec (project no. 248113.

and there is no control over data collection and usage based on purposes and consent. The state-of-the-art generally checks purpose-based privacy compliance in programs using privacy labels and static information-flow analysis [32, 47, 74], where the enforced policies are determined by policy labels specified by the programmer at compile time. This has its limitations. For example, users' consent and deadlines for data deletion cannot be expressed in policies. Moreover, temporal aspects cannot be handled when consent is dynamically granted or revoked or retention deadlines are reached.

In this paper, we enforce GDPR requirements at runtime. Our approach allows us to enforce richer policies than those enforced statically. For example, we can express temporal requirements on data deletion, and the data will be deleted from the system when the deadline arrives. In contrast to static approaches, our runtime approach is also more exact, since users' consent, given at runtime, determines for what purposes their data can be used, and these purposes are added to the policies. However, in contrast to static approaches, our approach comes at the price of runtime overheads.

The GDPR requirements that we handle concern data usage (see Section I.2.1), and we present language features to enforce these requirements. Our focus is on object-oriented and service-oriented languages, where objects are entities, method calls are processes, which may use personal data, and return values are the outputs of processes. The language involves the following features: 1) Object databases with commands for data storage and retrieval. 2) Interface encapsulation, which enforces programming to interfaces and prevents remote access to fields and methods. 3) Language constructs to build and manipulate privacy policies including **`policy`** to create privacy policies, **`opt-in`** for granting consent, and **`collect`** for collecting data. The policies describe to whom data belongs, for which purposes data can be used, and when data must be deleted. 4) Runtime checking to ensure that processes only access data as authorized by their privacy policies.

We define DPL, a data protection language enriched with the above features. Additionally, DPL supports the users' rights to opt-out of policies or delete their data at any time. Table I.1 summarizes the GDPR requirements we handle, the object-oriented (OO) features exploited in DPL, and our specific extensions for privacy-related operations. While we present DPL as a stand-alone language, our language features could also be implemented as an extension of existing languages. Its core is inspired by the concurrent active object language ABS [42], which achieves encapsulation by typing objects by interfaces (instead of classes). One could also use Java directly, which supports encapsulation using the private modifier for defining local methods and fields in classes, whereas methods in interfaces are defined with the public modifier. We believe that our approach could be carried out with other Java-like languages such as Java 8 [75], Scala/Akka [30, 84], and ASP [13].

In DPL, interfaces represent purposes, and their declared methods are the processes used to achieve the purpose. Personal data is collected after a user gives consent, and the identity of the corresponding privacy policy is attached to the data, which is then called *sensitive data*. DPL supports dynamic policy changes, where users grant or withdraw consent, and hence the policy evolves over time.

| GDPR requirements | OO features | Added features |
|---|---|---|
| consent | | **opt-in**, **opt-out** |
| purpose limitation | interface encapsulation<br>message passing | **policy**, **collect**<br>runtime checking |
| storage limitation | object model | **store**, **retrieve**<br>objects' databases |
| right to be forgotten | | delete |

Table I.1: GDPR requirements and associated features.

Moreover, policies expire and are deleted from the system when deadlines arrive or the user deletes her data. We define DPL's runtime system that tracks the flow of sensitive data and performs runtime checking (see Section III.4.2). A process is allowed to access sensitive data if its intended usage complies with the purpose in the privacy policy, and the policy has not expired. In DPL, the failure to comply to a policy will not result in a privacy violation but rather a runtime error. Moreover, DPL provides constructs to perform privacy-relevant checks so that programmers can write programs that avoid actions that would lead to policy violations rather than throwing runtime errors.

We formalize DPL's operational semantics in rewriting logic [52], which is supported by the Maude system [18]. This provides a prototype interpreter[1] for executing DPL programs. We also use Maude's model checker to complement our pen-and-paper proofs with model-checked examples that support our claims.

In summary, our contributions are as follows. We define DPL, an object-oriented language extended with features that support data protection. We map the GDPR data usage requirements to our language, and define an executable formal model for DPL, given by an SOS-style operational semantics. We provide a pen-and-paper proof that DPL programs satisfy the GDPR data usage requirements. DPL provides exact enforcement of privacy policies, it is user-centric in that users' consent is reflected in policies, and it enforces richer policies than those enforced by static approaches. We also formalize DPL in rewriting logic, use Maude's model-checker to verify our data protection properties on concrete programs, and illustrate on examples how privacy violations cannot occur. Overall, DPL is the first programming language designed for developing programs that comply to GPDR data usage requirements.

## I.2 Background

### I.2.1 GDPR requirements

Our focus is on the following requirements, which are central to the GDPR's restrictions on data usage.

[1]The Maude model is available at https://github.com/maude-gdpr/maude.

**Purpose limitation:** *"[Personal data shall be] collected for specified, explicit and legitimate purposes and not further processed in a manner that is incompatible with those purposes"* [69, Article 5, Sec. 1 (b)]. Data is considered to be personal data if it can be used, directly or indirectly, to identify a person [82]. To comply with this, the purposes for which personal data is collected must be made explicit, and the collected data must be used only for those purposes.

**Consent:** *Personal data is collected only if the data subject's consent is granted. In order to give consent, a data subject should be aware of the identity of the controller and the purpose of processes in which her data is used* [69, Article 6]. *Moreover, a data subject has the right to withdraw her consent at any time* [69, Article 7]. To comply with this, data collection requires consent, and personal data must no longer be processed after consent is withdrawn.

**Storage limitation:** *"[Personal data shall be] kept in a form which permits identification of data subjects for no longer than is necessary for the purposes for which the personal data are processed"* [69, Article 5, Sec. 1 (e)]. To comply with this, personal data shall be deleted after the purpose of processing is fulfilled [7]. For example, a credit card number is collected to make a purchase, and if a data subject consents, this information can be stored for subsequent purchases. The storage period shall be limited to a strict minimum, and a controller shall establish time limits for data erasure [69, Rec. 39].

**Right to be forgotten:** *"The data subject shall have the right to [...] the erasure of personal data concerning him or her [...] and the controller shall have the obligation to erase personal data without undue delay [...]"* [69, Article 17]. To comply with this, data must be promptly deleted on request.

### I.2.2 An example

To illustrate our methodology in subsequent sections, we use an example taken from [7]. The example features an online retailer whose core processes are:

**Register customer:** A customer provides her credit card information, her e-mail, and her postal address.

**Purchase:** A registered customer purchases a product from the retailer's online shop using her registered credit card. This process produces the customer's order along with an invoice, which is sent to her address.

**Mass Marketing:** A customer's email or postal address is used to send untargeted advertisements.

**Targeted Marketing:** A customer's email or postal address and her shopping history are used to send targeted advertisements.

The GDPR requires *consent statements* for processes using personal data. A consent statement describes what data is used for which purposes. For example, the consent statement for **Mass Marketing** is "we use your customer information (name and email address) for mass marketing."

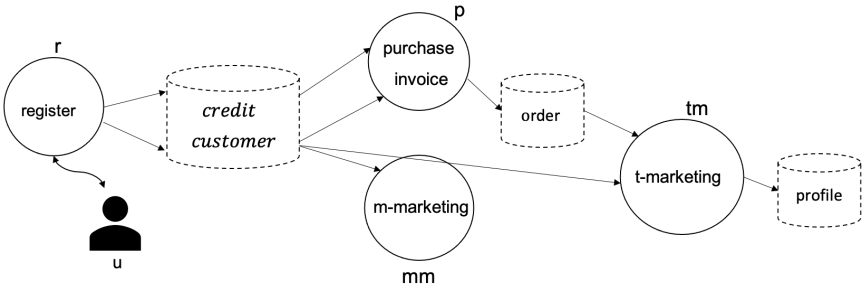


Figure I.1: The online-retailing example in our system model (after [7]).

### I.2.3 System model

Our system model formalizes how distributed applications process users' personal data. It features users, objects, and databases, where objects share data through message passing. We assume that all communication is cryptographically protected, e.g., using TLS, and focus on data protection in this distributed setting. Later we present DPL, which formalizes these distributed systems and their behavior.

Figure I.1 shows the on-line retailing example in our system model. In this example, the personal data is credit card information, customer information, the order, and the user profile. Circles represent objects with the names $r$, $p$, $mm$, and $tm$ and contain the name of the processes that use the personal data. Dashed cylinders represent the objects' databases. The curved bidirectional arrow represents interaction with a user to collect data over a user interface. For example, the method *register* of the object $r$ collects credit card information and customer information from the user $u$ and stores this data in its database. Arrows from databases to objects represent database data used by the objects' methods. Arrows from objects to databases represent that method results are stored in the databases.

## I.3 DPL: a calculus for data protection

We now describe the principles behind DPL as well as its syntax and semantics. The complete formalization of the syntax and semantics in Maude, along with all auxiliary functions and examples, can be found at https://github.com/maude-gdpr/maude.

### I.3.1 Language design principles

We now return to the GDPR requirements from Section I.2.1 and explain DPL's design principles and language features used to enforce the requirements.

In what follows, we will use the following notation. Let $x_1, x_2, ..., x_n$ represent a sequence of $n$ terms, where $\epsilon$ is the empty sequence. We use the notation $\overline{x}$ to range over sequences, possibly empty, and $\overline{x}^1$ to emphasize that the sequence has at least one element. We write lists using ":" and the empty list as []. We also employ standard list notation and write a list like $x1 : (x2 : [])$ as $[x1, x2]$.

**Purpose limitation** To enforce this requirement, the purposes for which personal data are collected and used should be made explicit. In DPL, instead of using interfaces, we explicitly define purposes by a declaration **purpose** $P\{\overline{Sig}^1\}$, with $P$ a name and $\overline{Sig}^1$ the method signatures (the methods' return types and parameters) required to achieve the purpose. Moreover, encapsulation is achieved by typing objects by purposes. Objects created from classes implementing purposes provide methods to achieve the purposes. We assume that programs are well-typed, which could be enforced by adapting a standard type system for interfaces (e.g., [42]), that all methods using personal data are declared in some purposes, and that each purpose contains exactly the methods needed to achieve it. For example, we can define the purpose *Purchase* as follows.

```
purpose Purchase {
Order purchase(String credit, String customer);
Invoice invoice(Order order, String customer); ... }
```

We propose a mechanism that prevents personal data from going to objects that implement no purpose, the wrong purpose, or even the right purpose when it has not been consented to by the data owner. First, we define *contracts* declaring which object may use the methods associated with a purpose. Afterwards, we define *privacy policies*, which are attached to collected data and contain sets of contracts. Contracts can be added to or removed from policies when a user opts in or opts out, respectively. The contracts define the objects' access rights; i.e., if a contract belongs to a policy, then the associated object can use the data.

**Definition I.3.1** (Contract)**.** A contract is an expression **contract**$(P, e)$, where $P$ is a purpose and $e$ is an object that belongs to a class implementing $P$.

For example, the expression **contract**$(Purchase, p)$ defines a contract for the *Purchase* declaration, where $p$ is an object of type *Purchase*. We say that an object's contract *complies to a policy* if the contract belongs to the runtime representation of that policy, defined in the following.

**Definition I.3.2** (Privacy policy)**.** A privacy policy is a runtime element represented as a five-tuple $(u, cp, cm, b, t)$, where $u$ is the identity of the user whose data is collected, $cp$ is a set of persistent contracts, $cm$ is a set of mutable contracts that are updated when opting in or out, $b \in \{true, false\}$ denotes whether the collected data should be stored persistently, and $t \in \mathcal{N}$ is a natural number representing a timestamp specifying when the collected data should be deleted.

The terms $u$, $cp$, and $cm$ in a privacy policy are initialized when a user logs in, the policy is created, and the user gives consent using **opt-in** statements,

respectively. The Boolean $b$ and the timestamp $t$ capture retention and deletion requirements, respectively. Time is an integral part of our model. We will later specify a system clock that decrements the timestamps in all policies; data deletion is triggered when deadlines arrive.

We require a privacy policy for all personal data collected. For example, since credit card and customer data are used for different purposes, we define a new policy for each kind of data. Note that a policy can be used for different types of data if the data is used for the same purposes.

**Consent** Here we explain how to write strings for consent statements to accurately represent intended purposes. In the declaration **purpose** $P$ $\{\overline{Sig}^1\}$, if the method parameters in $\overline{Sig}^1$ contain personal data such as credit card and customer data, then the consent statements are "We use credit card data for $P$" and "We use customer data for $P$". Moreover, we can use the identifier $X$ of the entity that will use the data instead of "we" and extend the statement with our retention and deletion policies. For example, the consent statement for *Purchase* becomes: "X uses your credit card number for purchasing, and your data is stored for one year" and similarly, for customer data. There is a correspondence between the consent statement and the contract of a purpose declaration. A consent statement is used to collect a user's consent, and the corresponding contract is used to control objects' access to the user's data.

**Right to withdraw consent** In DPL, a contract can be removed from a policy anytime, and the data associated with that policy is no longer used for the withdrawn purpose. This models the user action for withdrawing consent.

**Storage limitation** In DPL, objects encapsulate their states. An object state consists of a substitution for process-local variables, mapping variables to data, a substitution for fields, and a substitution for the object's database. Local variables are deleted after process termination, when a purpose is served. We do not allow assigning sensitive data to fields because fields store data as long as the object is alive, and all of the object's methods may have access to the fields.

Data must sometimes be stored for a time period captured by a time value $t$. For data storage, we integrate databases into DPL's object model, where for simplicity, our databases are just key-value stores. An object can store and retrieve data in its database, and remote access to databases is prohibited. In DPL, when the system clock advances, the timestamp of every policy is decremented. When the timestamp reaches one, the policy is deleted from the system, and data associated with the policy is also deleted from databases. Note that sensitive data in local variables can no longer be used since the policy is deleted.

**Right to be forgotten** In DPL, a policy can be deleted anytime, and the data associated with that policy is deleted from objects' databases. This models a user action for data deletion. Deletion is also triggered when deadlines arrive. Handling these restrictions without undue delay is nontrivial, as there can be

race conditions involving the time of (authorization) check and the time of use. For example, after we check compliance, a method is authorized to use data, but prior to its use, a deletion deadline may arrive or consent may be withdrawn.

In DPL, errors will be thrown if expired data is used. But race conditions between time-of-check and time-of-use mean that user-provided checks are insufficient to prevent all errors from arising. We observe though that such race-conditions are generally not critical in practice since data protection is not a hard real-time requirement. When data is deleted (or alternatively consent is revoked) "undue delay" does not mean that everything aborts and the data is instantly deleted, but rather, as soon as reasonably possible, the system will no longer process the data and it will be removed. When we use the services of Google or Facebook, our expectations for deletion are on the order of minutes or hours, not seconds.

To reflect this in DPL, we introduce *compliance scopes* where compliance is checked and assumed to remain valid within the scope. Namely, within a scope, we allow a finite number of program steps to proceed without checking compliance since the compliance was checked when entering the scope. The finite steps provide an abstract representation of the temporal notion "without undue delay." Of course, we must avoid non-terminating processes as otherwise compliance would not be checked for an arbitrary amount of time. Hence loops and recursive calls are omitted from our compliance scopes.

### I.3.2 Syntax

We define DPL's grammar in Fig. I.2. The types $T$ are base types $B$, purposes $PI$, as well as types for policies, contracts, users, consent statements (CStmt), keys, and sensitive data with associated policies. A program $PR$ includes purposes, classes, and a main block. A class may implement one or more purposes $\overline{P}^1$ and has methods $\overline{M}^1$. A method signature consists of a return type $T$, a method name $m$, and typed parameters declarations $\overline{T\ x}$. A method definition $M$ consists of a signature and a body with local variables and statements. Statements $s$ include sequential composition, assignment, and privacy-specific constructs to be discussed shortly. There is no surface syntax for directly constructing sensitive data; this happens indirectly via **`collect`**-statements. Right-hand-side expressions *rhs* include (pure) expressions $e$ and method calls, as well as object and policy creation expressions, which create references at runtime. Although method calls $e.m(\overline{e})$ and return statements are standard, they can transfer sensitive data between objects. Data $d$ consists of contracts **`contract`**$(P, e)$, where $P$ is a purpose and $e$ an object implementing that purpose, consent statements **`cstmt`**$(str)$, constructed from strings, and keys **`key`**$(u, str)$, where $u$ is a user and $str$ a string used as a tag to denote a particular attribute associated with $u$, in addition to strings, Boolean values, and natural numbers. Expressions $e$ consist of data $d$, variables $x$, and operations $op$ on $e$ (e.g., logical and arithmetic operators).

DPL has the following non-assignable *reserved variables*: the self reference *this*, the self contract *cnThis*, the caller reference *caller*, the caller's contract

$$
\begin{aligned}
B &::= \mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{String} \\
T &::= B \mid PI \mid \mathsf{Policy} \mid \mathsf{Contract} \mid \mathsf{User} \mid \mathsf{CStmt} \mid \mathsf{Key} \\
&\quad \mid \mathsf{Sensitive}\langle B, \mathsf{Policy}\rangle \\
PI &::= \mathbf{purpose}\ P\{\overline{Sig}^{1}\} \\
PR &::= \overline{PI}\ \overline{CL}\ \mathbf{main}\{\overline{T\ x}; s\} \\
CL &::= \mathbf{class}\ C(\overline{T\ x})\ \mathbf{implements}\ \overline{P}^{1}\{\overline{T\ x};\ \overline{M}^{1}\} \\
Sig &::= T\ m(\overline{T\ x}) \\
M &::= Sig\ \{\overline{T\ x};\ s\} \\
s &::= s; s \mid x := rhs \mid \mathbf{return}\ e \mid \mathbf{skip} \mid \mathbf{log\text{-}in}() \\
&\quad \mid \mathbf{store}(k, e)\ \mathbf{else}\ \{s\} \mid \mathbf{opt\text{-}in}(cs, cn, l) \mid \mathbf{log\text{-}out}() \\
&\quad \mid \mathbf{if\text{-}comply}(cn, \overline{e})\ \{s\}\ \mathbf{else}\ \{s'\} \\
&\quad \mid \mathbf{if\text{-}consent}(cn, l)\ \{s\}\ \mathbf{else}\ \{s'\} \\
&\quad \mid \mathbf{retrieve}(k, x)\ \{s\}\ \mathbf{else}\ \{s'\} \mid \mathbf{collect}(cn, l, x) \\
rhs &::= e \mid e.m(\overline{e}) \mid \mathbf{new}\ C(\overline{x}) \mid \mathbf{policy}(b, t) \\
d &::= \mathbf{contract}(P, e) \mid \mathbf{cstmt}(str) \mid \mathbf{key}(u, str) \mid str \mid bool \mid nat \\
e &::= d \mid x \mid e\ op\ e \\
op &::= + \mid - \mid ...
\end{aligned}
$$

Figure I.2: DPL's grammar; to simplify the presentation, let $b$ range over Boolean expressions, $u$ users, $cs$ consent statements, $cn$ contracts, $l$ policies, $P$ purposes, $t$ timestamps, and $k$ keys.

*cnCaller*, and the reference *user* for a logged-in user.

In DPL, data can be collected only within a session, which starts with **log-in**() and ends with **log-out**(), and a user can grant or deny consent using **opt-in**$(cs, cn, l)$ statements. Moreover, data collection is not a primitive, but rather composed from atomic statements such as:

$$
\begin{aligned}
&\mathbf{log\text{-}in}();\ l := \mathbf{policy}(b, t);\ \mathbf{opt\text{-}in}(cs, cn, l); \\
&\quad \mathbf{if\text{-}consent}(cn, l)\{\ \mathbf{collect}(cn, l, x)\ \};\ \mathbf{log\text{-}out}();
\end{aligned}
$$

Here, a user logs-in and a new privacy policy is created by **policy**$(b, t)$, which returns a unique policy identity, assigned to $l$. Then, if the user gives consent to the consent statement $cs$, the contract $cn$ is added to the policy $l$. The condition **if-consent**$(cn, l)$ checks if consent has been granted for $cn$ in the policy $l$, in which case data is collected from the user interface by **collect**$(cn, l, x)$, under the contract $cn$; the policy $l$ is attached to the data, and the resulting sensitive data is assigned to $x$. Note that we allow syntactic sugar where we omit the **else**-branches when they are not needed.

The statement **store**$(k, e)$ checks compliance for storage and, when compliant, the data $e$ is stored in the database, with the key $k$. Otherwise, the data is not stored, and the **else**-branch is executed. Conditional constructs enable the programmer to make checks to ensure GDPR compliance. These checks may be omitted, in which case the failure to comply will result in a runtime error in

DPL rather than a privacy violation in non-DPL systems.

The statement **if-comply**$(cn, \overline{e})$ checks that all elements in the list $\overline{e}$ are GDPR compliant with respect to the contract $cn$ and **if-consent**$(cn, l)$ checks that consent has been granted to a contract $cn$ under a policy $l$. The conditional constructs open and close compliance scopes in the **if**-branch. The statement **retrieve**$(k, x)$ checks if the key $k$ is in a database, in which case data is retrieved and assigned to $x$. In all three of these statements, if the check succeeds then the success branch $(s)$ is executed and otherwise the else branch $(s')$ is executed.

Note that to analyze the GDPR compliance of DPL programs in this paper, we capture the user actions for withdrawing consent and data deletion directly in DPL's operational semantics, such that they may occur at any time, instead of programming them explicitly in the program's surface syntax.

### I.3.3 Example

We illustrate how DPL provides the essential ingredients needed to develop a GDPR compliant system, as discussed in Section I.2.1, by implementing the online-retailing example of Section I.2.2. Figure I.3 presents the DPL code, focusing on consent statements, contracts, and data collection. We extend this code in Appendix I.A.4 to show how DPL enforces *purpose limitation* and *storage limitation* in remote objects receiving sensitive data. User actions for data deletion and for withdrawing consent may occur at any time, reflecting the user's *right to be forgotten* and *right to withdraw consent*.

Lines 1–5 define purposes for Purchase, MassMarketing, and Registration. We omit the details of classes Purchase-c, MMarketing-c, and Order, and focus on the class Register, which implements data collection, and its register method (line 8). First, a user is logged-in and two privacy policies l1 and l2 are created, which both expire at a given time t (here one year, written in seconds). The **opt-in** statements (line 14) let the user grant (or deny) consent to the consent statements cs1, cs2, and cs3, in which case the associated contracts are added to the policies. Credit card and customer data are collected from the user (lines 16 and 18), returning data credit and customer of types Sensitive⟨String,l1⟩ and Sensitive⟨String,l2⟩, respectively. Keys are constructed with the tags "credit" and "customer" and are used to store the sensitive data in the database (lines 17 and 19). The **log-out**() statement ends the registration, which returns a tuple with the user identifier and the two policies. (We go slightly beyond the defined syntax here by directly returning a tuple instead of creating a result object.)

In the main block, remote calls initiate the processing of the users' data. The main block first creates objects r, p, and mm, typed by purposes. Then consent statements (lines 34–39) and contracts (lines 40–41) are defined. The consent statements and contracts are passed to the method r.register (line 43), which returns a user identifier and policies (for simplicity, we simultaneously assign to all three variables $u$, $l1$, and $l2$ instead of going via a result object). Then, the methods r.getCredit and r.getCustomer are called to retrieve the user's credit card and customer data, respectively (lines 44–45). Afterwards, the purchase and m-marketing methods are called (lines 47–48).

```
1  purpose Purchase {
2      Order purchase(String credit, String customer); ... }
3  purpose MassMarketing{ String m-marketing(String customer);}
4  purpose Registration{ ⟨String, Policy, Policy⟩ register(...);
5      String getCredit(User u); String getCustomer(User u);}
6  // Definitions of classes Purchase–c, MMarketing–c, Order, ...
7  class Registration-c implements Registration () {
8      ⟨String, Policy, Policy⟩ register(Contract cn1, CStmt cs1, CStmt cs2,
9      Contract cn2, CStmt cs3) {
10         String credit; String customer; Nat t; t := 31,536,000; // Seconds
11         log-in(); // binds a user identifier to the variable user
12         // Privacy policies for collected data items
13         Policy l1 := policy (true, t); Policy l2 := policy (true, t);
14         opt-in(cs1, cn1, l1); opt-in(cs2, cn1, l2); opt-in(cs3, cn2, l2);
15         // Data collection and storage
16         if-consent(cn1, l1) { collect(cn1, l1, credit);
17             store(key(user,"credit"), credit) };
18         if-consent(cn1, l2) { collect(cn1, l2, customer);
19             store(key(user,"customer"), customer) };
20         log-out();
21         return (⟨user, l1, l2⟩);
22 }
23     String getCredit(User u) { String credit;
24         retrieve(key(u,"credit"), credit) {
25             if-comply(cnCaller, credit){return(credit)} else { return("0")}
26             else { return("0")} }
27         String getCustomer(User u) { ... }
28     }
29     main{ String credit, customer;
30         User u; Policy l1; Policy l2; Purchase p := new Purchase-c();
31         MassMarketing mm := new MMarketing-c();
32         Registration r := new Registration-c();
33         // Consent statements and contracts
34         CStmt cs1 := cstmt("X uses your credit card number for
35         purchasing and your data is stored for one year.");
36         CStmt cs2 := cstmt("X uses your customer information for
37         purchasing and your data is stored for one year.");
38         CStmt cs3 := cstmt("X uses your customer data for
39         mass marketing.");
40         Contract cn1 := contract(Purchase, p);
41         Contract cn2 := contract(MassMarketing, mm);
42         // Call the register method for data collection
43         ⟨u, l1, l2⟩ := r.register(cn1, cs1, cs2, cn2, cs3);
44         if-consent(cn1, l1) { credit := r.getCredit(u) };
45         if-consent(cn1, l2) { customer := r.getCustomer(u) };
46         if-comply(cn1, (credit, customer)) {
47             Order order := p.purchase(credit, customer) };
48         if-comply(cn2, customer) {mm.m-marketing(customer)}... }
```

Figure I.3: Online-retailing example in DPL.

The example uses conditional constructs to avoid runtime errors. For example, in line 44, `if-consent`(cn1, l1) checks if consent for Purchase is granted, in which case the method getCredit is called. In line 46, `if-comply`(cn1, (credit, customer))

$$
\begin{aligned}
\mathit{Cfg} &::= \{\mathit{cfg}\} \\
\mathit{cfg} &::= \emptyset \mid \mathit{obj} \mid \mathit{msg} \mid \mathit{policy} \mid \mathit{class} \mid \mathit{error} \mid \mathit{cfg}\ \mathit{cfg} \\
\mathit{obj} &::= o(a, p, db) \\
\mathit{msg} &::= m(\overline{v}, o', o, cn) \mid \mathit{com}(v, o) \mid n(\mathit{process}) \\
\mathit{policy} &::= l(u, cp, cm, b, t) \\
\mathit{error} &::= \mathit{errorU}(o, cn) \mid \mathit{errorC}(o, cn) \mid \mathit{error}(o) \\
p &::= \mathit{process} \mid \mathit{idle} \\
\mathit{process} &::= (\sigma, s@V) \\
v &::= o \mid l \mid d \mid \mathit{sensitive}(d, l) \\
s &::= m? \mid \mathit{cScope}(S) \mid \mathit{cont}(n) \mid \ldots
\end{aligned}
$$

Figure I.4: The runtime elements, where S is a set of policy-contract pairs.

checks if the contract cn1 complies to the privacy policies of the variables credit and customer, which are passed as parameters to the method purchase. Here, the method is only called if the compliance check holds. In line 16, if consent is not granted, data is neither collected nor stored. In line 25, **if-comply**(cnCaller, credit) checks compliance with respect to the caller's contract before the **return**-statement; if compliance fails, some default value "0" is sent instead of the credit card number. Moreover, if **retrieve**(**key**(u,"credit"), credit) in line 24 fails, a default "0" is sent in line 26. We omit further error handling in this example and do not test against these default values, but remark that they play the role of ad hoc option-types, suggesting ways to further enrich DPL.

### I.3.4 Operational semantics

We define DPL's operational semantics using multiset rewriting [52]. Although DPL is presented as a single-threaded system, multiple rewrite rules may be simultaneously enabled, and the execution of a program gives rise to multiple transition sequences (traces). For example, time can always advance, consent be withdrawn, or data deletion be triggered. These interleavings can give rise to race conditions between time-of-check and time-of-use for GDPR compliance.

**Runtime elements** DPL's runtime syntax is shown in Fig. I.4. A global configuration *Cfg* is a bracketed multiset of runtime elements: objects, messages, privacy policies, and classes. An *object* $o(a, p, db)$ has an identifier $o$, a substitution $a$, which maps the object's fields to values, an active process $p$, which may be *idle*, and a database $db$, which maps keys to data. A *process* combines a substitution $\sigma$, which maps process-local variables to values, with a runtime statement $s@V$, where $s$ is a statement and $V$ a compliance scope.

Messages represent process invocation, completion, and suspension. In an *invocation message* $m(\overline{v}, o', o, cn)$, $m$ is the name of the called method, $\overline{v}$ the actual parameters, $o'$ the callee, $o$ the caller, and $cn$ the caller's contract. A *completion message com(v,o)* contains a result value $v$ and a receiver object $o$. In

a *suspension message* $n(process)$, $n$ is a name and *process* a suspended process. *Policies* were already defined in Def. I.3.2. Classes are simple look-up tables for method definitions, and are omitted here. We use white space to denote the composition of configurations and $\emptyset$ for the empty configuration, which is the identity for the composition.

We consider three kinds of errors for a process executing in an object $o$: $errorU(o, cn)$ is a *data usage error* expressing that data usage in $o$ is not compliant with the contract $cn$; $errorC(o, cn)$ is a *data collection error* expressing that consent associated with the contract $cn$ is not granted; and $error(o)$ represents other errors.

*Values* $v$ include identifiers $o$ and $l$ for objects and privacy policies, data $d$, and sensitive data $sensitive(d, l)$. We extend the statements $s$ of Fig. I.2 as follows: $m?$ blocks an object after calling a method, *cScope* marks the end of a compliance scope; and $cont(n)$ schedules a suspended process $n$.

*Substitutions* bind fields, process-local variables, or keys to values, respectively. Thus, a substitution $\theta$ is a finite map, written $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$. We write $\theta(x)$ to lookup the variable $x$ in $\theta$, and $\theta[x \mapsto v]$ to update $\theta$ with the binding $[x \mapsto v]$. In the composition $\theta \circ \theta'$, the bindings in $\theta'$ shadow those in $\theta$, so $\theta \circ \theta'(x) = \theta'(x)$ if $x \in dom(\theta')$ and $\theta \circ \theta'(x) = \theta(x)$ otherwise. The notation $\theta|_C$ denotes domain restriction of $\theta$ to a set $C$ of variables and [] denotes the empty substitution.

**Compliance** At runtime, execution happens in the context of compliance scopes. DPL features specific condition constructs to interact with these scopes. The dynamic checking of compliance is formalized by a predicate *comply* that captures our notion of compliance between policies and a contract, either by inspecting the compliance scope of the executing process or by a direct dynamic check of the configuration. Since the scope is only extended by performing a compliance check, the scope introduces a delayed effect for compliance checking in that execution may continue within the scope even after consent has been withdrawn.

Let $V$ be a compliance scope and *ls* a list of policies. The *comply* predicate is defined inductively over a list of policies with an auxiliary predicate performing the runtime *check*:

$$
\begin{array}{l}
comply([], cn, cfg, V) = true \\
comply((l : ls), cn, cfg, V) = \\
\quad (\langle l, cn \rangle \in V \ \vee \ check(l, cn, cfg)) \ \wedge \ comply(ls, cn, cfg, V)
\end{array}
$$

$$
check(l, cn, cfg) = \begin{cases} cn \in (cp \cup cm) & \text{if } l(u, cp, cm, b, t) \in cfg \\ false & \text{otherwise} \end{cases}
$$

**The transition system** DPL's operational semantics is defined using multiset rewrite rules [52], which define a transition relation on configurations. Rewrite rules may be conditional, and we present them as inference rules with zero or more conditions as premises and a labelled transition $lhs \xrightarrow{l} rhs$ between patterns *lhs* and *rhs* as the conclusion. The rule can be applied to a sub-multiset *cfg*

of a configuration if *lhs* matches *cfg* for some substitution $\theta$ and the premises hold. The rule's effect is to replace *cfg* in the global configuration with *rhs*, to which the substitution $\theta$ is applied. Matching is modulo associativity and commutativity in the multiset and hence no structural rules are needed to reorder runtime elements in configurations.

We present the rewrite system in three parts: rules for user interaction, rules for storage, deletion and scopes, and rules for the standard execution of statements. Our focus here is on sensitive data; standard rules for non-sensitive data are given in Appendix I.A.2. We have formalized the rewrite system in Maude [18], and this formalization is further described in Appendix I.A.3.

The *evaluation of expressions* is formalized by a function $[\![e]\!]_\theta$ from expressions $e$ and substitutions $\theta$ to values. For example, **`contract`**$(P, e)$ evaluates to **`contract`**$(P, o)$ where $o$ is an object reference. Evaluation is untyped: we assume that programs are well-typed such that evaluation does not get stuck and produces meaningful values. We also employ other auxiliary functions, which are briefly explained the first time they are referenced.

Article 20 of the GDPR states that the processing of data shall not adversely affect the rights and freedoms of others. Therefore, we disallow binary operations on two sensitive data items where the policies belong to different users. For simplicity, binary operations are also disallowed if the policies belong to the same user but different data types. This avoids complications regarding the withdrawal of consent from one policy. We leave open more permissive solutions, e.g., involving the intersection of (consented) purposes in policies, as future work.

**User interaction** Figure I.5 presents rewrite rules involving sessions, policies, consent, data collection, and data deletion. Labels on the transition relation represent input data from a user interface or a "tick" from an external clock. A transition is unlabeled if no input is required.

Data can only be collected and contracts only added to a privacy policy within a *session*. We first consider the effects of **`log-in()`** on the active process of an object. If no user is currently logged-in, the reserved variable *user* is bound to the user identifier $u$ in LOG-IN, starting a session. Otherwise, ERROR-LOG-IN triggers an error. Rule LOG-OUT removes *user* from the local variables, ending the session. If no user is logged-in, ERROR-LOG-OUT triggers an error.

Rule POLICY creates a *privacy policy* with identifier $l$. The predicate $\mathit{fresh}(l)$ expresses that the name $l$ is unique in the global configuration. The persistent contracts are initialized to the contracts of $o$ and $\mathit{ob}(\mathit{main})$, and the set of mutable contracts is initially empty. These sets reflect the policy for data collection; only mutable contracts can be removed by the user. If no user is logged-in, ERROR-POLICY triggers an error.

A user may grant consent to the policy for a given contract. In OPT-IN, the user accepts a consent statement *cs*, represented by the label "*yes*", and the associated contract *cn* is added to the mutable contracts *cm* of the policy $l$. If no user is logged-in, the policy does not exist in *cfg*, or the user denies consent, NO-OPT-IN formalizes that the **`opt-in`** has no effect. Here, $\mathit{idExist}(l, \mathit{cfg})$

LOG-IN

$$\frac{user \notin dom(\sigma)}{o(a,(\sigma,(\textbf{log-in}();s)@V),db) \xrightarrow{\text{“}u\text{”}} o(a,(\sigma[user \mapsto u],s@V),db)}$$

ERROR-LOG-IN

$$\frac{user \in dom(\sigma)}{o(a,(\sigma,(\textbf{log-in}();s)@V),db) \rightarrow error(o)}$$

POLICY

$$\frac{\begin{array}{c} v_1 = [\![b]\!]_{a \circ \sigma} \quad v_2 = [\![t]\!]_{a \circ \sigma} \\ user \in dom(\sigma) \quad u = \sigma(user) \quad fresh(l) \end{array}}{\begin{array}{c} o(a,(\sigma,(x := \textbf{policy}(b,t);s)@V),db) \rightarrow \\ o(a,(\sigma,x{:=}l;s@V),db)\ l(u,\{contract(main,ob(main)),a(cnThis)\},\emptyset,v_1,v_2) \end{array}}$$

LOG-OUT

$$\frac{user \in dom(\sigma)}{o(a,(\sigma,(\textbf{log-out}();s)@V),db) \rightarrow o(a,(\sigma|_{dom(\sigma)\setminus\{user\}},s@V),db)}$$

ERROR-LOG-OUT

$$\frac{user \notin dom(\sigma)}{o(a,(\sigma,(\textbf{log-out}();s)@V),db) \rightarrow error(o)}$$

OPT-IN

$$\frac{\begin{array}{cc} cs = [\![e_1]\!]_{a \circ \sigma} & l = [\![e_3]\!]_{a \circ \sigma} \\ cn = [\![e_2]\!]_{a \circ \sigma} & user \in dom(\sigma) \end{array}}{\begin{array}{c} o(a,(\sigma,(\textbf{opt-in}(e_1,e_2,e_3);s)@V),db)\ l(u,cp,cm,b,t) \\ \xrightarrow{\text{“}yes\text{”}} o(a,(\sigma,s@V),db)\ l(u,cp,cm \cup \{cn\},b,t) \end{array}}$$

TICK

$$\{cfg\} \xrightarrow{tick} \{dec(cfg)\}$$

NO-OPT-IN

$$\frac{\begin{array}{c} l = [\![e_3]\!]_{a \circ \sigma} \\ user \notin dom(\sigma) \vee \neg idExist(l,cfg) \vee x = \text{“}no\text{”} \end{array}}{\{o(a,(\sigma,(\textbf{opt-in}(e_1,e_2,e_3);s)@V),db)\ cfg\} \xrightarrow{x} \{o(a,(\sigma,s@V),db)\ cfg\}}$$

ERROR-POLICY

$$\frac{user \notin dom(\sigma)}{o(a,(\sigma,(x := \textbf{policy}(b,t);s)@V),db) \rightarrow error(o)}$$

OPT-OUT

$$l(u,cp,\{cn,\overline{cn}\},b,t) \rightarrow l(u,cp,\{\overline{cn}\},b,t)$$

COLLECT

$$\frac{\begin{array}{ccc} user \in dom(\sigma) & cn = [\![e_1]\!]_{a \circ \sigma} & l = [\![e_2]\!]_{a \circ \sigma} \\ sd = sensitive(d,l) & (\langle l,cn\rangle \in V \ \vee \ cn \in (cp \cup cm)) & \end{array}}{\begin{array}{c} o(a,(\sigma,(\textbf{collect}(e_1,e_2,x);s)@V),db)\ l(u,cp,cm,b,t) \\ \xrightarrow{\text{“}d\text{”}} o(a,(\sigma[x \mapsto sd],s@V),db)\ l(u,cp,cm,b,t) \end{array}}$$

DELETE

$$\{l(u,cp,cm,b,t)\ cfg\} \rightarrow \{del(l,cfg)\}$$

ERROR-COLLECT

$$\frac{\begin{array}{cc} cn = [\![e_1]\!]_{a \circ \sigma} & l = [\![e_2]\!]_{a \circ \sigma} \\ user \notin dom(\sigma) \vee \neg comply(l,cn,cfg,V) & \end{array}}{\{o(a,(\sigma,(\textbf{collect}(e_1,e_2,x);s)@V,db)\ cfg\} \xrightarrow{\text{“}d\text{”}} \{errorC(o,cn)\ cfg\}}$$

Figure I.5: Rewrite rules for user interactions.

expresses that a policy with identifier $l$ is found in the configuration. To ensure the correct application of this predicate, the rule pattern matches over the global configuration. Note that NO-OPT-IN does not throw an error; doing so would require defining sanity checks for **opt-in** to avoid runtime errors. Instead, errors are triggered in subsequent commands if they try to use data for purposes associated to the missing contracts. In OPT-OUT, a mutable contract $cn$ is removed from the policy $l$.

Data can be collected from the user under a given contract. In COLLECT, the user provides data $d$ if the contract $cn$ is compliant with the policy $l$. The data is paired with the policy and stored in the local substitution $\sigma$. In ERROR-COLLECT, a data collection error is triggered when no user is logged-in or the *comply* predicate is false.

Time advancing is captured by TICK. It applies to global configurations and is always enabled to reflect that GDPR compliance is independent of execution speed. The function *dec* decrements each policy's timestamp by one. When a policy's timestamp is one and TICK fires, *dec* deletes the policy from *cfg* and associated sensitive data from the objects' databases. Rule DELETE captures that a user can request policy deletion at any time, and the policy is deleted from the configuration. The function *del* deletes the data associated with $l$ from the objects' databases. The functions *dec* and *del* are formally defined in Appendix I.A.1.

**Storage, deletion, and scopes** Figure I.6 presents rules for storage, retention, and conditional constructs. Sensitive data is stored in the object's database in STORE if the object's contract complies to the policy $l$ and data storage is allowed (i.e., $b = \mathit{true}$). Expired data can never be stored, so policy compliance is checked in STORE regardless of the compliance scope. Rule NO-STORE applies when policy compliance does not hold or data storage is not allowed (the latter is captured by the negated auxiliary predicate $\mathit{dataStorage}(l, \mathit{cfg})$).

Data from the database can be fetched to local variables in RETRIEVE, which selects the success branch $s$ when the object's contract complies to the associated policy $l$. Otherwise the **else**-branch is selected in NO-RETRIEVE. Data can never be retrieved with an expired policy, which is reflected by the empty scope in the *comply* predicate in NO-RETRIEVE. Note that an object storing data cannot retrieve the data if the policy is deleted or consent is withdrawn.

Policy compliance is checked dynamically in IF-CONSENT. If the contract $cn$ complies to the policy $l$, the compliance scope is extended and the **if**-branch is selected. Here, *cScope* marks the scope's end. Otherwise, the scope is unchanged and the **else**-branch is selected in NO-CONSENT. Rule CLOSE-SCOPE reduces the compliance scope at scope's end.

Compliance between expressions and a contract is checked in IF-COMPLY, where policies are extracted from the evaluated expressions and added to the compliance scope and execution continues with the **if**-branch before closing the scope. Otherwise, NO-COMPLY selects the **else**-branch. The function $\mathit{policyIn}(\overline{v})$ returns a list of policies from the sensitive data in $\overline{v}$ and $\mathit{pairs}(\mathit{ls}, cn)$ pairs

STORE

$$\frac{sd = [\![e]\!]_{a\circ\sigma} \qquad sd = sensitive(d, l) \qquad cn = a(cnThis) \qquad cn \in (cp \cup cm) \qquad b = true}{o(a, (\sigma, (\textbf{store}(k, e)\ \textbf{else}\{s\}; s')@V), db)\ \ l(u, cp, cm, b, t) \rightarrow o(a, (\sigma, s'@V), db[k \mapsto sd])\ \ l(u, cp, cm, b, t)}$$

NO-STORE

$$\frac{\neg dataStorage(l, cfg)\ \vee\ \neg comply(l, cn, cfg, \emptyset) \qquad sensitive(d, l) = [\![e]\!]_{a\circ\sigma} \qquad cn = a(cnThis)}{\{cfg\ o(a, (\sigma, (\textbf{store}(k, e)\ \textbf{else}\{s\}; s')@V), db)\} \rightarrow \{cfg\ o(a, (\sigma, (s; s')@V), db)\}}$$

RETRIEVE

$$\frac{sd = db(k) \qquad sd = sensitive(d, l) \qquad x \in dom(\sigma) \qquad cn = a(cnThis) \qquad cn \in (cp \cup cm)}{o(a, (\sigma, (\textbf{retrieve}(k, x)\{s\}\ \textbf{else}\{s'\}; s'')@V), db)\ \ l(u, cp, cm, b, t) \rightarrow o(a, (\sigma[x \mapsto sd], s; s''@V), db)\ \ l(u, cp, cm, b, t)}$$

NO-RETRIEVE

$$\frac{x \notin dom(\sigma)\ \vee\ k \notin dom(db)\ \vee\ cn = a(cnThis) \qquad sensitive(d, l) = db(k) \qquad \neg comply(l, cn, cfg, \emptyset)}{\{cfg\ o(a, (\sigma, (\textbf{retrieve}(k, x)\{s\}\ \textbf{else}\{s'\}; s'')@V), db)\} \rightarrow \{cfg\ o(a, (\sigma, (s'; s'')@V), db)\}}$$

IF-CONSENT

$$\frac{cn = [\![e_1]\!]_{a\circ\sigma} \qquad l = [\![e_2]\!]_{a\circ\sigma} \qquad cn \in (cp \cup cm)}{o(a, (\sigma, (\textbf{if-consent}(e_1, e_2)\{s\}\ \textbf{else}\{s'\}; s'')@V), db)\ \ l(u, cp, cm, b, t) \rightarrow o(a, (\sigma, (s; cScope(\langle l, cn\rangle); s'')@V \cup \{\langle l, cn\rangle\}), db)\ \ l(u, cp, cm, b, t)}$$

IF-COMPLY

$$\frac{\overline{v} = [\![\overline{e}]\!]_{a\circ\sigma} \quad ls = policyIn(\overline{v}) \quad comply(ls, cn, cfg, \emptyset) \quad S = pairs(ls, cn)}{\{cfg\ o(a, (\sigma, (\textbf{if-comply}(cn, \overline{e})\{s\}\ \textbf{else}\{s'\}; s'')@V), db)\} \rightarrow \{cfg\ o(a, (\sigma, (s; cScope(S); s'')@V \cup S), db)\}}$$

NO-COMPLY

$$\frac{\overline{v} = [\![\overline{e}]\!]_{a\circ\sigma} \qquad ls = policyIn(\overline{v}) \qquad \neg comply(ls, cn, cfg, \emptyset)}{\{cfg\ o(a, (\sigma, (\textbf{if-comply}(cn, \overline{e})\{s\}\textbf{else}\{s'\}; s'')@V, db)\} \rightarrow \{cfg\ o(a, (\sigma, (s'; s'')@V), db)\}}$$

CLOSE-SCOPE

$$o(a, (\sigma, (cScope(S); s)@V), db) \rightarrow o(a, (\sigma, s@V \backslash S), db)$$

NO-CONSENT

$$\frac{\neg comply(l, cn, cfg, \emptyset \qquad cn = [\![e_1]\!]_{a\circ\sigma} \qquad l = [\![e_2]\!]_{a\circ\sigma})}{\{cfg\ o(a, (\sigma, (\textbf{if-consent}(e_1, e_2)\{s\}\ \textbf{else}\{s'\}; s'')@V), db)\} \rightarrow \{cfg\ o(a, (\sigma, (s'; s'')@V), db)\}}$$

Figure I.6: Rewrite rules for data storage, deletion and scopes.

each policy in the list of policies *ls* with the contract *cn* and returns the set of policy-contract pairs.

**Standard Rules** Figure I.7 presents the rules for standard statements, augmented to dynamically check compliance. Sensitive data can be assigned to local variables by ASSIGN-LOCAL if compliance between the object's contract and the policy is guaranteed by the scope. Otherwise, ERROR-ASSIGN triggers an error. Since sensitive data cannot be assigned to fields (see Sec. I.3.1), ERROR-ASSIGN-FIELD triggers an error. Object creation initialises an object with an empty database in NEW, but triggers an error in ERROR-NEW if the constructor's actual parameters contain sensitive data. The function $atts(C, \overline{v}, o)$ returns the initial substitution $a$ for fields, where the formal parameters are bound to $\overline{v}$, the reserved variable *this* to the identifier $o$, and the reserved variable *cnThis* to $contract(P, o)$, where $P$ is the purpose implemented by $C$.

In CALL, method calls may only occur if the called method's actual parameter values are compliant with the callee's contract (accessed via an auxiliary function $contract(o')$, where $o'$ is the callee). In this case, a message is sent to the callee. This message includes the caller's contract $a(cnThis)$, such that the callee can check compliance before returning the method's result to the caller. The caller is blocked until it receives the result. Since the waiting time is unknown, the compliance scope is emptied and data will need to be rechecked once computation resumes. If the actual parameter values are not compliant with the contract, ERROR-CALL triggers a data usage error. The callee receives the message in CALLEE-INVC. The function $class(o)$ returns the class of object $o$ and $bind(o, C, m, \overline{v}, o', cn)$ creates a new process $(\sigma, s@\emptyset)$, where the reserved variable *caller* is bound to $o'$ in $\sigma$, the reserved variable *cnCaller* to the caller's contract *cn*, and the formal parameters to $\overline{v}$. Moreover, $s$ is the method body of $m$ in the class $C$ and the compliance scope is empty. Data may have expired and needs to be checked using the appropriate conditional constructs in the method body.

Upon method completion with sensitive data as the return value, RETURN sends a completion message to the caller if the caller's contract, which is stored in the callee's local variable *cnCaller*, complies with the policy $l$. Otherwise, ERROR-RETURN triggers a data usage error. Observe that the error can be avoided by testing the caller's contract; e.g., **`if-comply`**$(cnCaller, e)\{$**`return`**$(e)\}$. The caller receives the completion message and gets unblocked in GET-DATA. The data might have expired before the message is received, potentially triggering an error in subsequent statements.

Figure I.8 presents self calls, which are supported by SELF-CALL. (Observe that cyclic call chains give rise to deadlock in our semantics; these could be handled with scheduling messages by adapting the pattern of SELF-CALL to blocked objects with incoming calls.) In SELF-CALL, the compliance scope is emptied so the calling process will need to recheck compliance when it resumes. The new process, also with an empty compliance scope, ends with a *cont(n)* statement and the old process is wrapped in a scheduling message. These

ASSIGN-LOCAL

$$\frac{cn = a(cnThis) \quad x \in dom(\sigma)\ \langle l, cn\rangle \in V \ \vee\ cn \in (cp \cup cm) \quad sd = [\![e]\!]_{a\circ\sigma}\ sd = sensitive(d, l)}{o(a, (\sigma, (x := e; s)@V), db)\ l(u, cp, cm, b, t) \rightarrow o(a, (\sigma[x \mapsto sd], s@V), db)\ l(u, cp, cm, b, t)}$$

ERROR-ASSIGN-FIELD

$$\frac{x \in dom(a) \quad sensitive(d, l) = [\![e]\!]_{a\circ\sigma}}{o(a, (\sigma, (x := e; s)@V), db) \rightarrow error(o)}$$

ERROR-ASSIGN

$$\frac{x \in dom(\sigma) \quad sensitive(d, l) = [\![e]\!]_{a\circ\sigma} \quad cn = a(cnThis) \quad \neg comply(l, cn, cfg, V)}{\{o(a, (\sigma, (x := e; s)@V), db)\ cfg\} \rightarrow \{errorU(o, cn)\ cfg\}}$$

NEW

$$\frac{\overline{d} = [\![\overline{e}]\!]_{a\circ\sigma} \qquad a' = atts(C, \overline{d}, o')\ \ fresh(o') \qquad contract(P, o') = a'(cnThis)}{o(a, (\sigma, (x := \textbf{new}\ C(\overline{e}); s)@V), db) \rightarrow o(a, (\sigma, (x := o'; s)@V), db)\ o'(a', idle, [])}$$

ERROR-NEW

$$\frac{sd \in [\![\overline{e}]\!]_{a\circ\sigma} \qquad sd = sensitive(d, l)}{o(a, (\sigma, (x := \textbf{new}\ C(\overline{e}); s)@V), db) \rightarrow error(o)}$$

ERROR-CALL

$$\frac{o' = [\![e]\!]_{a\circ\sigma} \quad \overline{v} = [\![\overline{e}]\!]_{a\circ\sigma} \quad cn = contract(o') \quad ls = policyIn(\overline{v}) \quad \neg comply(ls, cn, cfg, V)}{\{o(a, (\sigma, (x := e.m(\overline{e}); s)@V), db)\ cfg\} \rightarrow \{errorU(o, cn)\ cfg\}}$$

CALL

$$\frac{o' = [\![e]\!]_{a\circ\sigma}\ \overline{v} = [\![\overline{e}]\!]_{a\circ\sigma}\ cn = contract(o') \quad ls = policyIn(\overline{v}) \quad comply(ls, cn, cfg, V)}{\{o(a, (\sigma, (x := e.m(\overline{e}); s)@V), db)\ cfg\} \rightarrow \{o(a, (\sigma, (x := m?; s)@\emptyset), db)\ m(\overline{v}, o', o, a(cnThis))\ cfg\}}$$

RETURN

$$\frac{sd = [\![e]\!]_{a\circ\sigma} \quad sd = sensitive(d, l) \quad o' = \sigma(caller) \quad cn = \sigma(cnCaller) \quad \langle l, cn\rangle \in V \ \vee\ cn \in (cp \cup cm)}{o(a, (\sigma, \textbf{return}\ e\ @V), db)\ l(u, cp, cm, b, t) \rightarrow o(a, idle, db)\ l(u, cp, cm, b, t)\ com(sd, o')}$$

GET-DATA

$$o(a, (\sigma, (x := m?; s)@V), db)\ com(v, o) \rightarrow o(a, (\sigma[x \mapsto v], s@V), db)$$

CALLEE-INVC

$$\frac{(\sigma, s@\emptyset) = bind(o, C, m, \overline{v}, o', cn)\ C = class(o) \quad \sigma(caller) = o' \quad \sigma(cnCaller) = cn}{\{o(a, idle, db)\ m(\overline{v}, o, o', cn)\ cfg\} \rightarrow \{o(a, (\sigma, s@\emptyset), db)\ cfg\}}$$

ERROR-RETURN

$$\frac{sensitive(d, l) = [\![e]\!]_{a\circ\sigma}\ o' = \sigma(caller)\ cn = \sigma(cnCaller)\ \neg comply(l, cn, cfg, V)}{\{o(a, (\sigma, \textbf{return}\ e\ @V), db)\ cfg\} \rightarrow \{errorU(o, cn)\ cfg\}}$$

Figure I.7: Rewrite rules for standard statements.

SELF-CALL

$$\frac{o = [\![e]\!]_{a\circ\sigma} \quad \overline{v} = [\![\overline{e}]\!]_{a\circ\sigma} \quad (\sigma', s'@\emptyset) = \mathit{bind}(o, C, m, \overline{v}, o, a(\mathit{cnThis})) \quad \mathit{fresh}(n) \quad \mathit{ls} = \mathit{policyIn}(\overline{v}) \quad \mathit{comply}(\mathit{ls}, a(\mathit{cnThis}), \mathit{cfg})}{\{o(a, (\sigma, (x := e.m(\overline{e}); s)@V), db)\ \mathit{cfg}\} \rightarrow \{o(a, (\sigma', (s'; \mathit{cont}(n))@\emptyset), db)\ n(\sigma, (x := m?; s)@\emptyset)\ \mathit{cfg}\}}$$

SELF-RETURN

$$\frac{a(\mathit{cnThis}) \in (\mathit{cp} \cup \mathit{cm}) \quad o = \sigma(\mathit{caller}) \quad v = [\![e]\!]_{a\circ\sigma}}{o(a, (\sigma, (\textbf{return}\ e; \mathit{cont}(n))@V), db)\ n(\sigma', (x := m?; s)@V')\ l(u, b, \mathit{cp}, \mathit{cm}, t) \rightarrow o(a, (\sigma'[x \mapsto v], s@V'), db)\ l(u, b, \mathit{cp}, \mathit{cm}, t)}$$

ERROR-SELF-RETURN

$$\frac{\neg \mathit{comply}(l, \mathit{cn}, \mathit{cfg}, V) \quad \mathit{cn} = a(\mathit{cnThis})\ \mathit{sensitive}(d, l) = [\![e]\!]_{a\circ\sigma} \quad o = \sigma(\mathit{caller})}{\{o(a, (\sigma, (\textbf{return}\ e; \mathit{cont}(n))@V), db)\ \mathit{cfg}\} \rightarrow \{\mathit{errorU}(o, \mathit{cn})\ \mathit{cfg}\}}$$

ERROR-SELF-CALL

$$\frac{\neg \mathit{comply}(\mathit{ls}, a(\mathit{cnThis}), \mathit{cfg}, V)\ o = [\![e]\!]_{a\circ\sigma} \quad \overline{v} = [\![\overline{e}]\!]_{a\circ\sigma} \quad \mathit{ls} = \mathit{policyIn}(\overline{v})}{\{o(a, (\sigma, (x := e.m(\overline{e}); s)@V), db)\ \mathit{cfg}\} \rightarrow \{\mathit{errorU}(o, \mathit{cn})\ \mathit{cfg}\}}$$

Figure I.8: Rewrite rules for standard statements part 2.

are matched to resume execution in SELF-RETURN, provided that the object's contract, found in the field *cnThis*, complies with the policy. Otherwise, ERROR-SELF-RETURN triggers a data usage error.

The initial state is derived from the main block $\textbf{main}\{\overline{T\ x}; s\}$ by creating an object $\mathit{ob(main)}([\mathit{cnThis} \mapsto \mathit{contract}(\mathit{main}, \mathit{ob(main)})], ([], s@\emptyset), [])$, with identity *ob(main)* and contract $\mathit{contract}(\mathit{main}, \mathit{ob(main)})$. Note that this contract must be added to created policies so that the $\mathit{ob}(\mathit{main})$ object can access sensitive data. In the object, the local substitution and the database are empty, and the active process $([], s@\emptyset)$ corresponds to the activation of **main**'s statements $s$.

## I.4 Correctness

DPL's operational semantics gives rise to a transition system, where states are configurations and transitions correspond to rule applications. There are infinitely many initial states reflecting the starting configurations of infinitely many programs. Moreover, a program may give rise to a nonterminating application of rules and, therefore, also infinitely many states.

We reason about this infinite state transition system and prove that DPL programs cannot lead to GDPR violations with respect to the requirements given in Section I.2.1. Namely, we formalize properties that ensure purpose limitation,

| Auxiliary formulas | Explanation |
|---|---|
| $use(o, \langle d, l \rangle, cn)$ | the object $o$ uses the data $\langle d, l \rangle$ for a purpose associated with the contract $cn$ |
| $complyTo(l, cn)$ | the policy $l$ exists and the contract $cn$ complies to the policy |
| $checked\text{-}scope(o, \langle l, cn \rangle)$ | the pair $\langle l, cn \rangle$ is in $o$'s compliance scope |
| $noExecIn(o)$ | the current execution step is not in $o$ |
| $errorU(o, cn)$ | a data usage error associated with the contract $cn$ occurred in the object $o$ |
| $errorC(o, cn)$ | a data collection error in $o$ when consent associated with the contract $cn$ is not granted |
| $collect(o, cn, l)$ | $o$ is executing a **collect**$(cn, l, x)$ statement |
| $optedIn(l, cn)$ | the contract $cn$ is added to the policy $l$ |
| $optedOut(l, cn)$ | the contract $cn$ is removed from the policy $l$ |
| $expired(l)$ | the policy $l$ is deleted from the configuration |
| $dbDel(l)$ | sensitive data associated with the policy $l$ is deleted from databases |
| $deleted(l)$ | the policy $l$ is deleted |

Table I.2: Explanation of predicates.

consent, the right to withdraw consent, storage limitation, and the right to be forgotten. Formal definitions and proofs are given in Appendix I.A.1.

In our formalization, we define a trace as a sequence of configuration and action pairs $(cfg_0, R_0), (cfg_1, R_1), \cdots$, where an action is the name of the rule that is fired at the configuration (i.e., $cfg_i \rightarrow cfg_{i+1}$ by applying the rule $R_i$). To formalize and reason about temporal properties of DPL programs, we use linear temporal logic (LTL) [4, 49] with the standard temporal operators: $\bigcirc$ (next), $\Box$ (always), $\Diamond$ (sometime), and *Until* and $W$, which are the strong and weak until operators respectively. The notation $\models \varphi$ denotes that the LTL property $\varphi$ holds for all traces of our transition system.

Table I.2 shows the state formulas we use and their informal interpretation. Note that some of our definitions state that predicates must eventually hold and for this to be the case we require a fair transition system. Since the rules TICK, DELETE, and OPT-OUT can fire infinitely often and at anytime, we specify strong fairness for our transition system, where if a rule is enabled infinitely often, then it fires infinitely often. We express strong fairness for our transition system as follows.

$$fair = \Box\Diamond enabled_1 \Rightarrow \Box\Diamond fired_1 \ \wedge \ \cdots \ \wedge \ \Box\Diamond enabled_i \Rightarrow \Box\Diamond fired_i.$$

In this formula, the index $i$ ranges over the names of our rules, $enabled_i$ is true (i.e., satisfied at a given point in a trace) when the rule $i$ is enabled namely, the premises of the rule are true and the left-hand-side of the rule matches the current configuration, and $fire_i$ is true when the rule $i$ is fired. We shall assume strong fairness when proving all our properties.

The following property $\mathscr{P}$ formalizes purpose limitation, where compliance is checked for any data usage. Namely, an object cannot use data for a purpose that is not compliant with the policy, and an error arises if the object attempts a non-compliant usage. In DPL, the only statements using data are assignments, calls, and **`return`**-statements. The *use* formula is true if one of these three statements is the first statement in the active process of an object. The property $\mathscr{P}_1$ says that if whenever *use* is true, then *checked-scope* is true (i.e., if compliance is checked prior to the usage in the appropriate conditional construct), then a data usage error never arises. Note that $\mathscr{P}_1$ holds regardless of whether the *complyTo* formula is true or false. The property $\mathscr{P}_2$ says that if data is used and the formulas *complyTo* and *checked-scope* are false, then in the next step, execution does not continue in the object until a data usage error arises or the user opts in.

$$
\begin{aligned}
&\mathscr{P} = \mathscr{P}_1 \ \wedge \ \mathscr{P}_2 \\
&\mathscr{P}_1 = \forall o, d, l, cn. \\
&\quad \Box(\mathit{use}(o, \langle d, l\rangle, cn) \Rightarrow \mathit{checked\text{-}scope}(o, \langle l, cn\rangle)) \\
&\quad \Rightarrow \Box\neg \mathit{errorU}(o, cn) \\
&\mathscr{P}_2 = \forall o, d, l, cn. \\
&\quad \Box((\mathit{use}(o, \langle d, l\rangle, cn) \ \wedge \ \neg \mathit{complyTo}(l, cn) \\
&\quad \wedge \ \neg \mathit{checked\text{-}scope}(o, \langle l, cn\rangle)) \\
&\quad \Rightarrow \bigcirc(\mathit{noExecIn}(o) \ \mathit{Until} \ (\mathit{errorU}(o, cn) \ \vee \ \mathit{optedIn}(l, cn))))
\end{aligned}
$$

**Theorem I.4.1** (Purpose limitation)**.** *The property $\mathscr{P}$ holds for all traces of our transition system.*

The proof of this theorem and all theorems in this section can be found in Appendix I.A.1.

The following property $\mathscr{C}$ formalizes that data is collected only if consent has been granted. The formula *collect* is true if there is a **`collect`**$(cn, l, x)$ statement in the active process of an object as the first statement. The property $\mathscr{C}_1$ says that if **`collect`**$(cn, l, x)$ is always in a conditional construct that checks compliance between a policy $l$ and a contract $cn$, then a data collection error never arises. The property $\mathscr{C}_2$ says that when collecting data, if the formulas *complyTo* and *checked-scope* are false, then in the next step, execution does not continue in the object until a data collection error arises or the user opts in.

$$
\begin{aligned}
&\mathscr{C} = \mathscr{C}_1 \ \wedge \ \mathscr{C}_2 \\
&\mathscr{C}_1 = \forall o, l, cn. \\
&\quad \Box(\mathit{collect}(o, l, cn) \Rightarrow \mathit{checked\text{-}scope}(o, \langle l, cn\rangle)) \\
&\quad \Rightarrow \Box\neg \mathit{errorC}(o, cn) \\
&\mathscr{C}_2 = \forall o, l, cn. \\
&\quad \Box((\mathit{collect}(o, l, cn) \wedge \\
&\quad \neg \mathit{complyTo}(l, cn) \ \wedge \ \neg \mathit{checked\text{-}scope}(o, \langle l, cn\rangle)) \\
&\quad \Rightarrow \bigcirc(\mathit{noExecIn}(o) \ \mathit{Until} \ (\mathit{errorC}(o, cn) \ \vee \ \mathit{optedIn}(l, cn))))
\end{aligned}
$$

**Theorem I.4.2** (Consent)**.** *The property $\mathscr{C}$ holds for all traces of our transition system.*

The following property $\mathscr{W}$ formalizes the right to withdraw consent: A user can withdraw consent, and the corresponding purpose is removed from the policy, which prevents subsequently using the data for that purpose. The formula $optedOut(l, cn)$ is true when the contract $cn$ is removed from the policy $l$. The formula $optedIn(l, cn)$ is true when the contract $cn$ is added to the policy $l$. The property $\mathscr{W}$ says that when $optedOut(l, cn)$ is true, then compliance with respect to the policy and the contract remains false until $optedIn(l, cn)$ is true (if it ever becomes true, hence we use LTL's weak-until operator).

$$\begin{aligned}&\mathscr{W} = \forall l, cn.\\ &\quad \Box(optedOut(l, cn) \Rightarrow (\neg complyTo(l, cn)\ W\ optedIn(l, cn)))\end{aligned}$$

**Theorem I.4.3** (Right to withdraw consent)**.** *The property $\mathscr{W}$ holds for all traces of our transition system.*

The following property $\mathscr{S}$ formalizes storage limitation: Data is deleted from the objects' databases when the deadline for data deletion arrives. It says that if the formula $expired(l)$ is true (due to policy expiration or the DELETE rule), then data with that policy is deleted from the objects' databases. In the rules TICK and DELETE, policy deletion and data deletion are specified using functions and equations. Thus when a policy is deleted, its data is deleted in the same state as well.

$$\mathscr{S} = \forall l.\ \Box(expired(l) \Rightarrow dbDel(l))$$

**Theorem I.4.4** (Storage limitation)**.** *The property $\mathscr{S}$ holds for all traces of our transition system.*

The following property $\mathscr{F}$ formalizes the right to be forgotten: A user can request to delete her data, and the data is then deleted from the objects' databases. The predicate $deleted(l)$ is true when the policy $l$ is deleted by the rule DELETE. The property $\mathscr{F}$ says that when a policy is deleted, the formula *expired* is true for that policy. Moreover, by Theorem I.4.4, data with that policy is deleted from databases.

$$\mathscr{F} = \forall l.\ \Box(deleted(l) \Rightarrow expired(l))$$

**Theorem I.4.5** (Right to be forgotten)**.** *The property $\mathscr{F}$ holds for all traces of our transition system.*

As mentioned, attempted GDPR violations do not succeed; they instead produce runtime errors. These errors can be systematically avoided by the following *hygienic measures*, which amount to good GDPR practice: i) all methods that use personal data are specified in the corresponding purpose declarations; ii) for each purpose declaration, the corresponding consent statement and contract are specified; iii) the pattern in Section I.3.2 is used

for data collection, which requires a logged-in user, a privacy policy for the user, opt-in options, and the appropriate **if-consent** construct prior to data collection; and iv) the construct **if-comply**$(cn, \overline{e})$ is used prior to the commands that use sensitive data in $\overline{e}$. That these hygienic measures are sufficient follows by careful inspection of the rules, and we also confirm this for concrete programs by model checking hygienic and non-hygienic programs (see Appendix I.A.3 for some examples).

## I.5 Maude formalization

We have formalized DPL's operational semantics in Maude [18]. This yields a prototype environment for simulating DPL programs and thereby provides us some confidence in our rules. Maude supports multiset rewriting logic, which we use to model the non-deterministic behavior of our system, where enabled rules can be interleaved at any point after each rewrite. We use a user object to non-deterministically give "*yes*" or "*no*" input to **opt-in** statements.

Our Maude formalization provides a prototype verification environment for DPL programs. It can be used to check all our properties on finite-state programs, such as the program given in Fig. I.3. (Technically, the programs may be infinite state, e.g., involve data from unbounded domains, provided only finitely many states are reachable.) We also validate statements about "hygienic programs" given in Section I.4. We check $\mathscr{P}_1$, $\mathscr{C}_1$, $\mathscr{W}$, $\mathscr{S}$, and $\mathscr{F}$ for the hygienic program given in Fig. I.3, where all the usage constructs are protected with conditional constructs. Maude verifies that errors never arise and all specified properties hold. We checked the properties $\mathscr{P}_2$ and $\mathscr{C}_2$ concerning programs raising errors on the program where different combinations of the conditional constructs are removed. Maude verifies that the corresponding errors arise and the properties hold. We present our results with Maude in Appendix I.A.3.

## I.6 Related Work

Purpose-based access control mechanisms [11, 12] have been proposed to control access to data in databases based on intended purpose information associated with the data. Users must state their access purposes when requesting data and can access the data if their stated purposes comply with the data's intended purpose. Both kinds of purposes are organized hierarchically, and compliance is defined based on a partial-order relation. In contrast, we enforce GDPR requirements using programming language constructs and runtime checks. In DPL, intended purposes are added to policies when users give consent, and policies can change over time when data subjects consent to new purposes or withdraw their consent.

*Privacy by Design (PbD)* [16] is a framework that introduces principles that should be considered when designing a system architecture. Schneider [72] explains that since a model specification can be very different from the implementation, PbD, by itself, cannot in general guarantee privacy unless it also encompasses the implementation. We believe that languages like DPL have

an important role to play in building privacy-by-design systems as DPL directly supports many of the principles espoused there. For example, it is a proactive approach that provides privacy by default.

Another design-oriented approach is [7], where GDPR compliance is checked at the design level. Business processes represent one or more purposes, and formal models of inter-process communication identify data collection and data usage points [7]. In order to identify purposes and data usage points, we build on the approach proposed in [7]. However, instead of business processes, the methods that implement a process are grouped together in a purpose declaration. Moreover, we provide language support for policy enforcement, whereas [7] only supports data protection through audits.

Researchers have used information-flow analysis [70, 71] to check privacy policy compliance in programs. There, types are annotated with privacy policy labels, and a notion of policy compliance is defined. We also track the flow of sensitive data, where left-hand-side expressions get the policy identity of right-hand-expressions. We expand upon the most closely related work here in the following.

In [56], the authors propose a decentralized label model for Jif to enforce role-based access control in programs. Jif principals represent entities with specific roles, which have a hierarchical structure. Program variables are annotated with policy labels, where a label contains the principal that owns the data and a set of readers who can read the data. Jif's type checking enforces information-flow control and protects principals' privacy. Moreover, a principal may declassify the label of the data that it owns. Declassification of roles requires runtime checking to determine whether a process is authorized to declassify data. However, most of a program can be certified statically with no overhead. In contrast, we enforce richer data protection policies, as required by the GDPR, such as the necessity of providing consent, the right to withdraw consent, purpose limitation, storage limitation, and the right to be forgotten. GDPR temporal requirements, where users can withdraw consent or data is automatically deleted when deadlines arrive cannot be enforced statically. In DPL, we enforce these requirements by runtime checking.

In [32], the authors enforce purpose-based and storage-based restrictions in Jif. Jif's principals represent purposes, ordered hierarchically. Data is annotated with the principal that owns the data, representing the purpose for which the data is collected. Methods that are needed for a purpose are annotated with the corresponding principal. By means of Jif's type checking, compile-time errors arise if data is used for non-compliant purposes. Similarly, for enforcing retention restrictions, Jif's principals represent retention labels. In [32], storage restrictions are limited to retention labels (in the sense of P3P), intended purposes are fixed, and access purposes are associated by the programmer. In DPL, users' consent determines the intended purposes in policies, and access purposes are automatically associated with created objects. Moreover, in DPL, instead of policies themselves, references to policies are attached to data, and thus if a policy changes due to the user actions or the passage of time, then this change is automatically enforced on any usage or storage of data with the policy reference. In addition, this requires less storage overhead at runtime and we can also enforce

deletion policies.

In [74], the authors propose a static approach to check privacy policy compliance in Bing, where privacy policies are specified in LEGALEASE, and GROK maps data types in a code to policies and tracks the flow of information. By taking a static approach, there are no runtime overheads. However, their approach cannot be used to enforce the GDPR requirements such as consent, the right to withdraw consent, the right to be forgotten, and temporal requirements for data deletion, where a runtime approach would be required.

## I.7 Conclusion

We have presented DPL, a programming language designed for data protection with provable guarantees. DPL and our Maude-based simulation environment are prototypical and allow us to simulate programs and experiment with our new language features. Our initial experience supports the thesis that custom language support can play an important role in building systems meeting strict data protection requirements like those of the GDPR.

Our work constitutes a first significant step in building a robust, usable language with formal GDPR guarantees. This work could be strenghtened in several ways, which suggest interesting directions for future work: (1) Large-scale case studies are needed to better assess the language's usability and further evaluate the runtime overheads involved. (2) For stronger correctness results, our pen-and-paper proofs (in Appendix A) could be further formalized in a theorem prover such as Coq or Isabelle. (3) A type and effect system for DPL could be used to enforce the correct use of scopes, with an associated type preservation theorem. This would make programming in DPL easier by eliminating runtime errors for well-typed programs. We have also highlighted other possibilities for future work in this paper's body. This includes defining fine-grained compliance scopes in programs, developing a more permissive solution for binary and general operations on data items with different policies, and, finally, building real language support.

**Acknowledgements.** This work was partly funded by the Research Council of Norway through IoTSec (project no. 248113).

## I.A Appendix

### I.A.1 Proofs

We prove that the properties in Section I.4 hold for all strongly fair runs of our transition system. To define the fairness and some of our formulas, we must track the current rule that is executed and the object's identity that the rule is applied to. We define a trace $\tau$ as a sequence of tuples, such as $\tau = (cfg_0, R_0, Id_0), (cfg_1, R_1, Id_1), \cdots$, that in addition to the configuration $cfg$ and the rule label $R$, also tracks the identity $Id$ of the object, which can be an object's identity or *other*, when the rules TICK, DELETE, and OPT-OUT are

fired. We define formulas on a state denoted by $\phi(cfg)$, or on tuples, denoted by $\phi(cfg, R, Id)$ as needed. We formalize the formulas in Table I.2 as follows.

$$\begin{aligned}
&use(o, sensitive(d, l), cn)(cfg) = \exists a, \sigma, x, e, e', \overline{e}, m, s, s', V, db, o'.\\
&\quad o(a, (\sigma, (s; s')@V), db) \in cfg \wedge\\
&\quad (s = x := e \ \wedge \ sensitive(d, l) = [\![e]\!]_{a \circ \sigma} \ \wedge \ cn = a(cnThis)) \vee\\
&\quad (s = \textbf{return}(e) \ \wedge \ sensitive(d, l) = [\![e]\!]_{a \circ \sigma} \ \wedge \ o' = \sigma(caller)\\
&\qquad \wedge \ cn = \sigma(cnCaller)) \vee\\
&\quad (s = \textbf{return}(e) \ \wedge \ sensitive(d, l) = [\![e]\!]_{a \circ \sigma} \ \wedge \ o = \sigma(caller)\\
&\qquad \wedge \ cn = \sigma(cnThis)) \vee\\
&\quad (s = x := e'.m(\overline{e}) \ \wedge \ sensitive(d, l) \in [\![\overline{e}]\!]_{a \circ \sigma} \ \wedge \ o' = [\![e']\!]_{a \circ \sigma}\\
&\qquad \wedge \ cn = contract(o')) \vee\\
&\quad (s = x := e'.m(\overline{e}) \ \wedge \ sensitive(d, l) \in [\![\overline{e}]\!]_{a \circ \sigma} \ \wedge \ o = [\![e']\!]_{a \circ \sigma}\\
&\qquad \wedge \ cn = a(cnThis))
\end{aligned} \tag{I.1}$$

$$\begin{aligned}
&complyTo(l, cn)(cfg) = \exists u, cp, cm, b, t.\\
&\quad l(u, cp, cm, b, t) \in cfg \ \wedge \ cn \in (cp \cup cm)
\end{aligned} \tag{I.2}$$

$$\begin{aligned}
&checked\text{-}scope(o, \langle l, cn\rangle)(cfg) = \exists a, \sigma, s, V, db.\\
&\quad o(a, (\sigma, s@V), db) \in cfg \ \wedge \ \langle l, cn\rangle \in V
\end{aligned} \tag{I.3}$$

$$\begin{aligned}
errorU(o, cn)(cfg) &= errorU(o, cn) \in cfg\\
errorC(o, cn)(cfg) &= errorC(o, cn) \in cfg
\end{aligned} \tag{I.4}$$

$$\begin{aligned}
&collect(o, l, cn)(cfg) = \exists a, \sigma, e_1, e_2, e_3, s, s', V, db.\\
&\quad o(a, (\sigma, (s; s')@V), db) \in cfg \wedge\\
&\quad s = \textbf{collect}(e_1, e_2, e_3) \ \wedge \ cn = [\![e_1]\!]_{a \circ \sigma} \ \wedge \ l = [\![e_2]\!]_{a \circ \sigma}
\end{aligned} \tag{I.5}$$

$$\begin{aligned}
&optedOut(l, cn)(cfg, R, Id) = \exists u, b, cp, cm, t.\\
&l(u, b, cp, cm, t) \in cfg \ \wedge \ R = \text{OPT-OUT}(l) \ \wedge \ cn \notin cm
\end{aligned} \tag{I.6}$$

$$\begin{aligned}
&optedIn(l, cn)(cfg, R, Id) = \exists u, b, cp, cm, t.\\
&\quad l(u, b, cp, cm, t) \in cfg \ \wedge \ R = \text{OPT-IN}(l) \ \wedge \ cn \in cm
\end{aligned} \tag{I.7}$$

$$expired(l)(cfg) = \exists u, b, cp, cm, t. \ l(u, b, cp, cm, t) \notin cfg \tag{I.8}$$

$$\begin{aligned}
&dbDel(l)(cfg) = \exists o, a, p, db, d.\\
&\quad o(a, p, db) \in cfg \ \wedge \ sensitive(d, l) \notin db
\end{aligned} \tag{I.9}$$

$$deleted(l)(cfg, R, Id) = R = \text{DELETE}(l) \tag{I.10}$$

$$noExecIn(o)(cfg, R, Id) = Id \neq o \tag{I.11}$$

In the following, we prove the properties in Section I.4. The initial state of the program $\overline{CL}$ $\overline{PI}$ $\textbf{main}\{\overline{T\ x}; s\}$ is a multiset including the object $ob(main)$, the classes $\overline{CL}$, and the purposes $\overline{PI}$, where $cfg_0 = \{CL\ PI\ ob(main)([cnThis \mapsto$

$contract(main, ob(main))], ([], s@\emptyset), [])\}$.

$$
\begin{aligned}
&\mathscr{P} = \mathscr{P}_1 \ \wedge \ \mathscr{P}_2 \\
&\mathscr{P}_1 = \forall o, d, l, cn. \\
&\quad \square(use(o, sensitive(d, l), cn) \Rightarrow checked\text{-}scope(o, \langle l, cn\rangle)) \\
&\quad \Rightarrow \square\neg errorU(o, cn) \\
&\mathscr{P}_2 = \forall o, d, l, cn. \\
&\quad \square((use(o, sensitive(d, l), cn) \ \wedge \ \neg complyTo(l, cn) \\
&\quad \wedge \ \neg checked\text{-}scope(o, \langle l, cn\rangle)) \\
&\quad \Rightarrow \bigcirc(noExecIn(o) \ Until \ (errorU(o, cn) \ \vee \ optedIn(l, cn))))
\end{aligned}
$$

**Theorem I.A.1** (Purpose limitation)**.** *The property $\mathscr{P}$ holds for all traces of our transition system.*

*Proof.* First, we prove $\mathscr{P}_1$ using a proof by contradiction. Let $\tau$ be a fair trace. The premise of $\mathscr{P}_1$ says that in every state in $\tau$, for all $o$, $d$, $l$, and $cn$, whenever $use(o, sensitive(d, l), cn)$ holds, then $checked\text{-}scope(o, \langle l, cn\rangle)$ holds. To achieve a contradiction, assume that $errorU(o, cn)$ holds in a reachable state in $\tau$ and consider the first such state $cfg'$ that it holds. This cannot be the initial state in $\tau$ (per definition) so there must be a transition from a predecessor state $cfg$ to $cfg'$ adding $errorU(o, cn)$ to the configuration. The only rules that could have added $errorU(o, cn)$ are the following:

- ERROR-ASSIGN: In $o(a, (\sigma, (x := e; s)@V), db)$, let $sensitive(d, l) = [\![e]\!]_{a \circ \sigma}$ and $cn = a(cnThis)$. Since the current program statement in $cfg$ is an assignment, then $use(o, sensitive(d, l), cn)$ holds in this state. Therefore, $checked\text{-}scope(o, \langle l, cn\rangle)$ holds in $cfg$, and hence also $\langle l, cn\rangle \in V$. Thus, in the rule ERROR-ASSIGN, the premise *comply* holds, and this rule cannot fire in $cfg$. Thus, $errorU(o, cn)$ does not hold in $cfg'$.

- ERROR-CALL: In $o(a, (\sigma, (x := e.m(\overline{e}); s)@V), db)$, let $o' = [\![e]\!]_{a \circ \sigma}$, $sensitive(d, l) \in [\![\overline{e}]\!]_{a \circ \sigma}$, and $cn = contract(o')$. Since the current program statement in $cfg$ is a call, then $use(o, sensitive(d, l), cn)$ holds in this state. The rest of this case is identical to the ERROR-ASSIGN case.

- ERROR-SELF-CALL: In $o(a, (\sigma, (x := e.m(\overline{e}); s)@V), db)\}$, let $o = [\![e]\!]_{a \circ \sigma}$, $sensitive(d, l) \in [\![\overline{e}]\!]_{a \circ \sigma}$, and $cn = a(cnThis)$. Since the current program statement in $cfg$ is a self-call, then $use(o, sensitive(d, l), cn)$ holds in this state. The rest of this case is identical to the ERROR-ASSIGN case.

- ERROR-RETURN: In $o(a, (\sigma, \textbf{return}\ e@V), db)$, let $sensitive(d, l) = [\![e]\!]_{a \circ \sigma}$, $o' = \sigma(caller)$, and $cn = \sigma(cnCaller)$. Since the current program statement in $cfg$ is a **return**, then $use(o, sensitive(d, l), cn)$ holds. The rest of this case is identical to the ERROR-ASSIGN case.

- ERROR-SELF-RETURN: In $o(a, (\sigma, (\textbf{return}\ e; cont(n))@V), db)\ n(\sigma', (x := m?; s)@\emptyset)$, let $sensitive(d, l) = [\![e]\!]_{a \circ \sigma}$, $o = \sigma(caller)$, and $cn = a(cnThis)$.

Since the current program statement in *cfg* is a self **return**, then $use(o, sensitive(d, l), cn)$ holds. The rest of this case is identical to the ERROR-ASSIGN case.

In all these cases, we conclude that $\neg errorU(o, cn)$ holds, so $\mathscr{P}_1$ holds.

Next, we prove $\mathscr{P}_2$. Given a fair trace $\tau$, consider any reachable state *cfg* where the premise of $\mathscr{P}_2$ holds, i.e., for some $o$, $d$, $l$, and $cn$, $use(o, sensitive(d, l), cn)$, $\neg checked\text{-}scope(o, \langle l, cn\rangle)$, and $\neg complyTo(l, cn)$ hold. Let $cfg'$ be the next state in $\tau$. We show that there exists a state $cfg''$ after $cfg'$ where $errorU(o, cn) \vee optedIn(l, cn)$ holds, and $noExecIn(o)$ holds for all states from $cfg'$ up to (but not necessarily including) $cfg''$.

For the rest of the proof, we use the fact that $noExecIn(o)$ holds when enabled rules not involving the object $o$ fire. There are four such kinds of rules: 1) TICK, 2) DELETE, 3) OPT-OUT, and 4) all enabled rules that apply to other objects than $o$.

Now, since $use(o, sensitive(d, l), cn)$ holds in *cfg*, there are five possible program statements that can execute within $o$. These correspond to the five disjunctions in the definition of *use* (Equation I.1). We consider the Assignment case below and four other cases for Call, Self-Call, Return, and Self-Return are analogous.

For the assignment case, consider how execution can progress after an assignment, from the next state $cfg'$. Either a program statement in the object $o$ fires, or one of the four kinds of rules fire not involving $o$. This yields the following two cases: 1) The only enabled rule involving $o$ is ERROR-ASSIGN, which produces a state $cfg''$ where $errorU(o, cn)$ holds. 2) For the other four kinds of rules that can be enabled, as discussed previously, (i) for all of these rules, ERROR-ASSIGN remains enabled except (ii) when the rule OPT-IN, enabled in another object $o'$, fires and thereby adds the contract $cn$ to the policy $l$. If 2(i) holds, then $noExecIn(o)$ holds in the successor state. We can only repeat this case finitely often since ERROR-ASSIGN remains enabled, and by fairness, it must eventually fire, leading to state $cfg''$ where $errorU(o, cn)$ holds. If 2(ii) applies (to any successor state), we then reach a state $cfg''$ where $optedIn(l, cn)$ holds. We conclude that from $cfg'$ onwards, $noExecIn(o)$ holds until we reach $cfg''$ when either $errorU(o, cn)$ or $optedIn(l, cn)$ holds. This establishes $\bigcirc(noExecIn(o)\ Until\ (errorU(o, cn) \vee optedIn(l, cn)))$. ■

$$
\begin{aligned}
&\mathscr{C} = \mathscr{C}_1 \wedge \mathscr{C}_2 \\
&\mathscr{C}_1 = \forall o, l, cn. \\
&\quad \Box(collect(o, l, cn) \Rightarrow checked\text{-}scope(o, \langle l, cn\rangle)) \\
&\quad \Rightarrow \Box\neg errorC(o, cn) \\
&\mathscr{C}_2 = \forall o, l, cn. \\
&\quad \Box((collect(o, l, cn) \wedge \\
&\quad \neg complyTo(l, cn) \wedge \neg checked\text{-}scope(o, \langle l, cn\rangle)) \\
&\quad \Rightarrow \bigcirc(noExecIn(o)\ Until\ (errorC(o, cn) \vee optedIn(l, cn))))
\end{aligned}
$$

**Theorem I.A.2** (Consent)**.** *The property $\mathscr{C}$ holds for all traces of our transition system.*

*Proof.* The proof of $\mathscr{C}_1$ is analogous to our previous proof of $\mathscr{P}_1$. Namely, we prove $\mathscr{C}_1$ using a proof by contradiction. Let $\tau$ be a fair trace. The premise of $\mathscr{C}_1$ says that in every state in $\tau$, for all $o$, $l$, and $cn$, whenever *collect(o,l,cn)* holds, then *checked-scope* holds. To achieve a contradiction, assume that $errorC(o, cn)$ holds in a reachable state in $\tau$ and consider the first such state $cfg'$ that it holds. This cannot be the initial state in $\tau$ (per definition) so there must be a transition from the predecessor state $cfg$ to $cfg'$ adding $errorC(o, cn)$ to the configuration. The only rule that could have added $errorC(o, cn)$ is ERROR-COLLECT. In $o(a, (\sigma, (\textbf{collect}(e_1, e_2, e_3); s)@V), db)$, let $cn = [\![e_1]\!]_{a \circ \sigma}$ and $l = [\![e_2]\!]_{a \circ \sigma}$. Since the current program statement in $cfg$ is $\textbf{collect}(e_1, e_2, e_3)$, then $collect(o, l, cn)$ holds. Therefore, $checked\text{-}scope(o, \langle l, cn\rangle)$ holds in $cfg$, and hence also $\langle l, cn\rangle \in V$. Thus, in the rule ERROR-COLLECT, the premise *comply* holds and this rule cannot fire in $cfg$. Thus, $errorC(o, cn)$ does not hold in $cfg'$, so $\mathscr{C}_1$ holds.

The proof of $\mathscr{C}_2$ is analogous to our proof of $\mathscr{P}_2$. In particular, given a fair trace $\tau$, consider any reachable state $cfg$, where the premise of $\mathscr{C}_2$ holds, i.e., for some $o$, $l$, and $cn$, the formulas *collect(o,l,cn)*, $\neg complyTo(l, cn)$, and $\neg$*checked-scope* hold. Let $cfg'$ be the next state in $\tau$. We show that there exists a state $cfg''$ after $cfg'$ where $errorC(o, cn) \vee optedIn(l, cn)$ holds, and $noExecIn(o)$ holds for all states from $cfg'$ up to (but not necessarily including) $cfg''$.

As with the previous proof of $\mathscr{P}_2$, we will use the fact that $noExecIn(o)$ holds when enabled rules not involving the object $o$ fire. There are four such kinds of rules: 1) TICK, 2) DELETE, 3) OPT-OUT, and 4) all enabled rules that apply to other objects than $o$.

Since *collect(o,l,cn)* holds in $cfg$, there is only one possible program statement that can execute according to the definition of *collect* in Equation I.5, which is $\textbf{collect}(cn, l, x)$. Now consider how execution can progress from the next state $cfg'$. Either a program statement in the object $o$ fires, or one of the four kinds of rules fire not involving $o$. This yields the following two cases: 1) The only enabled rule involving $o$ is ERROR-COLLECT, which produces a state $cfg''$ where $errorC(o, cn)$ holds. 2) For the other four kinds of rules that can be enabled, as discussed previously, (i) for all of these rules, ERROR-COLLECT remains enabled except (ii) when the rule OPT-IN, enabled in another object $o'$, fires and thereby adds the contract $cn$ to the policy $l$. If 2(i) holds, then $noExecIn(o)$ holds in the successor state. We can only repeat this case finitely often since ERROR-COLLECT remains enabled, and by fairness, it must eventually fire, leading to state $cfg''$ where $errorC(o, cn)$ holds. If 2(ii) applies (to any successor state), we then reach a state $cfg''$ where $optedIn(l, cn)$ holds. We conclude that from $cfg'$ onwards, $noExecIn(o)$ holds until we reach $cfg''$ when either $errorC(o, cn)$ or $optedIn(l, cn)$ holds. This establishes $\bigcirc(noExecIn(o)\ \mathit{Until}\ (errorC(o, cn) \vee optedIn(l, cn)))$. ■

$$\mathscr{W} = \forall l, cn.\ \Box(optedOut(l, cn) \Rightarrow (\neg complyTo(l, cn)\ W\ optedIn(l, cn)))$$

**Theorem I.A.3** (Right to withdraw consent)**.** *The property $\mathscr{W}$ holds for all traces of our transition system.*

*Proof.* Given a fair trace $\tau$, consider any reachable state *cfg* where $optedOut(l, cn)$ holds, i.e., according to the definition *optedOut* (in Equation I.6), the rule OPT-OUT has been fired and the contract *cn* does not belong to the policy $l$. We show that if there exists a state $cfg'$ in $\tau$, where $optedIn(l, cn)$ holds, then $\neg complyTo(l, cn)$ holds from *cfg* up to $cfg'$. Moreover, if $optedIn(l, cn)$ never holds in any state in $\tau$, then $\neg complyTo(l, cn)$ holds forever from *cfg* onwards.

Since $optedOut(l, cn)$ holds in *cfg*, the contract *cn* does not belong to the policy $l$, thus the formula $\neg complyTo(l,cn)$ also holds in *cfg*. Note that $\neg complyTo(l, cn)$ is invariant over all the rules except OPT-IN. So either 1) the rule OPT-IN eventually fires yielding a state $cfg'$ where $optedIn(l, cn)$ holds, and $\neg complyTo(l, cn)$ holds from *cfg* up to this point, or 2) OPT-IN never fires and then $\neg complyTo(l, cn)$ continuously holds from *cfg*. ■

For deletion, we formalize the auxiliary functions *dec* and *del* (in Section III.4.2) in the following. The function $delData(db, l)$ deletes sensitive data associated with the policy $l$ from the database *db*, which is a substitution. Equations are applied in order, top-down.

$$\begin{aligned}
&dec(l(u,cp,cm,b,t)\ cfg) = l(u, cp, cm, b, t-1)\ dec(cfg) \quad \text{if } t > 1\\
&dec(l(u,cp,cm,b,1)\ cfg) = dec(del(l,\ cfg))\\
&dec(cfg) = cfg
\end{aligned} \tag{I.12}$$

$$\begin{aligned}
&del(l,o(a,p,db)\ cfg) = o(a, p, delData(db, l))\ del(l,cfg)\\
&del(l,\ cfg) = cfg\\
&delData([sensitive(d, l'), \overline{v}], l) = delData([\overline{v}], l) \quad \text{if } l = l'\\
&delData([sensitive(d, l'), \overline{v}], l) =\\
&\qquad\qquad [sensitive(d, l'), delData([\overline{v}], l)] \quad \text{if } l \neq l'\\
&delData([], l) = []
\end{aligned} \tag{I.13}$$

$$\mathscr{S} = \forall l.\ \square(expired(l) \Rightarrow dbDel(l))$$

**Theorem I.A.4** (Storage limitation)**.** *The property $\mathscr{S}$ holds for all traces of our transition system.*

*Proof.* Given a fair trace $\tau$, consider any reachable state *cfg* where $expired(l)$ holds by Equation I.8, a policy $l$ does not exist in the state *cfg*. We show that $dbDel(l)$ also holds in *cfg*, where data with the policy $l$ is deleted from all objects' databases. Note that the state *cfg* cannot be the initial state because in the initial state there is no policy. In the predecessor state of *cfg*, the only rules that could have deleted a policy $l$, yielding $expired(l)$ in *cfg*, are the following:

$$\frac{d = [\![e]\!]_{a \circ \sigma}}{o(a, (\sigma, (\textbf{store}(k, e)\ \textbf{else}\{s\}; s')@V), db) \to o(a, (\sigma, s'@V), db[k \mapsto d])}\ \textsc{store}^*$$

$$\frac{x \in dom(a) \quad d = [\![e]\!]_{a \circ \sigma}}{o(a, (\sigma, (x := e; s)@V), db) \to o(a, (\sigma[x \mapsto d], s@V), db)}\ \textsc{assign-local}^*$$

$$\frac{x \in dom(a) \quad d = [\![e]\!]_{a \circ \sigma}}{o(a, (\sigma, (x := e; s)@V), db) \to o(a[x \mapsto d], (\sigma, s@V), db)}\ \textsc{assign-field}^*$$

$$\frac{d = [\![e]\!]_{a \circ \sigma} \quad o' = \sigma(caller)}{o(a, (\sigma, \textbf{return}\ e\ @V), db) \to o(a, idle, db)\ com(d, o')}\ \textsc{return}^*$$

$$\frac{o = \sigma(caller) \quad d = [\![e]\!]_{a \circ \sigma}}{o(a, (\sigma, (\textbf{return}\ e; cont(n))@V), db)\ n(\sigma', (x := m?; s)@V') \to o(a, (\sigma'[x \mapsto d], (s)@V'), db)}\ \textsc{self-return}^*$$

Figure I.9: Rewrite rules for operations on non-sensitive data.

- Tick: The TICK rule was applied and the timestamp of the policy was one. In this case, the *dec* function (in Equation I.12), deletes the policy, where *expired*($l$) holds in *cfg*, and the function *del* (in Equation I.13) deletes sensitive data associated with the policy $l$ from databases, so that *dbDel*($l$) holds in *cfg*. Hence, $\mathscr{S}$ holds.

- Delete: The rule DELETE was applied. In this case, the policy is deleted, where *expired*($l$) holds in *cfg*, and the function *del* deletes sensitive data associated with the policy $l$ in databases, so that *dbDel*($l$) holds in *cfg*. Hence, $\mathscr{S}$ holds.

■

$$\mathscr{F} = \forall l.\ \Box(deleted(l) \Rightarrow expired(l))$$

**Theorem I.A.5** (Right to be forgotten)**.** *The property $\mathscr{F}$ holds for all traces of our transition system.*

*Proof.* In a fair trace $\tau$, consider any reachable state *cfg* where *deleted*($l$) holds by Equation I.10, the rule DELETE was applied to a policy $l$. We show that *expired*($l$) also holds in *cfg*, and by Theorem I.A.4 data with that policy is deleted from databases. Note that the state *cfg* cannot be the initial state because in the initial state there is no policy. Since *deleted*($l$) holds in *cfg*, then in the predecessor state of *cfg*, the rule DELETE was applied, which deletes the policy, yielding *expired*($l$) in *cfg*, so $\mathscr{F}$ holds. ■

### I.A.2 Rewrite rules for non-sensitive data

Figure I.9 shows the rewrite rules for non-sensitive data, omitted from Sect. III.4.2. These rules mirror their non-starred counterparts for sensitive data, but without compliance checks. Rule STORE* stores non-sensitive data $d$ in the database, ASSIGN-LOCAL* assigns non-sensitive data $d$ to local variables, ASSIGN-FIELD* assigns non-sensitive data $d$ to fields, and RETURN* returns non-sensitive data $d$ to the caller.

### I.A.3 Maude formalization

**Maude model** We specify DPL's operational semantics in Maude, which gives us a prototype environment for program simulation and verification.

In our Maude model, a program is written in a main block with a multiset of classes: *main{L, SL} cfg*, where $L$ is an initial state (substitution), $SL$ is a list of statements, and *cfg* specifies the classes. An object is represented as

$$\langle\ O : C \mid Att: S,\ Pr: (L,\ SL),\ Lcnt: N,\ Db: DB\ \rangle$$

consisting of the object name $O$, the class name $C$, attributes $S$, the active process *(L, SL)* with local variables $L$ and statements $SL$, a counter for creating unique identities $N$, and the database $DB$. The counter is a technical device, omitted from the previous sections, used to create fresh identifiers, corresponding to the use of the *fresh* predicate in the operational semantics. For simplicity, class names represent purposes, and an object created from a class $C$ gets the purpose $C$.

A policy is represented as

$$PL\ \langle\ U,\ Cp,\ Cm,\ B,\ T\ \rangle$$

where $PL$ is the policy identity, $U$ is a user identity, $Cp$ is a set of persistent contracts, $Cm$ is a set of mutable contracts, $B$ is true if data is allowed to be stored persistently, and $T$ is the timestamp.

For **`opt-in`** options, we assign a user object to consent to or deny an **`opt-in`** option. The user object sends the message *optInMsg(true, 'contract, 'policy)* to consent and *optInMsg(false, 'contract, 'policy)* to deny consent. In Maude's syntax, a name is represented by *'name*. Input data for data collection is given in the *collect* command; i.e., *collect(int(1), 'contract, 'policy, 'x )*, where *int(1)* is the input data, and the function *int(1)* creates data of type Integer. Moreover, a userId is given in the command *logIn(str('u))*, where the function *str()* creates data of type String.

The rules TICK, DELETE, and OPT-OUT can be interleaved at any point in the program reflecting clock ticks and user actions. To specify fairness and some of the formulas in Table I.2, we add the element $ruleLabel(Id, R)$ to the configuration, where $Id$ is the identity of the current object that is executing, and $R$ is the current rule's label that is fired. When a rule fires, the parameters of *ruleLabel* change accordingly.

**A rewrite rule example** Here, we present the formalization of the DELETE rule in Maude. Note that in this paper, we omit the *ruleLabel(Id, R)* from the rules, where *Id* is the identity of the object that is executing, and *R* is an event constructed from a rule label and possibly parameters. In our Maude model, *ruleLabel(Id, R)* is added to all the rules. In the following, when the rule *delete* fires, *Id* changes to *Other*, which is a constant of type *Identity*, and *R* to *delete(PL)* of type *Event*. Note that if an object *O* is executing, then *Id* changes to *O*, which is also of type *Identity*.

```
rl [delete] :
{ PL ⟨ U, Cp, Cm, B, T ⟩ ruleLabel(Id, R) Cfg }
 =>
{ del(PL, ruleLabel(Other, delete(PL)) Cfg) } .
```

**Model checking results for programs** We define the LTL formulas in Section I.4 in Maude and use Maude's model checker to verify the GDPR properties that we previously formalized on the online retailer example in Fig. I.3.

**Hygienic programs**: We verify the properties $\mathscr{P}_1$, $\mathscr{C}_1$, $\mathscr{W}$, $\mathscr{S}$, and $\mathscr{F}$ on the (hygienic) online retailer program. Maude verifies that errors cannot occur and returns true for each of these properties.

**Non-hygienic programs**: To check the properties concerning programs giving rise to errors, we model check different scenarios for non-hygienic programs for Fig.I.3. Namely, we remove different combinations of the statements to systematically check the necessity of the all conditions for hygienic programs given in Section I.4.

The scenarios are as follows: i) For the Purchase purpose, we do not define the consent statements cs1 and cs2 and the contract cn1. Therefore, in the register method, credit card data and customer data are collected with the MassMarketing's consent statement and contract (cs3, cn2). Maude throws a data usage error when the method p. purchase(credit, customer) is called. To avoid this error, the **`if-comply`**(cn1, (credit, customer)) is required (line 45). ii) The pattern for data collection, in Section I.3.2, is not followed. In this case, we check two scenarios: 1) We skip the log-in() (line 11). Maude throws an error when creating a policy (line 13). 2) We remove the **`if-consent`** in line 16, to check that the appropriate error arises when collecting data for the Purchase purpose. Maude verifies that a data collection error arises. We check the consent property $\mathscr{C}_2$ for the credit data with the policy l1 with respect to the Purchase contract. Maude verifies the property $\mathscr{C}_2$. iii) We remove the **`if-comply`** for the call mm.m-marketing (line 47), and check $\mathscr{P}_2$. Maude verifies that the appropriate error arises when making the call m-marketing. Note that we check the purpose limitation property $\mathscr{P}_2$ on the customer data with respect to the 'MassMarketing contract.

Here, we present Maude's model checker results for a non-hygienic variant of the program in Fig.I.3, where the **`if-consent`** in line 16 and the **`if-comply`** for the call mm.m-marketing (line 47) are removed. In the following commands, *init* is the initial configuration built from the program's main block. The

term *sensitive(str('mail), policy(2))* is sensitive data associated with the customer data and the policy *policy(2)*. The term *contract(str('MassMarketing), ob('MassMarketing0))* is the contract for the object *MassMarketing0*. The term *contract(str('Purchase), ob('Purchase0))* is the contract for the object *Purchase0*. The term *policy(1)* is the policy for the credit data. In the following, Maude verifies the properties $\mathscr{P}$, $\mathscr{C}$, $\mathscr{W}$, $\mathscr{S}$, and $\mathscr{F}$, respectively:

```
red modelCheck(init, purposeLimitationconf
     (init, sensitive(str('mail), policy(2)),
       contract(str('MassMarketing), ob('MassMarketing0)))) .
result Bool: true

red modelCheck(init, consentconf(init,
     policy(1), contract(str('Purchase), ob('Purchase0)))) .
result Bool: true

red modelCheck(init, withdraw(init,
     contract(str('Purchase), ob('Purchase0)), policy(1))) .
result Bool: true
red modelCheck(init, storageLimitationconf(init, policy(2))) .
result Bool: true

red modelCheck(init, forget(init, policy(2))) .
result Bool: true
```

### I.A.4 Case study extension

We expand on the example from Section I.3.3 and show that DPL prevents illegal data usage and storage in objects receiving sensitive data. These objects can only store or process sensitive data if their contracts comply with the policy, i.e., if consent for the object's purpose is given. When a user withdraws consent or requests data deletion, the corresponding policy changes. Thus, prior to any data usage, conditional constructs are needed to avoid errors. Moreover, objects cannot illegally send sensitive data to other objects since errors occur.

We present the classes Purchase-c and MMarketing-c. The corresponding objects p and mm receive the credit and customer data. In class Purchase-c, the method purchase stores the customer data if storage is allowed and the user's consent for the Purchase purpose is given. In line 7, the object's contract is checked for compliance, then data processing continues; otherwise, the default value "0" is returned. In line 11, the caller's contract is checked for compliance before returning the method result. In the class MMarketing-c, we create a new object tm for targeted marketing and try to illegally send customer data to this object via a method call (line 28). The call triggers an error since tm's contract is not defined and does not comply with the customer's policy. Moreover, if we define the corresponding contract/consent statement, we still need a session and the user's consent to add the contract to the policy. In line 34, the m-marketing method checks if the object's contract complies with the policy, then continues processing the data. A message with the given text is sent to the customer and a copy of the message is returned to the caller, which is our main object.

```
1  class Purchase-c implements Purchase() {
2      Order purchase(String credit, String customer){
3          user = ... // Make user accounts for data storage
4          // Store data if storage is allowed, otherwise skip:
5          store(key(user, "customer"), customer) else { skip; }
6          // Check if data usage is allowed:
7          if-comply(cnThis, credit, customer){
8              ...
9              Order order=...
10             // Check if the caller is allowed to receive the result
11             if-comply(cnCaller, order) { return(order); }
12             else{ return 0; }
13         } // End of if-comply(cnThis, credit, customer)
14         else {return 0; }
15     }// End of the purchase method
16 } // End of class
17
18
19 purpose TargetedMarketing {
20 String targetedMarketing(String customer){...}}
21
22 class TargetedMarketing-c implements TargetedMarketing() {...}
23
24 class MMarketing-c implements MassMarketing() {
25     TargetedMarketing-c tm = new TargetedMarketing-c();
26     String m-marketing(String customer){
27     // Let us send data illegally to tm
28     tm.targetedMarketing(customer); // This results in an error
29
30     user = ... // Create user accounts for data storage
31     // Store data if storage is allowed, otherwise skip
32     store(key(user, "customer"), customer) else { skip; }
33
34     if-comply(cnThis, customer){ // Self-sanity check
35         String text=...
36         ... // Send MassMarketing text to the customer
37         return text;
38     } // End of if-comply
39     else {return 0; }
40     } // End of m-marketing method
41 } // End of class
```

Figure I.10: Extension of the online-retailing example in DPL from Fig. I.3.

Similarly, we run Maude's model checker for the program in Fig. I.10, and Maude verifies all the properties in Section I.4.

### Authors' addresses

**First Author** University of Oslo, Oslo, Norway, farzanka@ifi.uio.no

**Second Author** ETH Zurich University, Zurich, Switzerland, basin@inf.ethz.ch

**Third Author** University of Oslo, Oslo, Norway, einarj@ifi.uio.no

Paper II

# An Evaluation of Interaction Paradigms for Active Objects

**Farzane Karami, Olaf Owe, Toktam Ramezanifarkhani**

Published in Journal of Logical and Algebraic Methods in Programming, 2019, Volume 103, pp. 154–183. DOI: https://doi.org/10.1016/j.jlamp.2018.11.008

**Abstract**

Distributed systems are challenging to design properly and prove correctly due to their heterogeneous and distributed nature. These challenges depend on the programming paradigms used and their semantics. The *actor paradigm* has the advantage of offering a modular semantics, which is useful for compositional design and analysis. Shared variable concurrency and race conditions are avoided by means of asynchronous message passing. The object-oriented paradigm is popular due to its facilities for program structuring and reuse of code. These paradigms have been combined by means of concurrent objects where remote method calls are transmitted by message passing and where low-level synchronization primitives are avoided. Such kinds of objects may exhibit active behavior and are often called *active objects*. In this setting the concept of *futures* is central and is used by a number of languages. Futures offer a flexible way of communicating and sharing computation results. However, futures come with a cost, for instance with respect to the underlying implementation support, including garbage collection. In particular this raises a problem for IoT systems.

The purpose of this paper is to reconsider and discuss the future mechanism and compare this mechanism to other alternatives, evaluating factors such as expressiveness, efficiency, as well as syntactic and semantic complexity including ease of reasoning. We limit the discussion to the setting of imperative, active objects and explore the various mechanisms and their weaknesses and advantages. A surprising result (at least to the authors) is that the need of futures in this setting seems to be overrated.



## II.1 Introduction

Programming paradigms are essential in software development, especially for distributed systems since these affect large programming communities and a large number of applications users. The *actor model* [36] has been adopted by a number of languages as a natural way of describing distributed systems. The advantages are that it offers high-level and yet efficient system designs,

and that the operational semantics may be defined in a modular manner, something which is useful with respect to scalability. The actor model is based on concurrent autonomous units (actors) communicating by means of asynchronous message passing, and with a "sharing nothing" philosophy, meaning that no data structure is shared between actors. The actor model offers high level, yet efficient, constructs for synchronization and communication.

A criticism of the interaction mechanism of the actor model has been that its one-way communication paradigm may lead to complex programming when there are dependencies among the incoming messages. It is easy to make programming errors such that certain messages are never handled. And it is not straightforward to augment an actor model with support of additional messages and functionality. In the actor model one may not classify and organize the communication messages in request messages and reply messages, and it does not support object-oriented (OO) principles such as inheritance, late binding, and reuse (even though the original actor concept was inspired by the ideas behind object-orientation).

To overcome these limitations, one may combine the actor model and object-orientation, using the paradigm of concurrent, *active objects* and using methods rather than messages as the basic communication mechanism, thereby supporting imperative programming in a natural manner. The active object model has gained popularity and is an active research area [8]. A call of method $m$ on a remote object $o$ could have the form $x := o.m(\overline{e})$ where $\overline{e}$ is the list of actual parameters. This opens up for two-way communication where both the method call and the corresponding return value are transmitted by message passing between then caller and callee objects. The naive execution model is that the caller waits while the callee performs the call, and then stores the result in the program variable $x$. However, this can result in undesired blocking and possibly deadlock. Therefore non-blocking call mechanisms are needed.

One way of avoiding unnecessary waiting is provided by the *future mechanism*, originally proposed in [5] and exploited in *MultiLisp* [31], *ABCL* [85], and several other languages. A future is a read-only placeholder for a result that is desirable to share by several actors, where the placeholder may be referred to using the *identity* of the future. In particular, one may refer to a result even before it is produced, and a future identity may refer to a *future* method result. In languages with first-class futures, future identities can be passed around as first-class objects like references. Futures can give rise to efficient interaction, avoiding active waiting and low-level synchronization primitives such as explicit signaling and lock operations. The notion of *promises* gives even more flexibility than futures by allowing the programmer to refer to a computation result before it is known how to generate it, and by which process. For instance, a promise may be used to refer to the result of one of several futures.

A future object with its own identity can be generated when a remote method call is made. Then the caller may go on with other computations until it needs the return value, while the callee executes. The callee executes the called method and sends the return value back to the future object upon termination of the method invocation, at which time the future is said to be *resolved* (i.e., the future value is available). When the caller needs the future value it may request

the future value, and is blocked until the future is resolved. A programming language may have implicit or explicit support of futures. Consider first explicit futures: Typically a call statement defines the future identity, say $f := o!m(\overline{e})$, where $f$ is a *future variable* used to hold the future identity of the call (with $m$, $o$, and $\overline{e}$ as above). Here the symbol "!" indicates the difference from a blocking call (assuming both are allowed). When the result of the call is needed, the caller uses a construct like **get** $f$ where $f$ is an expression giving the future identity, for instance in an assignment $x :=$ **get** $f$. By letting futures be first-class entities, the objects may communicate future identities and thereby allow several objects to share the same method result, given as a future. Any object that has a reference $f$ to a future may perform **get** $f$. Implementation of call requests and future operations can be done by means of message passing.

Implicit futures are similar, except that the future variable is not available for the programmer, and the **get** operations are made implicitly as defined by the semantics. The call may now look like $x := o.m(\overline{e})$ (or say $x := o!m(\overline{e})$ to distinguish it from that of a synchronous call, if both are desired) where $x$ is of the return value type, and the implicit **get** operations may happen when the value of $x$ is needed (first time after the call). This is attractive in functional languages, avoiding the distinction between a function returning a future and one returning the future value, or receiving a future input versus a future value. However, implicit futures make static program analysis difficult since the waiting points are implicit, possibly depending on dynamic factors. In particular, certain kinds of textual analysis become infeasible.

In languages with futures, the two-way communication mechanism is replaced by a more complex pattern, namely that a method call generates a future object where the result value can be read by a number of objects, as long as they know the future identifier. A normal two-way call can be done by letting the caller ask and wait for the future. This means that each call has a future identity, and that the programmer needs to keep track of which future corresponds to which call. This gives an additional layer of indirectness in programming. Our experience is that the full functionality of futures is only needed once in a while, and that basic two-way communication suffices in most cases. Thus the flexibility of futures (and promises) comes at a cost. Implementation-wise, garbage collection of futures is non-trivial, and static analysis of various aspects including deadlock detection in presence of futures is more difficult. Even if one introduces a short-hand notation for the simple two-way call interaction, there is still a future behind the scene, and thus all calls are typically handled uniformly by this more expensive implementation mechanism.

Another drawback of the basic future mechanism is that once a **get** operation is done, the current object is blocked as long as the future is not yet resolved. To overcome this, one may allow *polling*, i.e., testing if a future is resolved or not, without blocking, for instance used in an if-test where the branches deal with the two cases. But polling may result in complex program structures since it opens up for explicit program control of the possible message ordering.

Another way of avoiding blocking is the notion of *cooperative scheduling* suggested in the *Creol* language [39], and the *OUN* language [19, 62], generalizing

the concept of guards from the guarded command language of Dijkstra [23] by adding a notion of process suspension. Cooperative scheduling can be achieved by a language construct, **`await`** $c$, where $c$ is a condition, either a boolean condition or a waiting condition, such as the presence of the result of a remote method call. If $c$ is not satisfied, the current executing method invocation ("process") is placed on the process queue of the object, which allows another enabled process on the queue, or an incoming request, to continue. A process on the queue is not enabled if it starts with an await with a condition that is not satisfied. Thus a method invocation may passively wait in the queue while the object is active and able to take care of other (enabled) processes. Thus cooperative scheduling provides local synchronization control and provides a constructive approach to the scheduling of processes internally in an object. Cooperative scheduling may be combined with the future mechanism, for instance with first-class futures as in the ABS language, or with non-first-class futures as in the Creol language, where the futures are local to a process. We refer to such non-first-class futures as *local futures*; and in general, we may talk about object-local and method-local futures. Object-local futures may not be communicated to other objects, but assignment of futures to fields and local variables is acceptable as well as passing of futures through parameters or return values of local methods. Method-local futures are local to a method instance (process) and may not be assigned to fields and may not be passed as parameters or return values of method calls.

In this paper, we will focus on the interaction mechanisms in imperative, active object languages, especially the paradigm of asynchronous call/return without use of futures, versus the different versions of the future mechanism, as well as cooperative scheduling and polling. The contribution of the paper is a comparison on the different interaction mechanisms, based on a survey of representative languages, and a unified syntactic and semantic formalization of the various language combinations. We give a critical discussion on the pros and cons of the various combinations of these mechanisms. As most recent imperative languages for active objects support explicit, first-class futures, we leave out implicit futures from our discussion. To complement the discussion, we suggest some language improvements in the setting of asynchronous call/return without use of futures. We compare the various interaction paradigms wrt. the following criteria:

- *expressiveness*
- *efficiency*
- *syntactic and semantic complexity*
- simplicity of program reasoning and static analysis
- information *security* aspects.

The paper is organized as follows: Section II.2 provides the context of the work, giving an overview of the interaction mechanisms of a number of active object languages, including ABCL, Rebeca, Creol, ABS, Encore, and

ASP/ProActive. A complementary communication model and language is proposed in Subsection II.2.7. In order to make a comparison easier, Section II.3 defines a unified syntax and semantics for the different interaction paradigms. Then Section II.4 evaluates the different interaction mechanisms along the comparison dimensions. Finally, conclusions are given in Section II.5.

## II.2 Background

In this section we review some representative languages based on *active objects*, and give a summary of their interaction models. We limit the discussion to imperative languages, since a majority of modern active object languages (with some exceptions like Scala) are imperative. For each language we identify support of active and/or passive behavior and interaction mechanisms, including synchronization mechanisms involving waiting and blocking, as well as cooperative scheduling. In particular, we explain support of explicit or implicit futures and polling mechanisms. We focus on explicit futures since the semantical issues of these are more clearly connected to syntactic constructs. This allows us to make a syntax-oriented, language-based comparison.

Furthermore, we look at shared futures as well as local futures. Local futures include *object-local futures*, not permitted to be communicated and shared with other objects, and *method-local futures*, not permitted to be stored in fields or passed to other method invocations than the one creating the future. Local as well as shared futures may in principle be read multiple times, but for method-local futures the value of multiple reads is questionable. It can be statically checked that a method-local future is read at most once, leading to a notion of single-use, method-local futures, which gives the simplest form of futures.

To illustrate and compare the interaction mechanisms in the different languages, we use a running example. It is part of a subscriber service that was originally made to take advantage of the benefits of first-class futures. The ABS solution in Figure III.2 is most close to the original version, and should be read first. In this example, the server, defined by class *Service*, searches for news and publishes them to subscribing clients, using proxies. The server communicates news to the proxies by means of first-class futures, so that the server itself does not wait for incoming news and is free to respond to any client request (apart from doing synchronized database operations). Instead the proxies wait for the incoming news. The proxies are organized in a list (growing upon need), letting each proxy handle a limited number of clients.

### II.2.1 ABCL

The integration of the actor model with object-oriented concepts was first introduced in ABCL [85]. In this language, concurrent objects interact via asynchronous message passing and futures. An object definition as depicted in Figure II.1 includes: the object's name, its state declaring the local object variables (fields) and initialization, and its script including patterns of messages

```
[object object-name
    (state representation of local memory ...)
        (script
           (=> message pattern1)
           (=> message pattern2))]
```

Figure II.1: Object definition in ABCL.

received by the object, and a set of corresponding actions. Each object has its own queue for storing the messages according to their arrival time. When an object receives a message matching one of the declared patterns, it performs the corresponding actions. ABCL uses first-class futures, which are explicitly created by the syntax *make-future*. Moreover, a future is a queue, and all receiving objects can write to it, but only the object which creates it, can access and check the future values, in contrast to other languages supporting first-class futures. In fact, most other languages implement a future object as once-writable and multiple-readable (by many objects).

Assuming an object $o$ sends a message $m$ to an object $o'$, ABCL supports three types of message passing [85]:

1. *Past-time* message passing (send and no wait):
   After sending the message, the sender $o$ immediately continues its process without waiting for a reply or delivery. If the reply should be sent to other objects, the (optional) *reply-destination* is the destination of those objects. The syntax for this kind of message passing is:

   $$o' <= m \ @\textit{reply-destination}$$

2. *Now-time* message passing (send and wait):
   Object $o$ blocks while waiting for the result from $o'$, then assigns the result to a program variable $x$. The notation for this type of message passing is:

   $$x := \ o' <== m$$

3. *Future-type* message passing (reply to me later):
   In this case, $o$ does not need the result immediately, and instead of blocking it can continue and later on check whether the future object contains the result or not. In this case of message passing, the reply destination is the specified future object. The notation for this kind of message passing is:

   $$o' <= m \ \$f$$

   where the future variable $f$ is bound to the future object.

In ABCL, an object that creates a future can check its values by the operation:

**ready?** $f$

If at least one reply is stored in the future $f$, the value of this form is $t$ (i.e., true); otherwise, *nil*. Thus polling is supported. Moreover, the operation

**next-value** $f$ *options*

returns the first element stored in the future $f$, and if the future is empty the owner object waits until a reply arrives. And with a **:remove t** option the future value is removed from the *future-object*. Whereas, with the **:remove nil** option, it still remains in the future queue even after evaluation of this form. The default option is **:remove t**.

**Example** Figure II.2 shows a subscriber example in the ABCL language. In ABCL, bracket forms are often used to build message patterns; and in a message pattern, a symbol starting with a colon (:) represents a tag, and other symbols are pattern variables. Executing an expression [*object*...] creates an object with a specified behavior defined in the expression. In this language, the symbol **Me** stands for the object which executes the operation in which the "**Me**" exists. Moreover, by using a reply form !$form$, the evaluation result of the form is sent back as a reply to the currently processed message.

In the subscriber example, the *publish* call in the state definition of *service* creates a cycle between service and proxy objects, since each such call leads to a *produce* and indeed another *publish* call in the object proxy. In the object service, since *publish* and *detectNews* calls are past-time, interleaving of other calls (such as *subscribe* and *unsubscribe* calls) is possible between each execution of *publish* or *produce*. In line 2, object service sends a [:publish] message to proxy. As a response, in lines 15 and 19, object proxy creates a future and appends it to a produce message toward object service, respectively. Therefore, waiting points for detecting news are delegated to the proxy by using futures. Object proxy owns the *future*, and only this object has the access to values, while other objects can only write to the *future*. In line 6, when object service receives a *produce* message, it sends a past-time *detectNews* message to the object producer, searching for news, with a reply destination *Me*. And according to an exclamation mark ! in line 35, the reply from evaluation of *detectNews* is sent back to *Me*. In line 6, according to the exclamation mark ! the result from the evaluation is replied to the *future* variable. In line 21, object proxy first checks if the future is available, then by the command *next-value* retrieves the value and multi-casts it to clients. In line 24, if nextProxy is empty, the object proxy continues to search for new news; otherwise, it publishes current news to clients subscribed to the nextProxy.

### II.2.2 Rebeca

Rebeca [76, 78] is an active object language that is more close to the actor-based model than the other languages considered here. In this language, active objects are called *rebecs* (reactive object). Each rebec is instantiated from a reactive class and has its own thread of control. A reactive class consists of an interface,

```
1  [object service
2      state dataBase db; int limit; [proxy <= [:publish]]; )
3          // proxy does the main job and initiates a produce call
4      (script
5      (=> [:produce proxy]
6          ![producer <= [:detectNews] @ Me]; //reply destination is Me
7          db <= [:logging];)
8          ...
9      (=> [:subscribe]...)
10     (=> [:unsubscribe]...))
11 ]
12
13 [object proxy
14     (state list myClients:=nil; News ns; proxy nextProxy:=nil;
15     future := (make-future);)
16     (script
17     ...
18     (=> [:publish]
19     [service <= [:produce Me] $ future];
20     // replies from object service saved in the future
21     if (ready? future){ // polling on the future
22         [ns := (next-value future)];
23         [myClient <= [:signal ns]];} // multi-cast the result
24     if (= nextProxy nil)
25         [Me <= [:publish]]
26     else
27         [nextProxy <= [:publish]];))
28 ]
29
30 [object producer
31     (state News ns;)
32     (script
33     (=> [:detectNews]
34         ...
35         ! ns:=...))
36 ]
37
38 [object myClient
39     (script
40     (=> [:signal ns] ....))
41 ]
```

Figure II.2: A version of the subscriber example in the ABCL language.

variables, method definitions (*message server*) for dealing with messages and initial methods. An initial method of a rebec triggers declared messages toward other rebecs. The receiving objects react to these messages according to their method definitions. Communication in this model is one-way asynchronous message passing, without shared variables, blocking receive, nor futures. Since the communication is by asynchronous message passing, each rebec has its own message queue, with FIFO order. Rebeca actors are isolated, therefore their analysis and verification become feasible.

Sometimes it is necessary to have synchronous communication, thus in extended versions of Rebeca the *component* concept is defined. A component encapsulates rebecs that may have internal synchronous communications [77]. External communication beyond a component is either an asynchronous broadcast or an asynchronous message toward another rebec. RebecaSys [67] is another extended model of Rebeca supporting global variables and the *wait*($e$) statement. This statement temporarily stops the execution of the process. The Boolean expression $e$ may only contain global variables that all rebecs have access to. Hence, the *wait* statement depends on the rebecs that update these variables.

**Example** Figure II.3 represents the subscriber example with the extended version of Rebeca, considering futures as global variables. The initial *produce* message in reactiveclass *Service* creates a cycle since each such message leads to a *publish* message, which in turn leads to another *produce* message. The *future* variable is a Boolean global variable that the prod rebec sets to true when it completes a *detectNews* call, in line 19. The service rebec as a server asynchronously broadcasts a *detectNews* message to anonymous receivers, which only one of the rebecs providing these messages reacts to, and also sends an asynchronous *publish* message to object proxy. In line 30, proxy rebec is blocked waiting for the *future* to become true. Then in line 31, it sends a *send* message, provided in prod rebec, to signal news to subscribed clients. It is possible to make a simpler version in Rebeca without global variables, but we here want to illustrate how futures can be simulated.

### II.2.3 Creol

Creol was developed from the OUN language [19] based on the notion of active concurrent objects. Interaction is by means of asynchronous methods, implemented by message passing, and remote field access is not allowed. The synchronization mechanisms include suspension, allowing passive (non-blocking) waiting on a Boolean condition or on the arrival of a return value from another object [39–41, 44]. This allows non-blocking as well as blocking method calls.

The visible behavior of objects is specified through interfaces. Thus methods not exported through an interface may only be used for self calls. The behavior of objects can change dynamically between active and passive (reactive) by means of asynchronous self calls. Multiple inheritance is supported as well as dynamic code modification. Basic Creol supports method-local futures (so-called "call labels"), i.e., futures may neither be passed as parameters nor assigned to

```
globalvariables {boolean future;}
reactiveclass Service() {
    init() {Producer prod; Proxy nextProxy; self.produce(DataBase db); }
    // initial action, starting a produce cycle
    produce(DataBase db){
        detectNews();
        proxy.publish(self, prod, nextProxy); // no waiting
        logging(){...} } // logging in a database for services
    ...
    subscribe(Client me){...}
    unsubscribe(Client me){...}
}

reactiveclass Producer() {
    News ns;
    init() { future= false;} // initialization
    detectNews(){
        ns=...; // wait for more news
        future= true;
    }
    send(Client myClients){
        if (future)
        myClients.signal(ns); }//assuming multi-casting is ok in Rebeca
}

reactiveclass Proxy() {
    List[Client] myClients:=null;
    ...
    publish(Service s, Producer prod, Proxy nextProxy){
        wait(future); // wait for the future
        send(myClients);
        if nextProxy == null
        s.produce();
        else
        nextProxy.publish();}
}

reactiveclass Client(){
    News latestNews;
    signal(News ns){ latestNews= ns;}
}

main {Service s(Database db); Proxy proxy;}
```

Figure II.3: A subscriber example in the Rebeca language.

fields. However, first-class futures are allowed in extensions of Creol. Methods are given with the keyword **op** and Creol methods may have several inputs as well as several outputs (indicated by the keyword **out**). Variables are declared by the syntax **var** $name : T = e$ where $e$ is the initial value of type $T$. Creol has a small-step operational semantics defined by a set of rewrite rules in the Maude format [26], used for proving the soundness of analysis and verification [24, 25, 38], and also providing an executable interpreter.

Communication between Creol objects is two-way, passing actual parameters from the caller to the callee object when a method is called, and passing method return values from the callee to the caller when the method execution terminates.

The asynchronous method call command $t!o.m(\overline{e})$ where $t$ is a call label ("tag"), sends a call request message to the callee $o$ and the caller object proceeds without waiting. This call generates a unique call identity for referencing the call, assigned to $t$. Passive waiting for return values is possible by means of *cooperative scheduling*. Each active object has an *internal process queue* containing the processes that are suspended, either waiting for a return value or a Boolean condition. In addition there is an external queue for receiving method call requests from other objects.

A process is suspended when the suspension statement **await** $c$ is executed in a state where the condition $c$ is false. The executing process is moved to the *process queue* of the object, and the object is then free to do something else, like serving an incoming call request or continuing an enabled process in the process queue of the object. Similarly the statement **await** $t?$ suspends when the return value for the call with the identity of $t$ has not arrived. Otherwise, the await statement is enabled, and execution continues with the next statement. In contrast, the command $t?(\overline{x})$ blocks while waiting for the return values, and assigning these to the variable list $\overline{x}$. A label $t$ is local to the current process and cannot be passed to other processes, nor assigned or read by other kinds of statements. The sequence $t!o.m(\overline{e}); t?(\overline{x})$ (abbreviated $o.m(\overline{e}; \overline{x})$) corresponds to a synchronous method call, blocking the processor of the current object until the return values are available, whereas the sequence $t!o.m(\overline{e})$; **await** $t?(\overline{x})$ (abbreviated **await** $o.m(\overline{e}; \overline{x})$) corresponds to a non-blocking call, where **await** $t?(\overline{x})$ abbreviates **await** $t?$; $t?(\overline{x})$. The label $t$ may be omitted in a ! call statement if that $t$ is not needed in a ? statement. Multi-casting can be allowed by the syntax $!o.m(\overline{e})$ where $o$ is a list of objects, in which case the replies cannot be received (since there is no associated label). Note that labels (of type *Label*) are not typed by the return value. Static type checking is possible by certain language restriction (avoiding that the same label is used for several return value types, at a given program point).

**Example** The subscriber example, which originally makes use of first-class futures, must be redesigned in Creol, for instance as done in Figure II.4. Here the passing of a future is replaced by suspension, which means that the Service object may continue with other tasks as in the version with first-class futures. The suspended process can be compared to an added proxy-like object, while the

```
type News = ...

interface ServiceI{
    with ClientI
        op subscribe(out result:Bool)
        op unsubscribe(out result:Bool)
    with Any
        op produce()
}
interface ProxyI{
    with ServiceI, ProxyI
        op publish(ns:News)
    ...
}
interface ProducerI{
    with ServiceI op detectNews(out result:News)
    ...
}
interface ClientI{with Any op signal(ns:News) ...}
interface DataBase{with Any op logging(...) ...}

class Service(limit:Nat, prod:ProducerI, db:DataBase) implements ServiceI {
    var proxy:ProxyI = new Proxy(limit,this); //proxy does the main job
    {!this.produce() } // initial action, starting a produce cycle

    op produce(){var ns:News;
        var t: Label;
        t!prod.detectNews();
        db.logging(. . .) // logging in a database for services
        await t?(ns)// waiting while suspending
        !proxy.publish(ns) } // sends the value

    with ClientI
        op subscribe(out result:Bool) {...}
        op unsubscribe(out result:Bool) {...}
}

class Proxy(limit:Nat, s:ServiceI) implements ProxyI{
    var myClients:List[ClientI]=Nil; var nextProxy:ProxyI;
    ...
    op publish(ns:News){
    !myClients.signal(ns);  // multi–cast the result
    if nextProxy=null
    then !s.produce() else !nextProxy.publish(ns) fi}
}
```

Figure II.4: A version of the subscriber example in the Creol language.

*Proxy* objects in the Creol solution are not blocked in contrast to the ABS version with first-class futures. Note that Creol insists on typing of object variables by interfaces, and we therefore sketch all interfaces. An interface consists of a number of operations (and semantic specifications, ignored here), and each operation has a *co-interface*, restricting what kind of objects may appear as callers. For instance, *subscribe* has *ClientI* as co-interface, meaning that method *subscribe* may only be called by objects supporting *ClientI*. This implies that the implicit *caller* parameter is of interface *ClientI*, which allows us to ensure statically that *myClients* is a list of *ClientI*, say by passing *caller* to a method

```
1 op add(c:ClientI) {
2     if length(myClients) < limit
3     then myClients:=append(myClients,c )
4     else if nextProxy=null then nextProxy:=new Proxy (limit,s) if;
5     !nextProxy.add(c) fi }
```

of class *Proxy*, by the asynchronous call !proxy.add(caller). Thus the co-interface *ClientI* is needed in the implementation of *subscribe* and *unsubscribe* in order to obtain a type-correct program, but it is not needed for the implementation of *subscribe* since *caller* is not used. By using the implicit *caller* parameter, one does not need the explicit caller parameter (ClientI me) used in the other solutions for *subscribe* and *unsubscribe*. For simplicity, we use the syntax {...} rather than **begin** ... **end**.

### II.2.4 ABS

Abstract Behavioral Specification language (ABS) [43] is an object-oriented language, inspired by Creol and JCoBox [73]. It is a concurrent programming language based on the cooperative scheduling from Creol and the notion of object groups from JCoBox, named Concurrent Object Groups (COG) [8]. Software product lines with Deltas are supported, but not class inheritance. In ABS, the unit of concurrency and distribution is the COG. Each COG includes a group of objects, a queue, and a processor. Objects in a COG share a common heap and processor, and there is no data sharing between COGs. At most one process (method activation) is active in a COG, while other processes are suspended in a process pool. In other words, parallel processes are executed by multiple threads in different COGs, but only one thread is active in a particular COG.

Objects in different COGs call each other asynchronously. Inside a COG, objects can call each other asynchronously or synchronously. The communication syntax is like Creol as well, using conditional await or await on a result/future. The statement **release** gives unconditional suspension. The **await** releases the thread if the specified condition does not hold, or if the future is not resolved (in case of await on a future), whereas the **get** $f$ statement blocks the thread until the future $f$ is resolved. Thus, the whole COG gets blocked. ABS futures are explicit, first-class, and typed by a parametric type **Fut** [$T$], where $T$ is the type of the future value.

```
1  data News = ...
2
3  interface ServiceI{
4      Bool subscribe(ClientI me)
5      Bool unsubscribe(ClientI me)
6      Void produce()
7  }
8  interface ProxyI{
9      Void publish(Fut[News] fut)
10     ...
11 }
12 interface ProducerI{
13     News detectNews()
14     ...
15 }
16 interface ClientI{Void signal(ns:News) ...}
17 interface DataBase{Void logging(...) ...}
18
19 class Service(Int limit, ProducerI prod, DataBase db) implements ServiceI {
20     ProxyI proxy:= new Proxy(limit,this); //proxy does the main job
21     {this!produce() } // initial action, starting a produce cycle
22
23     Void produce(){
24         Fut[News] fut := prod!detectNews();
25         proxy!publish(fut); // sends future, no waiting
26         db.logging(. . .) } // logging in a database
27     ...
28     Bool subscribe(ClientI me){...}
29     Bool unsubscribe(ClientI me){...}
30 }
31
32 class Proxy(Int limit,ServiceI s) implements ProxyI{
33     List[ClientI] myClients:=nil; ProxyI nextProxy;
34     ...
35     Void publish(Fut[News] fut){
36         News ns := get fut; // wait for the future
37         myClients!signal(ns);  // multi–cast the result
38         if nextProxy==null
39         then s!produce() else nextProxy!publish(fut) fi}
40 }
```

Figure II.5: A subscriber example in the ABS language.

**Example** Figure III.2 illustrates the subscriber example in ABS, making use of first-class futures, passing a future in the *publish* calls rather than the news value. In line 24, the asynchronous call creates a future identity, assigned to *fut*. Then *fut* is passed to a proxy object in line 25. In line 36, the proxy object is blocked until *fut* is resolved, after which the proxy continues to execute the next statement. For simplicity, we omit **`return`** *void* at the end of the body of a void method.

### II.2.5 Encore

Encore [9] is a parallel programming language based on active objects with explicit first-class futures, inspired by Creol and ABS. It is designed for multi-core platforms and is optimized for efficient execution. Encore supports both active object parallelism for coarse-grained parallelism, as in Creol, and parallelism within an object, using *parallel combinators* for building high-level coordination of active objects and low-level data parallelism. Encore offers high-level language constructs for coordination of parallel computations such as building pipelines of these computations. It offers parallel types, an abstraction of parallel collections, and also parallel combinators for operating on them. The parallel type **`Par T`** is a handle to a collection of parallel computations, and it can be thought of as a list of futures, which will eventually produce zero to multiple values of type **`T`**. Then operations on parallel types are called parallel combinators; accordingly, high-level typed coordination patterns, parallel dataflow pipelines, speculative evaluation and pruning, and low-level data parallel computations are supported by Encore [9].

Encore provides both passive and active classes (with the latter as default). Active objects have their own thread of control, or possibly multiple threads of control, and a FIFO message queue, and interact via asynchronous method calls. Passive objects do not have their own thread of control, like standard objects of Java. An object class is active by default or is declared as passive by a keyword **`passive`**. An asynchronous method call to an active object is stored inside the active object's queue. And the result of this asynchronous method call is a future. If the type of the return value is $T$, the returned future would be of type $Fut\ T$. Explicit synchronization constructs for accessing a future are **`get`**, **`await`**, and future chaining. Like Creol/ABS, **`get`** blocks an active object until the future is resolved, and **`await`** waits for resolving the future and blocks the current process, but not the current active object. Thus other methods of the active object can be invoked. In the chaining construct ($\sim\!\!\sim\!\!>$), a closure is attached to a future, and when resolved, the thread executes the closure, which might result in another future, containing the result of the executed closure. A closure is a set of computations, possibly including method calls.

**Example** To illustrate these operations on futures, Figure II.6 represents the subscriber example with the Encore language. In line 7, a future with identity *fut* is created as a result of an asynchronous call to the active object prod, and

```
1  passive class DataBase{...}
2  class Service(limit: int, prod: Producer, db: DataBase) {
3      proxy:= new Proxy(limit,this); //proxy does main job
4      {this.produce();} // initial action, starting a produce cycle
5      ...
6      def produce(): void{
7          let fut := prod.detectNews();
8          proxy.publish(fut); // sends future, no waiting
9      db.logging(...) } // logging in a database for services
10     def subscribe(me:Client): bool;
11     def unsubscribe(me:Client): bool;
12 }
13
14 class Proxy(limit: int, s: Service){
15     myClients: [Client];
16     nextProxy: Proxy;
17     ...
18     def publish(fut: Fut News): void{
19         ns : News;
20         ns := get fut; // wait for the future
21         myClients.signal(ns);  // multi−cast the result
22         if nextProxy==null;
23         s.produce();
24         else nextProxy.publish(fut);}
25 }
```

Figure II.6: A subscriber example in the Encore language.

then passed to the object proxy in line 8. In line 20, object proxy blocks until *fut* is resolved.

### II.2.6 ASP/ProActive

The goal of ASP [13] and ProActive [15] is to design a transparent concurrent programming language. ProActive is a Java library programming language [15], which implements ASP semantics in Java and inherits many properties from ASP. In ASP, an active object with its thread of control, its request queue, and its passive objects is called an *activity*. In addition, active objects are defined by a **newActive** command, and passive objects are standard Java objects. Only active objects are accessible between activities. Method calls to active objects are transparently turned into asynchronous calls, and those to passive objects are turned into synchronous local calls. Moreover, futures are created implicitly as a result of asynchronous method calls to an active object. In other words, an asynchronous method call is stored in the request queue of the callee, and

the caller creates a future object with a unique identity, referencing this request. When a future gets resolved, a reference to the corresponding request gets updated by the value.

ASP supports first-class futures, and its synchronization mechanism for accessing a future is *wait-by-necessity*. This synchronization mechanism blocks the thread whenever it needs to access a future value, until it is resolved. Although futures in ASP are implicit, the ASP runtime system needs constructs for implementation, update, garbage collection and synchronization of futures. It supports an explicit synchronization primitive *waitfor* which triggers an explicit *wait-by-necessity* on a future. It also provides primitives to test whether a future is updated or not. The primitive is denoted by *awaited(a)*, which returns true if $a$ is a future and false otherwise. An extension to multiactive objects has been made recently in [34].

**Example** Figure II.7 shows the subscriber example with the ProActive language. In ProActive, active objects are instantiated using the ProActive API:

$$B\ b = (B)\ ProActive.newActive(\text{"}B\text{"}, params, node);$$

It creates a new active object of type $B$, in which *params* specifies constructor parameters, and *node* specifies the location to put the active object. Another method to create an active object is by using **`turnActive(obj, node)`**, which makes an existing object (*obj*) active on a specified location (*node*). In fact, a thread is created and an associated pending request queue. In line 5, an active object proxy is defined by the **`newActive`** command, and in line 6 a passive object prod is transformed to an active one by the keyword **`turnActive`**. In addition, we assume that all object instantiations of class service is active as well. Correspondingly, all method calls toward these objects are implicitly transformed to asynchronous ones. Line 7 starts a *produce* cycle. In line 10, variable v is the result of an asynchronous *detectNews* call toward object prod, which is an implicit future. In line 11, this future is passed to object proxy without blocking. Then, in line 20, when the future value is needed to continue execution, an explicit wait-by-necessity synchronization **`waitfor`** is applied on the future v.

**Implementation strategies** To represent flow of futures and different update strategies for implicit futures, Figure II.8 is adopted from [13]. The gray arrows with numbers show the flow of futures between activities, and the black ones, indexed with letters, show the future references. In this example, activity $\gamma$ initiates a remote method call to activity $\delta$ and future $f_1$ is associated to the result of this call. Future flow number 1 corresponds to the creation of future $f_1$ involving $\gamma$ and $\delta$. Then $\gamma$ sends $f_1$ to $\beta$, for instance as the result of a request, flow number 2, and $\beta$ forwards the $f_1$ reference to $\alpha$ as the result of another request, flow number 3a. In parallel, $\gamma$ sends a request to $\alpha\prime$ with $f_1$ as a request parameter, flow number 3b. Finally, $\delta$ consumes the result associated with $f_1$, flow number 4.

```
1  class Service(Int limit, ProducerI prod, DataBase db) extends ServiceI
2  implements Active {
3      // assuming active instantiation of object service
4      Object [ ] params= {limit,this}
5      Proxy proxy:= (Proxy) ProActive.newActive ("Proxy", params, Node);
6      prod = (Producer) ProActive.turnActive (prod, Node);
7      {this.produce(); }
8
9      void produce(){
10         News v := prod.detectNews();//implicit asynchronous call
11         proxy.publish(v); // v can be passed without blocking
12         db.logging(. . .) }
13         ...
14 }
15
16 class Proxy(Int limit,ServiceI s) extends ProxyI implements Active{
17     Client[ ] myClients:=null; ProxyI nextProxy;
18     ...
19     void publish(News v){
20         News ns := waitfor(v); // explicit wait-by-necessity on v
21         myClients.signal(ns);
22         if nextProxy==null;
23         then s.produce(); else nextProxy.publish(v);}
24 }
```

Figure II.7: A subscriber example in the ProActive language.

In the case of first-class futures, a future value list $F_\alpha$ inside an activity stores future values calculated by itself or other activities. There are three strategies for updating a future value: 1) *no partial object forwarding*, 2) *eager strategy*: either *forward-based* or *message-based*, and 3) *lazy strategy*. The simplest strategy is that no partial object (the objects containing future references or values) can be forwarded between activities. In other words, futures are not first-class. This strategy leads to fewer number of future references, simpler update process, and avoids maintaining a future value list inside an activity. However, it is too synchronized and may lead to waste of time and deadlock. For example, according to Figure II.8, while $\gamma$ is waiting for a response from $\delta$, other activities are stuck (waste of time).

The second strategy is called the *eager strategy*, a future gets updated as soon as it is resolved, thus future value lists are avoided. In the case of *forward-based strategy*, when a future reference is sent to an activity, the sender is responsible for updating its value, but not the source activity; hence the source activity does not need to keep the future value any longer. Consequently, when there are too many intermediate nodes, this strategy increases the delay between when

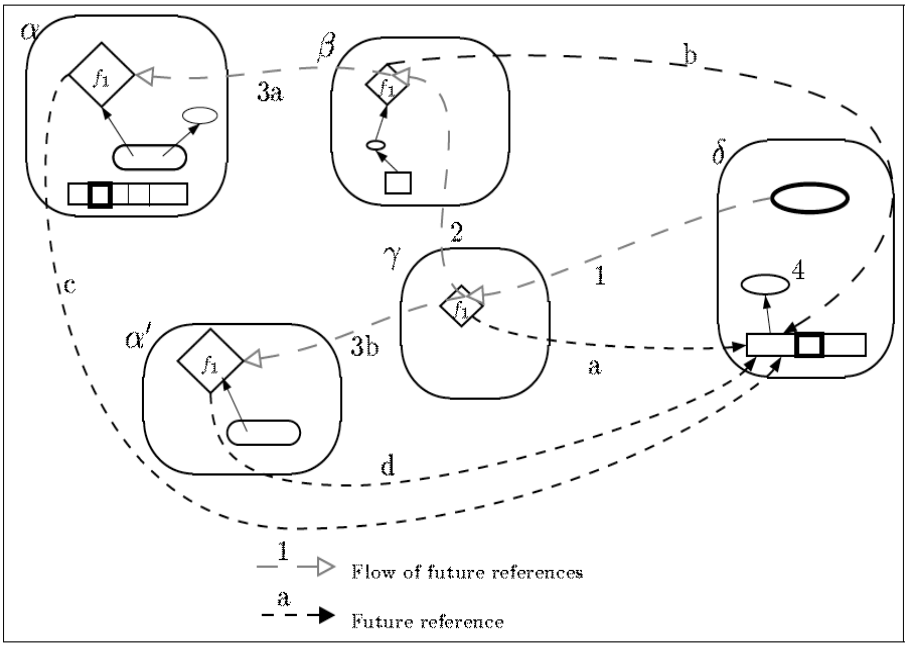


Figure II.8: Future flow in ASP [13].

a future is resolved and when it gets updated. In Figure II.8, when based on this strategy, the future $f_1$ is first updated in $\gamma$, then it sends this value to $\beta$, $\alpha'$, and then $\beta$ can forward this value to $\alpha$. Consequently, there is a delay before updating the future value of $\alpha$.

In the case of *message-based strategy*, when a future is forwarded between two activities, a message, created from the receiver or the sender activity, is sent to the source activity. Hence the source activity gets informed about them, and when the future value is calculated, it can directly update it in all other activities. This message-based mechanism minimizes the delay of updating. For example, in Figure II.8, when $\gamma$ sends the future reference $f_1$ to $\alpha'$, either $\gamma$ or $\alpha'$ sends a message to the source activity $\delta$; correspondingly, when the future value is calculated in $\delta$ it can be immediately updated in $\alpha'$.

The third strategy is the *lazy* future update. A future gets updated only when an activity requires it (wait-by-necessity). The activity directly asks for the future value by sending a message to the source activity. In the lazy strategy, a future value list is required to be kept in the source activity in order to store the future values and update them whenever there is a request for them.

The implementation of ProActive supports several update strategies, including no partial object forwarding and forward-based strategy [13], and Henrio et al. have implemented the four strategies in ProActive as a middleware to study the efficiency of different update strategies [13]. The lazy strategy is faster than

the two eager ones, since less updates are required. This strategy is suitable for scenarios in which the number of processes requiring the future value are considerably less than the total number of processes. In this strategy, there is less load at the source process. However, it leads to additional delays and needs more resources to keep the future value list inside the source process for later updates. The eager forward-based strategy gives more delay since the intermediate nodes have to forward a future value to a target. As a result, this strategy is suitable for scenarios with small number of nodes. Eager message-based strategy necessitates more bandwidth and resources at the source process, since all other nodes communicate with it.

To complement the discussion we introduce some additional language features and a new language called *FutFree*, in the next section.

### II.2.7 A proposed future-free language with improved expressiveness (*FutFree*)

As we have seen, implicit futures have the weakness that the implicit occurrences of the **get** operation cannot always be identified in a context-free manner. This makes modular reasoning and static analysis difficult. And a weakness of the future-free paradigm, as represented by the future-free subsets of the languages discussed above, is that there is no way of expressing that a **get** operation should be performed in a given state. For the purpose of this paper, we therefore propose a new language exploiting the future-free interaction paradigm while adding a new mechanism for non-blocking waiting. The language is an extension of the future-free subset of Creol/ABS and is called *FutFree*. We do this in order to complement the comparison of languages with and without futures.

*FutFree* is without call-labels, and without explicit/implicit futures, and consists of the asynchronous call statement $o!m(\overline{e})$, the high-level future-free call mechanisms of Creol, including the **await** construct, extended with a *"tail"* construct to better control return value points, and a *delegation* mechanism to enable simple sharing (to an object other than the caller). The tail construct may or may not be combined with **await**:

$$[\textbf{await}]\ x := o.m(\overline{e}) < s >$$

where the tail $s$ is any statement list, being performed while waiting for the future of the call to $m$ to be resolved, and therefore $s$ may not use (the new value of) $x$. The syntax [**await**] denotes an optional **await**, allowing the statement to suspend after $s$ if the future is not resolved at that point. Thus, the statement $x := o.m(\overline{e}) < s >$ will block after execution of $s$ while waiting for the return value to appear (if it has not already appeared). And the statement **await** $x := o.m(\overline{e}) < s >$ will suspend after execution of $s$ while waiting for the return value to appear. The former statement is equivalent to $f := o!m(\overline{e}); s; x :=$ **get** $f$, using ABS, and the latter is equivalent to $f := o!m(\overline{e}); s;$ **await** $x :=$ **get** $f$.

Note that $< s >$ may be empty or include additional calls as in for instance

$$[\textbf{await}]\ x := o1.m1(\overline{e1}) < calculate\ e2; [\textbf{await}]\ y := o2.m2(\overline{e2}) < s >; use\ y >$$

Here the first suspension point is after $s$, passively waiting for the *last* call to complete (receiving the result in $y$), and the second suspension point passively waits for the $m1$ call to complete. With respect to the expressiveness of this construct, we observe that programs with nested call-get structures can be expressed without futures. In particular, one may wait for completions of several calls and continue when all calls have completed by letting the right brackets (“>”) stand together, as in:

$$[\textbf{await}]\ call1 < ...; [\textbf{await}]\ call2 < ... >>$$

To await completions in one specific order, one would need to make the calls in the opposite order, as in

$$[\textbf{await}]\ call1 < ...; [\textbf{await}]\ call2 < ... >; ... >$$

where the return value of $call2$ is handled before that of $call1$. We use **await** when passive waiting is desired.

**Delegation** Consider the case that a method body (say method $m$) ends by returning the result of a call to $n$, as in the method body $\{...; x := o.n(\overline{e}); \textbf{return}\ x\}$ where $x$ is a local variable in the body. Here the result of the call is not used by the current process, and it may be desirable to avoid the waiting. With futures this could be done efficiently by returning a future, as in $\{...; f := o!n(\overline{e}); \textbf{return}\ f\}$ changing the type of the return value accordingly. The same efficiency can be achieved in the future-free setting by a form of *delegation* [58]. The body of $m$ is now written as $\{...; \textbf{delegate}\ o.n(\overline{e})\}$. The statement

$$\textbf{delegate}\ o.n(\overline{e})$$

makes the current call (of $m$) terminate without producing a result, while delegating to the remote call $o.n(\overline{e})$ to send a result back to the caller of $m$. Type checking must ensure that the result type of $n$ is appropriate. This gives the same efficiency for the current process as in the solution with futures, and without the need to change the return type.

*FutFree* is more high-level than languages with futures or call labels, since the syntactic complication of futures/labels is avoided, and it is more expressive than the future-free restriction of both Creol and ABS. However, *FutFree* does not support non-parenthetic nesting of calls and returns, nor delegation to more than one object.

A weakness of the tail construct is that it involves blocking at the end of the tail. The delegation mechanism **delegate** $o.n(\overline{e})$ avoids this waiting point and still makes use of the result. If $o$ here is this, a local continuation (i.e., relative to the caller) will be triggered asynchronously. This is a bit similar to a continuation

```
1  class Service(Int limit, ProducerI prod) implements ServiceI {
2      ProxyI proxy:= new Proxy(limit,this); //proxy does the main job
3      {this!produce{}} // initial call
4      ...
5      Void produce(){ News ns;
6      await ns := prod.detectNews()                    // no blocking
7      < db.logging(...) >;
8      proxy!publish(ns)}                                 // send the news
9  }
10 class Proxy(Int limit,ServiceI s) implements ProxyI{
11     List[ClientI] myClients:=Nil; ProxyI nextProxy;
12     ...
13     Void publish(News ns){
14         myClients!signal(ns); // multi–cast the result
15         if nextProxy==null
16         then s!produce() else nextProxy!publish(ns)}
17 }
```

Figure II.9: The Publishing Example rewritten in the future-free language

associated to a future, which could be added as a suspended process and enabled when the associated future is resolved. Such a continuation should then maintain the class invariant. When a method body contains a number of continuations, there may be a need for coordination of the different activities. Encore has solutions for this using futures [9]. A syntax for this concept of asynchronous continuation can also be made in the future-free setting (i.e., without explicitly mentioning the associated future). However, we will not add further syntax to *FutFree* since this can be simulated (even though less elegant). The concept of delegation is also useful in the setting of languages with futures, avoiding one level of future referencing, reducing a result of type $Fut[T]$ to $T$.

**Example** In Figure II.9 we show the publishing example rewritten to the new format without futures (with changes in blue). This is essentially the same solution as that in Creol, Figure II.4, but expressed without call labels/local futures. We also assume interfaces as in the Creol. The changes are straight forward. The *pl*Service object makes the same call to *pl*publish, but at a later time, when the news are available. By using a suspending *pl*publish call, the *pl*Service object is not blocked and has therefore similar efficiency as in the first version. And the blocking that used to be in the *pl*Proxy object is removed, and thus the *pl*Proxy objects will be more responsive. This indicates that passing futures as parameters can be avoided without loss of efficiency by using asynchronous call/return in combination with suspension. Instead of having a responsive *pl*Service object at the cost of blocking in *pl*Proxy objects, as in the original version, the future-free version has a responsive *pl*Service object as well as *pl*Proxy objects, but now the process queue of the *pl*Service object may be non-empty. Thus there is less need for first-class futures in a future-free language with cooperative scheduling than in one without. In general one may

| | ABCL | Rebeca | Creol | ABS | Encore | ASP | *FutFree* |
|---|---|---|---|---|---|---|---|
| futures | yes | no | yes | yes | yes | no | no |
| syntactic | yes | yes | yes | yes | yes | no | yes |
| first-class | yes | no | no | yes | yes | yes | no |
| cooperative | no | no | yes | yes | yes | no | yes |
| polling | yes | no | no | no | no | yes | no |
| dyn. creation | yes | no | yes | yes | yes | yes | yes |
| passive data | obj | obj | adt | adt | obj | obj | adt |

Figure II.10: Overview of future support in the selected languages.

use the process queue and suspension rather than blocking separate object(s) in get statements. Forwarding a future can be replaced by a suspended process forwarding the future value. And this gives a deadlock-free solution.

The versions in Figures II.9 and III.2 are similar in that the Service objects are not blocked (apart from the logging part). And if there are no other *produce* and *publish* calls than the ones in Service and Proxy, there will be at most one uncompleted *produce* process and also at most one uncompleted *publish* process, for either version. For the version in Figure II.9 there is an explicit interleaving point after the logging, whereas in the other version (in Figure III.2) this interleaving is implicit since the *produce* process terminates. Thus the two versions may give rise to different executions.

### II.2.8 A summary of active object languages

A summary of the main interaction-related features of some different languages for active objects is given in Figure II.10. In the table, the entry “futures" is indicating whether a language supports the future implementation explicitly or not. The entry “syntactic" shows whether the waiting points are syntactically identified or not, in the context of a given class. For example, in ASP when a future is passed to an activity and its value is needed inside that, it is not textually clear whether it is a waiting point or not. For Rebeca the “yes” refers to the extension with *wait*. Moreover the table compares whether these languages support first-class, cooperative, polling, cooperative scheduling or not. The table entry “dyn. creation" indicates if dynamic object creation is supported, and the entry “passive data" indicates how temporary internal data structure is built, either by means of (passive) objects without their own execution thread, marked “obj”, or user-defined data types, marked “adt”.

An advantage of explicit futures is that it provides explicit (syntactic) identification of waiting points, typically by the **`get`** construct. For the category of languages with implicit future support, we may distinguish between those with explicit and implicit identification of waiting points.

The table summarizes the support of futures and related concepts for the different languages. The considered languages are chosen to give a certain variety, all in the setting of imperative programming with active objects.

For the category of implicit futures, there is a distinction between languages where the waiting points are syntactically given and those where they are not, as in the case of wait-by-necessity where the usage of a variable $x$ bound to the result of a method call requires that the value is available. Thus sending $x$ as a parameter does not require that the value is there, but updating or testing the value would normally require the value to be available. For instance, this means that a method $m(Int\ x)\{s_1; y := x+1; s_2\}$ will have an implicit potential waiting point at $y := x + 1$ when $m$ is called with an actual parameter representing an implicit future (and when $s_1$ does not use $x$) whereas there is no such waiting point when $m$ is called with an actual parameter representing an available value. This makes program reasoning complicated, for instance deadlock reasoning, and modular semantics is not possible since waiting depends on external objects. We will therefore limit the discussion to languages with explicit waiting points.

Operations on futures can be blocking, such as getting the result from a future, suspending, or it can be asynchronous and non-blocking, such as attaching a callback or a continuation to a future. For example, in AmbientTalk the future access is a non-blocking asynchronous operation, in which actors desiring a future value are registered as observers, then when the future is resolved, its value is sent to these registered observers. An observer actor can register a closure, a block of code, to a future that is applied when the future gets resolved [20]. An extended version of Encore [27] also offers high-level non-blocking coordination constructs operating on futures. For instance, it can apply a function on the first result of a bunch of futures and terminate the computations associated with the other futures. It offers complex coordination, including pipe-lining and speculative parallelism on futures, when they might be dependent on other futures. To have these complex coordinated workflows, they offer a new non-blocking asynchronous parallel abstraction $ParT$ (or $Par\ T$), which is a handle to parallel computations or a data structure for collecting future values, and also offer parallel combinators to operate on. Parallel combinators are non-blocking constructs that control and coordinate $ParT$ collections without blocking threads.

A recent survey [8] compares several active object languages such as ABS, Encore, Rebeca and ASP/Proactive according to their design aspects, the degree of synchronization, the degree of transparency and the degree of data sharing. It identifies the design purpose of these languages. For example Creol, ABS, and Rebeca are designed with program analysis in mind, while Encore and ASP/ProActive are optimized for efficient execution. The degree of synchronization is compared according to their synchronization primitives. This survey compares explicit and implicit futures as a degree of transparency. The degree of data sharing between active objects are compared as well. None of the active object languages support data sharing, apart from futures, since they are oriented toward distributed systems, where copying and sending data is more safe and efficient.

## II.3 Unified Syntax and Semantics

Based on the overview above, we consider the main categories of language support for interaction mechanisms, given by

- support of first-class futures, object-local and method-local futures, and future-free (asynchronous call/return) interaction
- cooperative scheduling or not
- polling or not (for languages with futures)

This gives the following main categories for interaction mechanisms, where the six first categories (those with futures) may be with or without polling: i) cooperative scheduling and first-class futures with ABS/Encore as representatives, ii) cooperative scheduling and object-local futures, iii) cooperative scheduling and method-local futures with Creol as a representative, iv) non-cooperative scheduling and first-class futures with something in the direction of ABCL as a representative, v) non-cooperative scheduling and object-local futures, vi) non-cooperative scheduling and method-local futures, vii) cooperative scheduling and no support of futures, with *FutFree* as a representative, and viii) non-cooperative scheduling and no support of futures, with Rebeca (or the await-free restriction of *FutFree*) as representative. However, basic Rebeca does not use the call/reply paradigm.

As mentioned, we restrict ourselves to imperative languages, and assume syntactic identification of any waiting points. In particular, to keep the discussion uniform, we use the following ABS-inspired syntax for the different language mechanisms representing the basic ways of using futures or asynchronous call/return interaction without futures.

### II.3.1 Syntax

The unified syntax is given in Figure III.1. We assume static type checking. The different language combinations are obtained by including/excluding polling, await, first-class, or object/method-local futures. In the case of method-local futures, futures may not be assigned to fields nor passed as parameters or method results. In the case of object-local futures, futures may be assigned and passed as parameters/result but only for local methods (methods not exported through any interface). This guarantees static control of what is legal. For first-class future languages, futures may be assigned and passed to parameters/result to any method (modulo static typing restrictions).

Moreover, the different language combinations are achieved by taking the basic and future-free constructs, adding no futures, or adding futures, either with first-class future operations (assignment and parameter passing), no first-class future operations (other than assignment to local variables) for the case of method-local futures, or assignments to future variables and passing of futures in local methods for the case of object-local futures, and furthermore adding suspension

| *Basic constructs* | |
|---|---|
| $x := e$ | assignment ($x$ a variable, $e$ an expression) |
| $x :=$ **new** $C(\overline{e})$ [**at** $o$] | object creation ($\overline{e}$ actual class parameters) |
| **return** $e$ | creating a method result/future value |
| **if** $c$ **then** $s$ [**else** $s'$] **fi** | if-statement ($c$ a Boolean condition) |
| **while** $c$ **do** $s$ **od** | while-statement |

| *Future-free constructs* | |
|---|---|
| $o!m(\overline{e})$ | simple asynchronous call, non-blocking |
| [**await**] $x := o.m(\overline{e})$ | blocking/non-blocking call |
| [**await**] $x := o.m(\overline{e}) < s >$ | blocking/non-blocking call with tail |
| **delegate** $o.m(\overline{e})$ | termination and delegation |

| *Basic future constructs* | |
|---|---|
| $f := o!m(\overline{e})$ | asynchronous call, non-blocking |
| [**await**] $x :=$ **get** $f$ | getting a future value |

| *Cooperative scheduling* | |
|---|---|
| **await** $c$ | conditional suspension |
| **await** $f?$ | await on a future |

| *Polling* | |
|---|---|
| $f?$ | checking if a future is resolved |

Figure II.11: Unified Syntax. Here $f$ is a declared future variable and $x$ an ordinary program variable, $s$, $s'$ denote statement lists, and [...] denotes optional parts.

(by the **await** keyword) or not, and adding polling or not for combinations with futures. Furthermore, languages with method-local futures may be divided further by allowing multiple-read or single-read futures. Single-read futures depend on static constraints ensuring that each path through a method has at most one read of a given future, whereas multiple-read languages are free from this restriction. Thus we cover in all *eighteen language combinations.*

The future-free constructs (including tail) are included in all language combinations, however, some of these constructs can be omitted in languages with futures since they can be expressed by means of futures. For instance, the following statements are inter-definable: The call [**await**] $x := o.m(\overline{e})$ is the same as [**await**] $x := o.m(\overline{e}) < skip >$, and the call [**await**] $x := o.m(\overline{e}) < s >$ may be simulated by means of futures by $f := o!m(e); s;$ [**await**] $x :=$ **get** $f$, for some fresh future variable $f$, where [**await**] $x :=$ **get** $f$ again can be seen as a shorthand for [**await**] $f?; x :=$ **get** $f$. Furthermore, we may extend the **await** notation to a conjunction of futures. We use dot-notation for suspending/blocking calls and ! for making an invocation request.

Polling of a future means checking if a future is resolved or not, for instance

in an if-test, say

$$\textbf{if } f? \textbf{ then } x := \textbf{get } f \textbf{ else } \ldots \textbf{ fi}$$

Polling may lead to complicated branching structures, and is often avoided in languages with support of explicit futures. Basic constructs are quite standard, but object creation has as an optional part (**at** $o$) for specifying the placement of the new object. The default is **at** this, i.e., the same location as the parent object.

### II.3.2 Operational Semantics

In this part, we present (relevant parts of) the operational semantics of the different language combinations, including future-free, method-local, object-local, and first-class future languages, using the style of structural operational semantics. The purpose of the operational semantics is to show the underlying asynchronous communication between active objects at runtime. Each rule in the operational semantics corresponds to one execution step.

A system state is given by a configuration, which is a multi-set of objects and messages, either invocation messages or reply messages, in addition to future objects in the case of first-class future languages. An object is represented as

$$\textbf{ob}(o \mid s, a, l, q)$$

where $o$ denotes the object identity, $s$ is the list of active statements, $a$ the state of object fields (attributes), $l$ the state of local variables defined in a method (including the parameters), and $q$ the internal process queue. The process queue is only needed in languages with cooperative scheduling. The pair $(s \,\&\, l)$ represents (the remaining part of) the active process, with statements $s$ and local state $l$. At suspension, this active process is moved to the process queue, making the object *idle*. When idle, an object may continue with another (enabled) process from the process queue, or start a new method invocation. We use the syntax

$$\textbf{invoc}\,(o, u, m, \overline{d})\,\textbf{to}\;\; o'$$

for an invocation message from object $o$ to object $o'$ where $m$ is the name of the called method, $u$ the identity of the call/future, and $\overline{d}$ the list of actual parameters. An object will have an associated invocation queue, and also a reply queue in the case of non-first-class future languages. We let $\textbf{iq}(o \mid p)$ denote the invocation queue associated with $o$ containing messages $(p)$ to $o$.

It is worth mentioning that an object identity is unique, which can be achieved by using the parent object identity and a counter [39]. Similarly a unique call or future identity $u$ can be achieved by using the caller object identity and a separate counter. The counters can be represented by implicit fields in $a$. In the case of first-class future languages, this identity acts as the future identity.

The operational semantics is given by a number of rewrite rules. A rule can be applied to a configuration if the left-hand-side matches a subset of the

configuration (possibly reordered). If the left-hand-sides of two rules match disjoint parts, they can be applied at the same time, reflecting concurrent behavior. Each rule involves at most one object, reflecting that the objects are executing independently from each other. Thus the objects execute in parallel. Non-determinism is achieved when (at least) two left-hand-sides match overlapping parts of a configuration.

States are given by mappings from variable names to values, and $l[x \mapsto d]$ denotes the local state $l$ updated so that variable $x$ binds to data value $d$ (adding such a binding if $x$ is not bound in $l$). We use $+$ for map composition with overwriting, so that $a + l$ is the union of map $a$ and $l$, using $l$ for variable names with a binding in both $a$ and $l$, reflecting that the binding of a variable name in the inner scope $l$ shadows any binding of that name in the outer scope $a$. Therefore, $a + l$ represents the total state of an object as used for evaluation. For an expression $e$, the notation $[e]_{a+l}$ abbreviates the evaluation of $e$ in the context of $a + l$. For a variable $x$, the evaluation $[x]_{a+l}$ equals $[x]_l$ if $l$ has a binding for $x$, otherwise $[x]_a$, i.e., the value of a variable $x$ is found in $l$ if $l$ has a binding for $x$, otherwise in $a$. Therefore the semantics of an assignment statement is given by two rules; one for the case that $x$ is a local variable and one for the case that $x$ is a field:

$$
\begin{array}{lll}
\textsf{assign-local}: & \mathbf{ob}(o \mid x := e; s, a, l, q) & \\
\longrightarrow & \mathbf{ob}(o \mid s, a, l[x \mapsto [e]_{a+l}], q) & \\
\mathbf{if} & x \in l &
\end{array}
$$

$$
\begin{array}{lll}
\textsf{assign-field}: & \mathbf{ob}(o \mid x := e; s, a, l, q) & \\
\longrightarrow & \mathbf{ob}(o \mid s, a[x \mapsto [e]_{a+l}], l, q) & \\
\mathbf{if} & x \notin l &
\end{array}
$$

Type checking ensures that $x$ is defined in $l$ or $a$. Similarly there are two rules for if-statement, depending on the value of the if-condition.

$$
\begin{array}{lll}
\textsf{if-then}: & \mathbf{ob}(o \mid \mathbf{if}\ c\ \mathbf{then}\ s1\ \mathbf{else}\ s2\ \mathbf{fi}; s, a, l, q) & \\
\longrightarrow & \mathbf{ob}(o \mid s1; s, a, l, q) & \\
\mathbf{if} & [c]_{a+l} = true &
\end{array}
$$

$$
\begin{array}{lll}
\textsf{if-else}: & \mathbf{ob}(o \mid \mathbf{if}\ c\ \mathbf{then}\ s1\ \mathbf{else}\ s2\ \mathbf{fi}; s, a, l, q) & \\
\longrightarrow & \mathbf{ob}(o \mid s2; s, a, l, q) & \\
\mathbf{if} & [c]_{a+l} = false &
\end{array}
$$

The semantics of the call $[\mathbf{await}]\ x := o.m(\overline{e})$ is given by that of $f := o!m(e); s; [\mathbf{await}]\ x := \mathbf{get}\ f$, for some fresh future variable $f$. And similarly the semantics of the call $o!m(\overline{e})$ is given by that of $f := o!m(e)$, for some fresh future variable $f$. Thus the semantics of the future-free languages is given by means of futures at the run-time level.

The operational semantics of languages with first-class futures differs from those with local futures or without futures, since they need a representation of shared future objects, and we therefore consider the two classes of operational semantics in two subsections.

### II.3.2.1 Operational semantics of languages without first-class futures

In the case of local or no futures, we need to deal with reply messages, and let **reply** $(u, v)$ **to** $o$ represent a reply message to the caller $o$, where $u$ is the call identity and $v$ the returned value. Each object $o$ must keep track of the received reply messages, in a reply queue **rq**$(o \,|\, r)$, where $r$ is a queue of $(u, v)$ pairs. When a **get** $f$ operation is executed, one must check if there is a value for $f$ in the reply queue. Thus the order of the reply queue is irrelevant, and the handling of the internal process queue can be seen as non-deterministic. Call identities are invisible to the programmer and cannot be passed to other objects (where these call identities would be unknown). For a statement containing **get** $f$, the future (expression) $f$ must first be evaluated. Therefore **ob**$(o \,|\, [\textbf{await}]\ x := \textbf{get}\ f; s, a, l, q)$ reduces to **ob**$(o \,|\, [\textbf{await}]\ x := \textbf{get}\ [f]_{a+l}; s, a, l, q)$.

Figure II.12 defines the operational semantics for local futures, with or without suspension (ignoring all rule instances with **await** in the latter case), and with polling if including polling rules similar to those in Figure II.13 (using the reply queue for checking the presence of a future). The difference between object-local and method-local futures is not visible in the rules, but in the underlying language, restricting assignment and passing of futures (in the ways mentioned). The operational semantics for the future-free languages is obtained by replacing future-free calls by calls with futures as explained above, and allowing/disallowing suspension as desired. The syntax [**await**] denotes an optional **await**, $o$ and $o'$ denote object identities, and an object is **idle** when there is no active process. In a left hand side or condition, the symbol "_" matches any term.

The async-call rule executes an asynchronous call where $o$ is the caller invoking method $m$ of object $e$ with $\overline{e}$ as actual parameters. This rule generates a globally unique call identity $(u)$, which is assigned to the future variable $f$. The call creates the invocation message **invoc** $(o, u, m, [\overline{e}]_{a+l})$ **to** $[e]_{a+l}$, where the actual parameters $\overline{e}$ and the callee $e$ are evaluated before sending this message.

The start rule says that when an object is **idle** and there is an invocation message in its invocation queue, the object may start to execute the corresponding method. It then captures the method's body $s$ as its active process with local variables given by the method declaration (bound to default values), binding formal parameters to the actual ones, and storing the caller object in the (implicit) caller parameter and the call identity in the (implicit) callid parameter. These two implicit variables are needed to execute the return statement.

The reply rule represents the case when an object $o$ returns a value $e$ to the caller $o'$. It creates a reply message with the callid as its first argument and the evaluation of $[e]_{a+l}$ as the value and forwards this message back to the caller, as given by the local variable caller. We assume each method body ends with a return statement.

The get-reply rule describes how a reply message to an object $o$ with label $u$ and return value $v$ enters into the reply queue (**rq**$(o \,|\, r)$) of object $o$.

The get, suspend, await-pq rules represent query statements for retrieving

async-call :
$\mathbf{ob}(o \mid f := e!m(\overline{e}); s, a, l, q)$
$\longrightarrow$ $\mathbf{ob}(o \mid f := u; s, a, l, q)$
$\mathbf{invoc}\,(o, u, m, [\overline{e}]_{a+l})\ \mathbf{to}\ [e]_{a+l}$
**where** $u$ is a fresh locally unique identity

invocation :
$\mathbf{invoc}\,(o, u, m, \overline{d})\ \mathbf{to}\ o'$
$\mathbf{iq}(o' \mid p)$
$\longrightarrow$ $\mathbf{iq}(o' \mid p \cdot \mathbf{invoc}(o, u, m, \overline{d}))$

start :
$\mathbf{iq}(o \mid \mathbf{invoc}(o', u, m, \overline{d}) \cdot p)$
$\mathbf{ob}(o \mid \mathbf{idle}, a, \mathit{empty}, q)$
$\longrightarrow$ $\mathbf{iq}(o \mid p)$
$\mathbf{ob}(o \mid s, a, [\mathsf{caller} \mapsto o', \mathsf{callid} \mapsto u, \overline{x} \mapsto \overline{d}, \overline{l} \mapsto \overline{l_0}\,], q)$
**where** method $m$ binds to $m(\overline{x})\{\overline{T\ l}; s\}$
(with initial values $\overline{l_0}$ of $\overline{l}$)

reply :
$\mathbf{ob}(o \mid \mathbf{return}\ e, a, l, q)$
$\longrightarrow$ $\mathbf{ob}(o \mid \mathbf{idle}, a, \mathit{empty}, q)$
$\mathbf{reply}\,([\mathsf{callid}]_l, [e]_{a+l})\ \mathbf{to}\ [\mathsf{caller}]_l$

get-reply :
$\mathbf{reply}\,(u, v)\ \mathbf{to}\ o$
$\mathbf{rq}(o \mid r)$
$\longrightarrow$ $\mathbf{rq}(o \mid r \cdot (u, v))$

get :
$\mathbf{rq}(o \mid r)$
$\mathbf{ob}(o \mid [\mathbf{await}]\ x := \mathbf{get}\ f; s, a, l, q)$
$\longrightarrow$ $\mathbf{rq}(o \mid r)$
$\mathbf{ob}(o \mid x := v; s, a, l, q)$
**if** $([f]_{a+l}, v) \in r$

suspend :
$\mathbf{rq}(o \mid r)$
$\mathbf{ob}(o \mid \mathbf{await}\ x := \mathbf{get}\ f; s, a, l, q)$
$\longrightarrow$ $\mathbf{rq}(o \mid r)$
$\mathbf{ob}(o \mid \mathbf{idle}, a, \mathit{empty}, q \cdot (\mathbf{await}\ x := \mathbf{get}\ [f]_{a+l}; s \,\&\, l))$
**if** $([f]_{a+l}, \_) \notin r$

await-pq :
$\mathbf{rq}(o \mid r)$
$\mathbf{ob}(o \mid \mathbf{idle}, a, \mathit{empty}, q \cdot (\mathbf{await}\ x := \mathbf{get}\ u; s \,\&\, l) \cdot q')$
$\longrightarrow$ $\mathbf{rq}(o \mid r)$
$\mathbf{ob}(o \mid x := v; s, a, l, q \cdot q')$
**if** $(u, v) \in r$

tail :
$\mathbf{ob}(o \mid [\mathbf{await}]\ x := e.m(\overline{e}) < s >; s', a, l, q)$
$\longrightarrow$ $\mathbf{ob}(o \mid f := e!m(\overline{e}); s; [\mathbf{await}]\ x := \mathbf{get}\ f; s', a, l[f \mapsto \mathit{nil}], q)$
**where** $f$ is a fresh local (future) variable

Figure II.12: Operational rules for local futures and future-free statements.

the reply. For the first two rules the query is already in the active process, and for the third it is in the internal queue. The reply value is not removed from the reply queue, thus multiple reads are possible. (For single-read futures it should be removed.) When a reply is needed and the reply message is in the queue, the response value $v$ is fetched. The suspend rule takes care of the case when a reply is needed in an await statement, but the corresponding reply message is not in the queue. Then the whole process with its local variables are suspended and placed at the end of the internal queue. The notation $q \cdot z$ denotes that a process $z$ is appended to $q$. A suspended process with the state of its local variables are represented by a pair, written $(\textbf{await}\ x := \textbf{get}\ u; s \,\&\, l))$, where $\textbf{await}\ x := \textbf{get}\ u; s$ is the suspended process with local state $l$.

The tail rule says that we implicitly create a fresh local variable $f$ (like a future) to talk about the value resulting from the method call. Then in the next step, the async-call rule binds the $f$ variable to a unique identity $u$. We ignore here the rules for other statements such as if and while, since they are not central to our discussion.

The rules in Figure II.12 can be used to define object-local futures as well as method-local futures and future-free languages, restricting the language accordingly (disallowing passing of futures in non-local methods, disallowing all passing of futures and assignments of futures to fields, and disallowing the use of futures variables in a program, respectively). In the latter case, future variables and the **get** statement are not part of language syntax. However, call identities and **get** statements are implicitly generated by the rules. Asynchronous calls, which use futures, are therefore not part of the language as seen by the programmer, but simple asynchronous call is. The rule for a simple asynchronous call is similar to that of asynchronous call, i.e.,

$$\textbf{ob}(o \mid e!m(\overline{e}); s, a, l, q) \longrightarrow \textbf{ob}(o \mid s, a, l, q) \quad \textbf{invoc}\,(o, u, m, [\overline{e}]_{a+l})\,\textbf{to}\ [e]_{a+l}$$

where $u$ is locally fresh as before. There is no state update since there is no future variable involved. (In fact, one could use *nil* instead of $u$ in the invocation message to indicate that no return is needed.)

As mentioned, the rules in Figure II.12 define object-local and method-local futures. For both these language classes we can define the versions without cooperative scheduling by removing the process queue ($pq$) and removing rules involving **await** statements. However, the addition of first-class futures requires sharing of futures, which is not possible with the rules of Figure II.12. This is considered below.

The rules for polling (given for the case of first-class futures in Figure II.13) describe that the test $f?$ gives true if and only if the future is resolved/the reply is in the reply queue.

#### II.3.2.2 Operational semantics of languages with first-class futures

In Figure II.13, we define operational semantics of first-class futures, representing a future as a global object with a unique identity (the future/call identity). We let $(\textbf{fut}(u \mid ))$ denote an unresolved future object with identity $u$, and $(\textbf{fut}(u \mid v))$

async-call' :
$$\begin{array}{ll} & \mathbf{ob}(o \mid f := e!m(\overline{e}); s, a, l, q) \\ \longrightarrow & \mathbf{ob}(o \mid f := u; s, a, l, q) \\ & \mathbf{fut}(u \mid) \\ & \mathbf{invoc}\,(o, u, m, [\overline{e}]_{a+l})\ \mathbf{to}\ [e]_{a+l} \\ & \mathbf{where}\ u \text{ is a fresh and globally unique identity} \end{array}$$

reply' :
$$\begin{array}{ll} & \mathbf{fut}(u \mid) \\ & \mathbf{ob}(o \mid \mathbf{return}\ e, a, l, q) \\ \longrightarrow & \mathbf{fut}(u \mid [e]_{a+l}) \\ & \mathbf{ob}(o \mid \mathbf{idle}, a, \mathit{empty}, q) \\ \mathbf{if} & [\mathsf{callid}]_l = u \end{array}$$

get' :
$$\begin{array}{ll} & \mathbf{fut}(u \mid v) \\ & \mathbf{ob}(o \mid [\mathbf{await}]\ x := \mathbf{get}\ f; s, a, l, q) \\ \longrightarrow & \mathbf{fut}(u \mid v) \\ & \mathbf{ob}(o \mid x := v; s, a, l, q) \\ \mathbf{if} & [f]_{a+l} = u \end{array}$$

suspend' :
$$\begin{array}{ll} & \mathbf{fut}(u \mid) \\ & \mathbf{ob}(o \mid \mathbf{await}\ x := \mathbf{get}\ f; s, a, l, q) \\ \longrightarrow & \mathbf{fut}(u \mid) \\ & \mathbf{ob}(o \mid \mathbf{idle}, a, \mathit{empty}, q \cdot (\mathbf{await}\ x := \mathbf{get}\ u; s \,\&\, l)) \\ \mathbf{if} & [f]_{a+l} = u \end{array}$$

await-pq' :
$$\begin{array}{ll} & \mathbf{fut}(u \mid v) \\ & \mathbf{ob}(o \mid \mathbf{idle}, a, \mathit{empty}, q \cdot (\mathbf{await}\ x := \mathbf{get}\ u; s \,\&\, l) \cdot q') \\ \longrightarrow & \mathbf{fut}(u \mid v) \\ & \mathbf{ob}(o \mid x := v; s, a, l, q \cdot q') \end{array}$$

polling1 :
$$\begin{array}{ll} & \mathbf{fut}(u \mid v) \\ & \mathbf{ob}(o \mid \mathbf{if}\ f?\ \mathbf{then}\ s1\ \mathbf{else}\ s2\ \mathbf{fi}; s, a, l, q) \\ \longrightarrow & \mathbf{fut}(u \mid v) \\ & \mathbf{ob}(o \mid s1; s, a, l, q) \\ \mathbf{if} & [f]_{a+l} = u \end{array}$$

polling2 :
$$\begin{array}{ll} & \mathbf{fut}(u \mid) \\ & \mathbf{ob}(o \mid \mathbf{if}\ f?\ \mathbf{then}\ s1\ \mathbf{else}\ s2\ \mathbf{fi}; s, a, l, q) \\ \longrightarrow & \mathbf{fut}(u \mid) \\ & \mathbf{ob}(o \mid s2; s, a, l, q) \\ \mathbf{if} & [f]_{a+l} = u \end{array}$$

Figure II.13: Operational rules for first-class futures.

denote a resolved future object with value $v$. In this paradigm, a future is once writable, but readable many times by objects that have a reference to it. Objects can individually and synchronously access the future object value when resolved. Some of the rules from Figure II.12 must then be modified, as shown in Figure II.13 with primed versions of the relevant rules. The rules invocation, start, and tail are as before and are therefore not redefined.

The async-call' rule represents the creation of a unique future object corresponding to a method call. It creates an unresolved future object with a unique identity (details omitted). This rule creates an invocation message to the callee as before. However, the caller identity ($o$) is not needed in the case of first-class futures unless the language supports the implicit caller parameter, and one could remove $o$ from the invocation message in this case (simplifying both the async-call' and the invocation rule).

The reply' rule says that when object $o$ returns the value of $e$, it becomes **idle** and the reply goes to the future object with identity $u$. The get', await-pq', and suspend' rules capture the cases when there is a query statement to get the result as before, but now using the future object. In the get' rule, the query succeeds since the reply value is resolved in the future object **fut**$(u \mid v)$, hence if the evaluation of $[f]_{a+l}$ according to the local variables and object fields equals $u$, then the object can update $x$. For the await-pq' rule, this query is in the internal queue and the active process is **idle**, and then the object deals with this suspended query. If the reply value is resolved, the object can update $x$. For the suspend' rule, an **await** statement is in the internal queue and the reply is not resolved yet, therefore the whole process with its corresponding local variables is suspended in the internal queue.

The semantics of polling is similar to that of get/get' for the case that the future is resolved, in which case the expression $f?$ is replaced by *true*, and suspend/suspend' for the case that the future is not resolved, in which case the expression $f?$ is replaced by *false*.

For languages without cooperative scheduling we omit $pq$ and omit the suspend' and await-pq' rules, and the optional **await** in get'. This means that we have defined the operational semantics of all the chosen classes of languages. For simplicity we have not considered aspects such as garbage collection of futures.

## II.4 Evaluation

We evaluate the various mechanisms for interaction between active objects with respect to the dimensions stated in the introduction. As mentioned we consider imperative languages with or without futures but not implicit futures.

### II.4.1 Syntactic complexity

We compare the different versions of the unified syntax in Figure III.1, including related static checking aspects and also immediate pragmatic issues. It is obvious that explicit futures require programming awareness of call identities or futures,

representing a new kind of entity, and new programming choices like how and when to use them. Thus the notion of future variables comes with a syntactic and pragmatic cost, especially since the addition of future variables does not reduce the need of ordinary variables.

When a new method is declared in a language with first-class futures, one must decide for each parameter and return value if it should be represented by a future or not. Moreover, the passing of futures in a typed language requires typing support of future types, as in $Fut[T]$. And if $T'$ is a subtype of $T$, one may want $Fut[T']$ to be a subtype of $Fut[T]$. This means that first-class futures and object-local futures give a more complex type system, whereas method-local futures can be handled with a simple predefined type, say $Fut$. Pragmatically, the resolving of futures of futures may lead to some confusion.

For local futures, we have seen in subsection II.3.2.1 that it is possible to ensure that futures really are object-local or method-local, as appropriate, by means of simple language restrictions. First-class future operations reuse basic language mechanisms and do not add further to the syntactic restrictions of a language. For all languages with futures, one may add static restrictions to ensure that a get operation is not done on a *nil* future (one that has not been associated with a call yet), a problem which is similar to avoiding nil references. For first-class or object-local futures, one may for instance insist that a future can only be passed when statically not *nil*. For method-local futures, the situation is much simpler since there is no passing of futures. However, static restrictions are needed to ensure that a get statement $x :=$ **get** $f$ is type-correct when $x$ is simply typed by the predefined type $Fut$ [41].

Future variables give a level of indirectness in that the retrieval of the result of a call is no longer syntactically connected to the call, compared to future-free languages. The connection might range from trivial (as in $f := o.m(...); x :=$ **get** $f$) to non-trivial, for instance when the future is received as a parameter. In the latter case a get statement in a given method may not statically correspond to a unique call statement. This is a complicating factor in static analysis, especially modular static analysis. One may overestimate the set of call statements that correspond to a given get statement, but this requires access to the whole program, which is not possible in open-ended software systems.

From a pragmatic point of view, we notice that in languages with first-class futures, the passing of futures is decided when a method is declared rather than when it is called, i.e., for each method parameter or return value one must decide at declaration time if it should be represented by a future or not, and all calls made later must obey this regime. Then, too few parameters given as futures may imply that desired passing of futures is not supported, and too many future parameters lead to dummy future variables on the call-site and syntactic overhead. This can be a problem in languages with first-class futures since the need of the future mechanism depends on the calling objects. Thus first-class future languages may not provide the desired flexibility. This is not a problem for method-local futures since there is no passing of futures. And the problem is rather limited for object-local futures since future passing is limited to local

methods, for which all calls appear in the class definition (but class inheritance may reintroduce the flexibility problem).

Consider next cooperative scheduling. The **await** mechanism can be decided for each call upon need, and is not pre-decided in method declarations. This means that the same method may be called by means of blocking calls as well as by suspending calls. This gives a high degree of flexibility. The addition of cooperative scheduling is syntactically cheap since essentially only one keyword (**await**) is needed, and type analysis is not affected. Pragmatically, the use of suspension should reflect the need to wait for something without blocking, and the choice between blocking and non-blocking waiting should be obvious for a conscious programmer. However, the combination of first-class futures and cooperative scheduling gives two different mechanisms that may be used to deal with waiting without blocking, and the choice between them is less obvious.

Finally we consider the concept of polling. Adding polling is syntactically cheap, essentially a new predefined function ("f?") is added. Type checking issues are trivial, but some language restrictions may be added to control where polling is allowed, for instance restricting it to if-conditions (possibly allowing Boolean expressions involving several polling checks). Pragmatically, the use of polling gives increased programmer control. For instance, one may wait for two results and react to them in the order they appear. This may look like

```
if f1? then react1; get f2; react2 else if f2? then react2; get f1; react1 fi fi.
```

However, when there are more than two futures that should be handled individually, the branching becomes rather involved and with repetition of parts of the code. Use of futures or cooperative scheduling may provide more elegant solutions: With cooperative scheduling one may start one process for each future, and let each process await the corresponding future. With first-class futures one may pass each future to a new object taking care of the proper reaction.

All in all, we have considered syntactic complexity and immediate pragmatic issues. We have seen that futures add more syntactic complexity and more pragmatic complications than cooperative scheduling and polling. And first-class futures add more pragmatic complications than object-local futures, which again add more pragmatic complications than method-local futures. Polling without cooperative scheduling may lead to a complex programming style, whereas languages with first-class futures or cooperative scheduling are less depending on polling.

### II.4.2 Semantic complexity

The operational semantics (Figures II.12 and II.13), although abstractly defined, shows that local futures and future-free languages can be handled by reply queues (or sets) on the caller side, whereas first-class futures is handled by shared future objects. For method-local futures, the replies can be removed when the method instance (process) that generated the future has terminated. This can be formalized in the semantics by including the identity of the calling

process (given by *callid*) in the future identity, for instance as a pair $(i, j)$ where $i$ is the process identity and $j$ the future identity relative to $i$. We may then add a rule

$$\mathbf{rq}(o \mid r \cdot ((i,j),v) \cdot r')\ \mathbf{ob}(o \mid s,a,l,q) \longrightarrow \mathbf{rq}(o \mid r \cdot r')\ \mathbf{ob}(o \mid s,a,l,q)\ \ \mathbf{if}\ \ i \notin q$$

where $i \notin q$ checks whether a process with callid equal to $i$ is in the process queue $q$. In the case of single-read futures as well as future-free languages, a reply in $rq$ may even be removed as soon as it is read, in the rules get and await-pq (by removing the reply message from $r$ in the right-hand-side). This still requires that removal is handled upon method termination, as for multiple-read local futures, in case the future is not read. (Alternatively this could be controlled by commands inserted in the code during static checking.) Thus for method-local futures (both single-read and multiple-read), the reply messages can be removed efficiently without general garbage collection. The same scheme can be used for future-free languages.

First-class futures involve the creation of future objects. The placement of the futures in a distributed system is not specified by the abstract operational semantics (other than being placed somewhere in the system configuration), nor is the removal of these objects, but could be added by rules that consider the total system (letting futures that are no longer referred to in the system be deleted). A more efficient implementation is possible, as discussed in subsection II.4.4.

The notion of cooperative scheduling adds an internal process queue to each active object, together with en-queuing and de-queuing operations. The semantics of basic statements (other than call, return, get, await, polling) is not affected by futures nor cooperative scheduling. For the considered mechanisms, first-class futures make the most complex change of the basic operational semantics, since this mechanism changes the notion of system configuration and necessitates garbage collection of futures, while that of cooperative scheduling is less involved, and that of polling is fairly trivial.

### II.4.3 Expressiveness

We first show that the future mechanism can be encoded in an active object language with future-free constructs, using the asynchronous call/return paradigm together with dynamic object creation. Given a method $Vm(\overline{\mathsf{T}\ \mathsf{x}})$ declared in an interface $I$ (where $V$ is the type of the return value), we define an interface for futures for this call by

```
1 interface Future_m {
2     Bool resolve()          // waiting until resolved
3     V get()                 // gets the value when resolved
4 }
```

and an extension of this interface for the case that we allow *polling*:

```
1 interface PFuture_m extends Future_m{
2     Bool resolved()         // polling
3 }
```

```
1 class PFUTURE_m(I o, T̄ x̄) implements PFuture_m {
2     Bool res:= false; // is the future resolved?
3     V val; // the value of the future when resolved
4     // initial code
5     {await val:=o.m(x̄); res:=true}// non−blocking
6     Bool resolved(){return res}
7     Bool resolve(){await res; return true}
8     V get(){await res; return val}
9 }
```

```
1 class FUTURE_m(I o, T̄ x̄) implements Future_m {
2     Bool res := false; // is the future resolved?
3     V val; // the value of the future when resolved
4     // initial code
5     {val:=o.m(x̄); res := true} // blocking
6     Bool resolve(){return true} // await res is implicit
7     V get(){return val} // await res is implicit
8 }
```

Figure II.14: Implementation of simulated futures with and without polling.

These interfaces are implemented by two classes, FUTURE_m and PFUTURE_m respectively, given in Figure II.14. A new future then corresponds to a new future object, using the appropriate class. Thus, if $f$ is an object of interface Future_m, the creation of a future by

$$f := \textbf{new}\ \ FUTURE\_m(o, e)$$

corresponds to the call $f := o!m(e)$ in a language with futures. The call $f.resolve()$ corresponds to blocking while waiting for a result, and the call $x := f.get()$ corresponds to $x := \textbf{get}\ f$ in a language with futures. For $f$ of interface PFuture_m, the call $f.resolved()$ corresponds to the test $f?$. Furthermore, first-class operations on $f$ correspond to first-class future operations. In case of cooperative scheduling, the call $\textbf{await}\ x := f.get()$ corresponds to $\textbf{await}\ x := \textbf{get}\ f$ in a language with futures, and the call $\textbf{await}\ f.resolve()$ corresponds to $\textbf{await}\ f?$ in a language with futures.

Consider the implementation of futures with polling given in Figure II.14 by class PFUTURE_m (for a given method $m$ as above). The call to $m$ in the class implementation uses **await** so that the future object will be able to perform incoming calls before the future is resolved, and in this way be able to return the appropriate result of polling requests. Thus, implementation of polling requires suspension (or could be predefined outside the language).

For the case without polling we may implement the future mechanism without use of suspension, by class FUTURE_m in Figure II.14. Here we can use a blocking call to $m$ since nothing can be done by the future object when not resolved. Therefore we do not need **await** $res$ in the implementation since the object cannot respond to any request when the future is not resolved. This could lead to more efficient scheduling. Since this implementation requires only the blocking

call construct, it can be expressed in all considered languages. Thus built-in futures are not strictly needed in languages without futures since they can be expressed (simulated) within these languages, but polling requires suspension (or similar mechanisms). In either case, the implementation of a future uses one object with its own thread. This gives a simple high-level model of the future mechanism. It may seem like an inefficient solution, but due to the passiveness of the future objects, it suffices that they are given CPU time only when there is an incoming call/reply from the environment.

Each call involving a future is implemented as an object of class FUTURE (or PFUTURE). The future object makes the call and (eventually) receives the return value, as well as dealing with requests about the future value. We support polling of future values as well as blocking and non-blocking waiting primitives for requesting the future value.

On one hand it is obvious that the addition of language features may increase the expressive power. With our unified syntax the future-free language is a subset of that for method-local futures, which is a subset of that for object-local futures, which again is a subset of that for first-class futures. Their expressiveness is accordingly. And it is clear that the addition of cooperative scheduling adds expressive power. On the other hand we have seen that first-class futures can be expressed in languages without first-class futures or without futures, by means of object generation using predefined classes. Thus if shared futures are occasionally desirable, they can be defined by means of explicit future objects definable within the language. We have also seen that simulation of polling requires the await mechanism (or similar).

### II.4.4 Efficiency

In this part we compare the efficiency of active object languages, considering the amount of message exchange and complexity of the garbage collection process. For example, in a distributed system with wireless IoT devices, the amount of network traffic is vital, and should be taken into account. Since IoT devices suffer from resource and power limitations, communication efficiency, as well as space and time limitations, matter. Correspondingly, a high amount of message passing and cumbersome garbage collection cause high resource and power consumption, which in general should be avoided.

In active object languages, the future paradigm and the interaction mechanisms influence the number of communication messages and the garbage collection. Consider a method result that is needed by several objects in addition to the caller. For languages supporting the asynchronous call/return paradigm, the network traffic consists of two messages (i.e., from caller to callee and back). Assuming $n$ (other) objects need the value and that the caller can reach them indirectly, it then takes at least $n$ messages to forward this value to them, say $n'$ ($n' \geq n$). Thus the number of exchanged messages is $2 + n'$. We assume that the forwarding is done by parameter passing, say through void methods with no return message, and we count the number of messages needed for all $n$ objects to

receive the value (including the forwarding), not counting other related messages. Moreover, by using suspension one can avoid blocking the caller.

In the case of first-class futures, the number of exchanged messages depends on the update strategy, as mentioned in the ASP part (in subsection II.2.6). We here assume that the future object is kept on the caller side. In the *eager-forward-based strategy*, the caller forwards the future reference to $n$ objects and forwards the future value when resolved. With the assumptions above, the number of exchanged messages would be $2 + 2\,n'$. Considering the *eager-message-based strategy* or subscription scheme, the caller sends the future reference to $n$ objects, each object receiving the future subscribes itself to the future, and then the future value is sent back to all of them when resolved. Hence the number of exchanged messages would be $2 + 3\,n'$, assuming the $n'$ messages go to distinct objects. In the case of *lazy strategy* with blocking when needed, the caller sends the future reference to $n$ objects, and when the value is required, they ask for it from the future object, which sends back the value. Hence the number of messages would be $2 + n' + 2\ n$, assuming all $n$ objects really need the value.

We see that the number of messages gets higher with the future mechanism. In a distributed system this could cause problems. And in particular for IoT systems, this could lead to exhaustion of network capacity or battery time if $n$ is greater than 1 for a large portion of the calls.

Consider next *garbage collection*. The purpose of garbage collection is to deallocate memory resources that are no longer in use at runtime. There are different kinds of strategies for garbage collection. One is *reference counting*, in which the number of references to an object (say a future object) is counted. An object can be deallocated when the number of references to it reaches zero. A disadvantage of reference counting is that for every object a resource must be reserved for storing the number of references to it, and this number must continuously be updated.

Another common way of garbage collection is *reference tracing*, i.e., following all references. It distinguishes between reachable and unreachable objects, and deallocates the latter ones. An object is reachable if it is referenced by a variable in an (potentially) active object directly or through a chain of references. A traditional reference tracing collector temporarily stops the program execution when it wants to deallocate unreachable objects. This guarantees that reachablility does not change while doing garbage collection. Halting the program execution can be undesirable in distributed systems with active entities. In these systems, an object might be used by several distributed units, necessitating distributed garbage collection which is complex, slow, and costly.

The given operational semantics does not include garbage collection. Futures are typically many and short-lived, which may cause much garbage. In contrast, the active objects are long-lived. Thus in languages without passive objects, there is then little or no need for garbage collection other than the future objects. Therefore the garbage collection of futures can be a problematic issue. For the given operational semantics, a straightforward implementation of garbage collection for first-class futures requires distributed garbage collection of future objects, since the futures are global runtime objects, whereas object-local futures

require local garbage collection within each object, since the futures are stored and referenced locally in each object.

As already explained in subsection II.4.2, method-local futures can be disposed without use of garbage collection, and the reply messages used in future-free languages can be removed efficiently. And the same implementation can be used for single-read futures.

For languages with first-class futures, the garbage collection mechanism for futures highly depends on the future update strategy, as already discussed in subsection II.2.6. If a strategy is eager, a future is updated as soon as its value is computed, then a local garbage collection is enough since it does not need to be stored globally. Local garbage collection can be performed by classical garbage collection within an object or by combining different garbage collection techniques with static analysis approaches [13]. If an update strategy is lazy, a future value must be kept in a future value list in the callee for potential later requests; moreover, a future can be disseminated to many active objects, thus it is non-trivial to decide when a future value can be removed from a future value list. In this case, distributed garbage collection is required, which can be done by reference counting or a combination of garbage collection mechanisms [13].

### II.4.5 Security/Privacy challenges

The future concept comes with a notion of future identity, but not a notion of associated caller, callee, or creator. However, if the identity of the caller of the associated method call is incorporated in the future identity (as indicated in the operational semantics), it could in principle be possible to extract this at runtime. But the object creating the value would still in general be unknown. For instance, in the case of first-class futures, a caller may create a future corresponding to a method call on $o$ and pass the future reference as a parameter to other objects. When the future is resolved, they obtain its value from the future object without knowing who has created this value. In fact, for a future received as a parameter there is in general no available static or dynamic information about the creator. It is not known where the future value comes from, who has generated it, or how fresh it is. This opens up for third party information with indirect/implicit handling of private information. An object may implicitly reveal futures with private information. However, dynamic information could be provided by the addition of language attributes. Thus with some overhead dynamic checking would be possible.

Static information flow analysis of the secrecy level of first-class futures is therefore imprecise. In order to have secure information flow for first-class futures, dynamic checking is required, which is expensive compared to static analysis. Class-wise information flow analysis has been suggested for a future-free version of Creol [60]. The approach is based on static declaration of secrecy levels for each input parameter and result value of a method. This would be difficult when allowing futures as parameters, because the set of external calls that may result in an actual parameter is not statically given, and because the calls in this set are not uniform with respect to secrecy levels. Static declaration of secrecy levels

must consider the worst case possibility (i.e., the highest secrecy level), and this will easily lead to inflation of secrecy levels, something which is not desirable since then it would severely limit statically acceptable information passing and call-based interaction, or require dynamic checking.

In a paper by Attali et al. [2], secure information flow for the ASP language is provided by dynamically checking for unauthorized information flows. In their approach security levels are assigned to activities and transmitted data between these activities. For verification, dynamic checks are implemented at activity creation, requests, and replies. However, future references can be freely transmitted between activities because they do not hold any valuable information, they just hold addresses. But for updating a future and getting its value, the secrecy level of this transmission will be performed dynamically by security rules of a secure reply transmission.

Therefore static information flow checking of first-class futures is problematic, which means that it must be compensated by dynamic checking. In this sense, first-class futures do not promote security at the programming level.

### II.4.6 Program reasoning

We compare the simplicity of rules for reasoning about method calls for future-free and future-based languages, and the usefulness of these rules in practice, considering two typical kinds of applications. The rules given below cover future-free and future-based languages, with or without suspension and the tail construct.

In the setting of active objects, compositional reasoning is typically based on the notion of communication traces, often called *histories* [79]. The specification of an object can then be given by specifying invariants by means of predicates over the local history, i.e., the history of events generated or observed by that object. Modular reasoning about classes can be based on *class invariants* referring to the fields of the class and the local history $h$. The class invariant considers a given object (this) and should hold whenever the object is idle, but may be violated when not idle [25]. In addition, one may use pre- and post-conditions to specify any additional properties of the methods in the class. The Hoare triple $\{P\} : s : \{Q\}$ specifies that if the program $s$ is started in a state satisfying the precondition $P$ then the final state will satisfy the postcondition $Q$, provided the program terminates, thereby expressing partial correctness [37]. For instance, an assignment statement $x := e$ satisfies the triple $\{Q^x_e\} : x := e : \{Q\}$, where the notation $Q^x_e$ denotes textual substitution, replacing the expression $e$ for all (free) occurrences of the variable $x$ in $Q$. This assignment axiom holds in our language setting since there is no remote field access. This rule is left-constructive, expressing the weakest precondition. A class invariant $I$ must hold after class initialization and be maintained by each method of the class, as well as between any suspension points. This allows us to rely on the invariant when a new method is started or re-started after suspension. The triple $\{I\} : s : \{I\}$ expresses that the statement list $s$ maintains the invariant $I$.

simple call $\quad \{Q^{h}_{h\cdot(this \to o.m(\overline{e}))}\} : o!m(\overline{e}) : \{Q\}$

sync-call $\quad \{o \neq \mathsf{this} \wedge \forall x' \,.\, Q^{x,h}_{x',h\cdot(this \to o.m(\overline{e}))\cdot(\mathsf{this} \leftarrow o.m(\overline{e};x'))}\} : x := o.m(\overline{e}) : \{Q\}$

tail
$$\frac{\{o' = o \wedge \overline{e}' = \overline{e} \wedge P\} : s : \{\forall x \,.\, Q^{h}_{h\cdot(\mathsf{this} \leftarrow o'.m(\overline{e}';x))}\}}{\{o \neq \mathsf{this} \wedge P^{h}_{h\cdot(this \to o.m(\overline{e}))}\} : x := o.m(\overline{e}) < s > : \{Q\}}$$

await $\quad \{L \wedge I \wedge h = h'\} : \mathbf{await}\ e : \{L \wedge I \wedge h' \leq h \wedge e\}$

Figure II.15: Reasoning rules for call-related statements for future-free languages. Here $L$ is a local condition (not referring to fields).

**Notation** For histories, we let $h/S$ denote projection by means of a set $S$, i.e., the sub-sequence of all the events in the set $S$. And we let $h \cdot z$ denote the history $h$ appended with the event $z$. Furthermore, $\leq$ denotes the sequence prefix relation, i.e., $h' \leq h$ expresses that $h'$ is an initial part (prefix) of $h$, and $\subseteq$ denotes the subsequence relation, i.e., $h' \subseteq h$ expresses that $h'$ is a subsequence of $h$.

The local history $h$ of an object $o$ is related to the global history, $H$, by the equation $h = H/\alpha(o)$ where $\alpha(o)$ is the alphabet of $o$ (the set of events generated/observed by $o$), i.e., $h$ is the projection of $H$ by the events visible to $o$. Compositional reasoning is then possible by conjunction of the local invariants, replacing the local history $h$ by $H/\alpha(o)$, together with a "compatibility" assertion stating the partial order between the events corresponding to meaningful execution order (i.e., an event reflecting a method invocation must come before the event reflecting the corresponding method completion), following the compositional principle of [79].

### II.4.6.1 Rules for future-free calls

Reasoning rules for the future-free call constructs, including call with and without tail, are given in Figure II.15, based on [58]. In this setting we consider two kinds of events: *call events* and *return events*. (Events reflecting the start and end of a method execution may be considered as well, but we here focus on the treatment of calls.) The execution of the asynchronous call $o!m(\overline{e})$ by *this* object is represented by the call event $this \to o.m(\overline{e})$, and the observation by *this* object of the return value $v$ generated by object $o$ resulting from this call is represented by the return event $this \leftarrow o.m(\overline{e}; v)$. Thus, call statements have side-effects on the local history, a simple call appends the local history by a call event, and a synchronous call (with or without a tail) appends the local history by a call event as well as a return event. In the case of a synchronous call with a tail, the call and return events are added to the history before and after the tail, respectively.

Rule simple call expresses that the execution of the simple asynchronous call $o!m(\overline{e})$ has the effect of appending the corresponding call event to the local history. Rule sync-call expresses that a synchronous call $x := o.m(\overline{e})$ has the effect of appending the history with both the call event and the corresponding return event. The rules involving return events use universal quantification to reflect that the return values are under-specified in local reasoning, and primed variables are used to freeze pre-values of program variables that may change (when needed in a postcondition). Since self calls have a different semantics than remote synchronous calls, we add the restriction $o \neq \mathsf{this}$ in the precondition of a synchronous call to $o$ with or without tail. The call rules are all left-constructive. In Rule tail, the return event is added when transforming the overall postcondition $Q$ to a postcondition of the tail, and the call event is added when transforming the precondition of the tail to an overall precondition. The reasoning rules can be understood by letting the call to $o$ have the side-effect

$$h := h \cdot (\mathsf{this} \rightarrow o.m(\overline{e}))$$

on the history, where $\overline{e}$ are the actual parameters, and letting the observation of a return have the side-effect

$$h := h \cdot (\mathsf{this} \leftarrow o.m(\overline{e}; v))$$

where $v$ is the return value. More precisely, reasoning about the statement $[\mathbf{await}]\ x := o.m(\overline{e}) < s >$ is the same as reasoning about the sequence

$$h := h \cdot (\mathsf{this} \rightarrow o.m(\overline{e})); s; [\mathbf{await}\ \ true;]\ x := \mathbf{some}; h := h \cdot (\mathsf{this} \leftarrow o.m(\overline{e}; x))$$

where **some** is a locally unknown value such that $\{\forall x\,.\,Q\} : x := \mathbf{some} : \{Q\}$. The tail rule (without **await**) can be derived from this understanding. In fact, all call rules can be derived from this understanding when ignoring any optional parts (await/tail $s$) not used. And for languages supporting suspension, rules for await-call with or without tail can be derived as well, using the await rule.

The await rule says that the class invariant $I$ will be maintained over suspension, even though fields may change. The history may only grow, as formalized by the postcondition $h' \leq h$ where $h'$ is the history in the prestate. And since the local state is unchanged during suspension, any local condition $L$, not referring to fields, is also preserved. In addition the **await** condition $e$ is guaranteed immediately after suspension. In the same way as a conditional await, an await-call statement must ensure the invariant at the beginning of suspension, and may assume the invariant immediately after suspension.

### II.4.6.2 Rules for calls with futures

For languages with futures, the asynchronous call invocation is textually separated from the result query (the get statement). Thus in the analysis of a get statement, neither the method name ($m$) nor the callee are in general known at verification time, not even in the case of local futures. Therefore neither

async-call $\quad \{\forall f' \,.\, Q^{f,h}_{f',h\cdot(\mathsf{this}\rightarrow o.m(f',\overline{e}))}\} : f := o!m(\overline{e}) : \{Q\}$

get $\quad \{\forall x' \,.\, Q^{x,h}_{x',h\cdot(\mathsf{this}\leftarrow(e,x'))}\} : x := \mathbf{get}\ e : \{Q\}$

Figure II.16: Hoare style rules for futures.

of these can be part of the event reflecting the completion of a get statement, in contrast to the case of future-free languages. Instead, the future identity must be included in the call and get events: We now let a call generate the call event $\mathsf{this} \rightarrow o.m(u, \overline{e})$ where $u$ is the future identity. And we let the *get event* $o \leftarrow (u, v)$ correspond to the observation by $o$ of the value $v$ of future $u$.

The reasoning rules for calls with futures are shown in Figure II.16, based on [25]. Rule async-call is similar to Rule simple call, but the generated future identity is under-specified, which explains the quantifier. Rule get expresses that the history is appended with the corresponding get event, with an under-specified future value. Thus this setting gives rise to events where the connection between call and get events is made by means of the future identity, rather than the method name. The rules for futures are therefore more indirect and complex to apply in practice, as discussed next in Subsection II.4.6.3.

### II.4.6.3 Application of the rules

We will consider two typical cases of program reasoning, namely reasoning about method calls by means of input/output relations, and reasoning about the ordering of method calls, exemplified by verification of sequential ordering.

**Reasoning about input/output relations** Reasoning about a method call often deals with the relationship between the input and output values of a method call. For a method body $s; \mathbf{return}\ e$ with parameters $\overline{x}$ and return value $e$, one may specify and verify such a relationship, say $R(\overline{x}, e)$, by verifying the Hoare triple $\{true\} : s : \{R(\overline{x}, e)\}$ by ordinary Hoare analysis. However, for external calls with futures, it is not straight forward to use such facts, since the input and output to a call may not in general be known in a single state of the caller, due to the decoupling of the get statement form the invocation statement. But one may use the history. Consider the code $f := o!m(\overline{e}); ...; y := \mathbf{get}\ f$ where $o$ is an external object (different from this). This gives rise to the call event $(\mathsf{this} \rightarrow o.m(u, \overline{e}))$ where $u$ is the future identity assigned to the future variable $f$, and the query gives rise to the get event $(\mathsf{this} \leftarrow (u, y))$ for some return value $y$. We may then state

$$(\exists u \,.\, (\mathsf{this} \rightarrow o.m(u, \overline{x})); (\mathsf{this} \leftarrow (u, y)) \subseteq h) \Rightarrow R(\overline{x}, y)$$

(for any values of $\overline{x}, y$) expressing that $R$ holds for the input and output values (found in separate call and get events in $h$) for the same $u$. Therefore reasoning from this fact involves quantifiers.

In contrast, for a synchronous call [**await**] $y := o.m(\overline{e})$, where $o$ is different from this, we may use the fact

$$(\mathsf{this} \leftarrow o.m(\overline{x}; y)) \in h \Rightarrow R(\overline{x}, y)$$

(for any values of $\overline{x}, y$), and we obtain $R(\overline{e}, y)$ in the post-state of the call since $(\mathsf{this} \leftarrow o.m(\overline{e}; y)) \in h$ obviously holds in the post-state of the call (for $y$ not occurring in $\overline{e}$). Such return events appear for synchronous calls with or without a tail as well as for suspending calls with or without a tail. Thus reasoning about such calls from the history can be done in the same manner, without quantifiers.

Finally, we may look at simulated futures. In this case, events are generated from the creation of the future object $o'$ and from the interaction with that object, typically through the *get* method. This gives one creation event, one call event, and one return event, of which the first (the creation event) and last are the ones needed to express an input/output relation $R$. Similarly to the case of first-class futures, the identity of the future object is arbitrary. This means that for a method $m$ (with body as above, satisfying $R$), called by simulated futures, we may state

$$(\exists o' \,.\, (\mathsf{this} \rightarrow o'.new\ FUTURE_m(o, \overline{x})); (\mathsf{this} \leftarrow o'.get(; y)) \subseteq h) \Rightarrow R(\overline{x}, y)$$

where $\mathsf{this} \rightarrow o'.new\ C(\overline{x})$ denotes the creation event of an object $o'$ of class $C$ with class parameters $\overline{x}$. Thus the situation is quite similar to the case of futures and involves quantifiers. Hoare-style reasoning about method calls by means of input/output relations is therefore substantially more complicated for languages with futures than for future-free languages. And reasoning about simulated futures is comparable to reasoning with futures.

**Reasoning about ordering of calls** A history-based invariant may for instance state that the remote $m$ calls to $o$ made by the current object are sequential, in the sense that a new $m$ call can only be made when the results of the previous $m$ calls to the same object $o$ have been observed by the current object. To express this in the setting of future-free languages, we may write the class invariant

$$\forall o \,.\, \#(h/\{\mathsf{this} \rightarrow o.m\})\ \ \mathbf{follows}\ \ \#(h/\{\mathsf{this} \leftarrow o.m\})$$

where **follows** denotes sequential connection as defined by $x\ \mathbf{follows}\ y \triangleq (x = y \vee x = y + 1)$, and $\#$ denotes sequence length, and $\{\mathsf{this} \rightarrow o.m\}$ denotes the set of call events to $o$ from this object for method $m$. (Alternatively, one could use equality since the invariant is only required to hold when the object is *idle*.)

In the case of languages for futures, the above invariant will be more complicated since the method name $m$ is not visible in the event observed at a **get** statement. In this case the class invariant can be expressed by

$$\forall o \,.\, \#(h/\{\mathsf{this} \rightarrow o.m\})\ \ \mathbf{follows}\ \ \#(h/\{\mathsf{this} \leftarrow (u, \_) \mid u \in \mathit{futures}(h, o, m)\})$$

letting $\mathit{futures}(h, o, m)$ be defined as the set of futures generated by a call event of form $\mathsf{this} \rightarrow o.m$, and letting $\{\mathsf{this} \leftarrow (u, \_) \mid c\}$ be the set of all get events to

this object with future identity $u$ such that condition $c$ is satisfied. Clearly, it is significantly harder to formulate the invariant in this case, and also reasoning becomes more complicated. Proving maintenance of the invariant by a call $y := o.m(\overline{e})$ is straight forward in the case of future-free languages. For instance

$$\{I\} : y := o.m(\overline{e}) : \{I\}$$

reduces to $I \rightarrow \forall y' \,.\, I^h_{h \cdot (this \rightarrow o.m(\overline{e})) \cdot (\mathsf{this} \leftarrow o.m(\overline{e};y'))}$, which is trivial for our invariant since $m$ is explicit in the events. In the case of languages with futures, the reasoning becomes non-trivial, since the $m$ is no longer explicit in the get event. One must reason indirectly through future identities across events, possibly also generated by different processes (on this object). This involves the $u$ implicitly quantified through the conditional set expression.

For simulated futures we may use the invariant

$$\forall o \,.\, \#(h/\{\mathsf{this} \rightarrow \_.new\ FUTURE_m(o,\_)\})\ \ \textbf{follows}\ \ \#(h/\{\mathsf{this} \leftarrow o'.get \mid o' \in futures(h,o,m)\})$$

with *futures* defined almost as above. Reasoning in this case is similar to the case of languages with futures.

We may conclude that specifications and invariants are more indirect in languages with futures than in future-free ones. In particular it is harder to express relationships between input parameters and corresponding result values. Verification conditions get more complex to prove when they depend on get events, as demonstrated in the examples. We have also seen that the reasoning rules for local futures and first-class futures mainly have the same complexity. Furthermore reasoning about simulated futures is comparable to reasoning about futures. Thus reasoning with a future-free language is significantly simpler than reasoning about future-based calls and simulated futures.

## II.5 Conclusion

Programming paradigms are essential in software development, especially for distributed systems since these affect large programming communities and a varied range of applications. Thus, investigation and comparison of different programming paradigms are valuable. We have focused on interaction paradigms for imperative programming languages based on the active object model, ignoring implicit futures. This model has gained popularity due to its modular semantics and natural support of concurrency and autonomy. Most of the recent active object languages adopt the future mechanism, rather than a two-way interaction paradigm. First-class futures give a possibility of sharing information and of partially avoiding blocking. An active object that has generated a call with a future may pass the future to a number of clients, and as long as it does not need the future value itself, it can continue to serve clients without being blocked. Waiting is then delegated to those clients that need the future value. This gives a programming style that avoids deadlocking and blocking in server objects, allowing the servers to be continuously responsive to client requests.

| criteria | FF+CS | FF | LF+P | LF+CS | NF+CS | NF |
|---|---|---|---|---|---|---|
| expressiveness | + | 0 | 0/0 | 0 | 0 | − |
| efficiency | − | − | −/0 | + | 0 | − |
| synt./sem. complexity | − | − | 0/0 | 0 | + | + |
| security aspects | − | − | 0/+ | + | + | + |
| static analysis | − | − | −/0 | 0 | + | + |
| program reasoning | − | − | −/− | − | + | + |

Figure II.17: A simplified summary of the evaluation of the different paradigms. The case of local futures with polling (LF+P) is split in two subcases, object-local and method-local futures.

Cooperative scheduling for active objects is another recent mechanism that avoids blocking and allows passive waiting. Without use of futures, a server object can wait for the future value in a suspended (sleeping) process, avoiding blocking, and thereby be continuously responsive to client requests, even in the case that the server itself needs the value. In this case the clients will get the future value when it is available (when the suspended process is enabled), and thus waiting is also avoided in the clients as well. This means that there is less need for futures in languages with cooperative scheduling than in those without, even though cooperative scheduling does not provide sharing. This means that languages with first-class futures or cooperative scheduling more directly support propagation of method results without waiting, and are in this respect more expressive than those without neither of these two mechanisms.

Polling has the advantage of more fine-grained synchronization control. For instance one may await the results of a number of outstanding calls in a given order. However, polling may lead to more complex program structure. For instance to cover any ordering of the completions of $n$ calls, one could end up with $n!$ branches. In contrast, with cooperative scheduling there could be one suspended process for each of the outstanding calls. With first-class futures one could delegate to $n$ other objects (by passing futures), so that each waits for one completion. Thus there is less need for polling in languages with cooperative scheduling or first-class futures.

Object-local futures offer more flexibility than method-local futures, but in the presence of cooperative scheduling one may use suspension to compensate. In particular single-read method-local futures give several advantages in the case of cooperative scheduling.

This leaves seven language paradigms for interaction as the most interesting: i) first-class futures and cooperative scheduling (FF+CS), ii) first-class futures without cooperative scheduling (FF), iii) method-local futures and cooperative scheduling (LF+CS), iv and v) local futures with polling (LF+P), with subcases for object-local and method-local futures, vi) no futures and cooperative scheduling (NF+CS), and vii) no futures (NF). We have focused on these interaction paradigms and evaluated them along the chosen criteria. For a rough overview, some main points of the evaluation results are illustrated in

Figure II.17. Here + is better than 0, which is better than −.

With respect to expressiveness, we have seen that first-class futures can be expressed by means of explicit object generation (using predefined classes). Thus the need for built-in first-class futures is not so critical. Even though simulated futures are less flexible (at least syntactically) than built-in futures, they have the advantage that the cheaper (implementation-wise) alternatives are available by default when first-class futures are not strictly needed. Cooperative scheduling gives an expressiveness that cannot be simulated with (first-class) futures without use of dynamic object generation and blocking. Languages with built-in first-class and cooperative scheduling futures have the highest expressiveness (marked as "+" in Figure II.17) whereas the ones without (only simulated ones) have somewhat less expressiveness. Future-free languages are less expressive than those with futures, but the addition of the tail construct results in similar expressiveness as languages with local futures. Polling allows non-blocking programming (while compromising program structure), and increases expressiveness and efficiency in the setting of local futures without cooperative scheduling.

We have seen that first-class futures require garbage collection, which is non-trivial in the case of distributed systems. And we have seen that first-class futures give raise to more messages than languages without. There is no uniformly best choice of update strategy for first-class futures [35]. Different implementation strategies may give more efficient garbage collection, but at the cost of more internal messaging. For IoT systems this could be critical. In contrast, cooperative scheduling adds to efficiency, and gives scheduling control. Local future languages may sometimes lead to less waiting than in future-free languages due to more expressive synchronization control.

We have also seen that first-class futures may cause difficulties with respect to information security. In particular information flow analysis is problematic. Furthermore, the notion of futures, even local futures, make program reasoning more complex than reasoning for future-free languages, by adding a level of indirectness in the reasoning. The considered examples show properties where simple reasoning in the future-free setting becomes non-trivial in the case of futures, due to additional quantifiers. Static analysis has similar problems, and for first-class futures this is in general more complex than for local futures. For several kinds of static reasoning it is necessary to detect the set of calls that corresponds to a given **`get`** statement. In the case of first-class futures this set cannot be detected in class-wise analysis.

In general the more constructs a language has, the more expressive it is, but on the negative side, the more complex it is wrt. syntax, semantics, security, and analysis. This is illustrated clearly in the (somewhat oversimplified) table in Figure II.17. There is a trade-off between these different choices depending on the requirements in a given context, considering expressiveness, efficiency, and simplicity. The main benefits of first-class futures are the added flexibility and information sharing, some of which can be compensated by cooperative scheduling. For distributed IoT programs we have argued that first-class futures are less suited. This is also the case for information flow analysis. And if simplicity of program reasoning is a major concern, first class, and even local

futures, raise non-trivial complications. In our treatment, the limitations of future-free programming have been reduced by the addition of delegation and a syntactic tail construct for calls. Consequently, future-free programming can be attractive in a number of settings.

We have focused on imperative languages for single-threaded active objects. The setting of multi-threaded active objects is attractive in that it may provide higher efficiency, but its semantics is more complex. To keep our framework simple, we have avoided the multi-threaded setting, as well as advanced coordination constructs for futures, such as asynchronous continuations associated with futures. With respect to our evaluation criteria, these constructs increase the expressiveness and efficiency of first-class futures.

**Acknowledgements.** We thank the reviewers for their insight and thorough feedback, which definitely has increased the quality of the paper.

This work was partially supported by the project IoTSec - Security in IoT for Smart Grids, with number 248113/O70 part of the IKTPLUSS program funded by the Norwegian Research Council, and by the project SCOTT (www.scott-project.eu) funded by the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737422. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme of Austria, Spain, Finland, Ireland, Sweden, Germany, Poland, Portugal, Netherlands, Belgium, and Norway.

### Authors' addresses

**First Author** University of Oslo, Oslo, Norway, farzanka@ifi.uio.no

**Second Author** University of Oslo, Oslo, Norway, olaf@ifi.uio.no

**Third Author** University of Oslo, Oslo, Norway, toktamr@ifi.uio.no

Paper III

# Information-Flow-Control by means of Security Wrappers for Active Object Languages with Futures

Published in Nordic Conference on Secure IT Systems, 2020, Lecture Notes in Computer Science, volume 12556, pp. 74–91. DOI: 10.1007/978-3-030-70852-8_5

**Abstract**

This paper introduces a run-time mechanism for preventing leakage of secure information in distributed systems. We consider a general concurrency language model where concurrent objects interact by asynchronous method calls and futures. The aim is to prevent leakage of secure information to low-level viewers. The approach is based on a notion of *security wrappers*, where a wrapper encloses an object or a component and controls its interactions with the environment. Our run-time system automatically adds a wrapper to an insecure component. The wrappers are invisible such that a wrapped component and its environment are not aware of it.

The security policies of a wrapper are formalized based on a notion of security levels. At run-time, future components will be wrapped upon need, and objects of unsafe classes will be wrapped, using static checking to limit the number of unsafe classes and thereby reducing run-time overhead. We define an operational semantics and sketch a proof of non-interference. A service provider may use wrappers to protect its services in an insecure environment, and vice-versa: a system platform may use wrappers to protect itself from insecure service providers.

## III.1 Introduction

Given the large number of users and service providers involved in a distributed system, security is a critical concern. It is essential to analyze and control how confidential information propagates between nodes. When a program executes, it might leak secure information to public outputs or send it to malicious nodes. *Information-flow-control* approaches track how information propagates during execution and prevent leakage of secure information [71]. Program variables are tagged typically with security levels; such as *high* and *low*, to indicate secure

[0]The Norwegian Research Council has funded this work by project *IoTSec* (no. 248113/O70).

and public data. In this setting, an "attacker" could be seen as a low-level object that is not supposed to see high information. The basic semantic notion of information-flow security is *non-interference* [29]. This means that in any two executions of a program, if high inputs are changed, but low inputs are the same, then the low outputs will be the same (at least for locally deterministic programs). This way, an attacker (a low object) cannot distinguish between observable behaviors of the two executions since low outputs are independent of the high inputs [33].

We will consider a high-level model for object-oriented distributed systems suited for service-oriented systems, namely the *active object model* [8]. Method interaction is implemented by message passing; moreover, most active object languages support a communication paradigm called *futures* [8]. A future is a component that is created by a remote method call and eventually will contain the corresponding return value [5]. Therefore, the caller does not need to block while waiting to get the return value: it can continue with other tasks and later get the value from the corresponding future. Futures can be passed to other objects, called *first-class futures*. In this case, any object that has a reference to a future can access its content, which may be a security threat if the future contains secure data. Futures offer a flexible way of communication and sharing results, but handling them appropriately in order to avoid security leakages requires run-time checking (described in Sec. III.2.1).

Our goal is to design a permissive and precise security mechanism for controlling object communications in active object languages supporting first-class futures. Our security mechanism is inspired by the notion of *wrappers* in [63], where a wrapper encloses an object and enforces safety rules. In the present paper, we suggest a notion of *security wrapper*, which wraps an object or a future at run-time and performs security controls. Such wrappers are added by the operational semantics upon need, and a wrapped component and its environment are not aware of the presence of the wrapper. Security wrappers block object communications that lead to leakage of secure data to low objects. A future is wrapped if it contains a high value, and the wrapper blocks illegal access by low objects. The operational semantics of a wrapper is defined based on run-time security levels, resulting from a flow-sensitive information-flow enforcement [70]. We enrich the operational semantics with dynamic information-flow rules [70] where security levels of variables are allowed to change after an assignment. Therefore, our dynamic approach guarantees a degree of permissiveness and is precise since it deals with the exact run-time security levels.

The operational semantics of our security framework is provided in the style of Structured Operational Semantics (SOS). In order to minimize run-time overhead, we suggest static analysis to limit the number of classes where security checking and wrappers are needed since often only a few methods deal with secure information. In the resulting hybrid approach, the static analysis determines which classes cannot produce any high output, so-called *safe classes*, while the run-time system takes care of the precise security checking of objects of unsafe classes and futures created by such objects. Assuming a sound static analysis, we show that our proposed hybrid approach ensures the non-interference property.

In summary, our contributions are: i) a notion of security wrappers for enforcing noninterference and security control in object interactions (Sec III.4), ii) the use of static analysis to reduce the run-time overhead (Sec III.4.1), iii) defining the operational semantics for the dynamic information-flow enforcement with automatic deployment of wrappers (Secs. III.4.2, III.4.3) for our language (Sec III.4), and iv) an outline of the proof that our approach satisfies non-interference.

## III.2 Background

Information-flow-control approaches detect illegal flows. During program execution, there are two kinds of leakage of information, namely *explicit* and *implicit flows* [71]. For simplicity, we assume two security levels, $L$ (low) and $H$ (high). In the setting with observable and non-observable variables, an explicit flow happens when assigning a low variable ($l$) with a high value ($h$) by $l := h$. In the setting without observable variables, one may deal with this by letting the level of $l$ be dynamically changed to $H$. In an implicit flow, there is an indirect flow due to control structures. For example, in the **if** statement: $l := 0$ **if** $h$ **then** $l := 1$ **fi**, the guard $h$ is high, and it affects the value of $l$ indirectly. In order to avoid implicit flows, a program-counter label ($pc$) is introduced [71]. If the guard is high, then $pc$ becomes high, indicating a high context. (In run-time analysis, one may use a stack to deal with nested control structures.)

Information-flow-control approaches are divided into two categories, static and dynamic [70]. Static analysis is conservative [33]: to be sound, it over-approximates security levels of variables (for example, it over-approximates a formal parameter to high, while at run-time, a corresponding actual parameter can be low). This causes unnecessary rejections of programs, especially when the complete program is not statically known, as is usually the case in distributed systems. On the other hand, static analysis has less run-time overhead since security checks are performed before program execution [33]. *Dynamic information-flow* techniques perform security checks at run-time, and this introduces overhead. But they are more permissive and precise since they deal with the exact security levels instead of an over-approximation [33].

For example, consider the following method body:

{**if** *low_test* **then** $x :=$ *high_exp* **else** $x :=$ *low_exp* **fi**; **return** $x$}

where *low_test* and *low_exp* evaluate to low values, while *high_exp* evaluates to a high value. A sound static analysis will detect a high method result here since the value of *low_test* is not known; while at run-time, an execution of the method may give a low result (when *low_test* evaluates to *false*). The example shows that static analysis over-approximates the security level, in contrast to run-time analysis. Similarly, the parameter mechanism gives rise to static over-approximation. For a method $T$ *triv*($T$ $x$){**return** $x$}, where $T$ is a type containing both high and low values, static analysis will detect a (potentially) high result, whereas for calls with a low input value, the result is detected as low

at run-time. However, this could be handled by multiple static method profiles as in [59] (when low $T$ values are reflected by a subtype of $T$). For first-class futures, the situation is worse: a get statement on a future is detached from the call statement and also from the method name. Therefore, the static analysis of a get statement must over-approximate the level of the possible future values, while the exact level is revealed during run-time. This means that static analysis of security levels in languages with first-class futures can easily lead to a high degree of over-approximation.

In what follows, we briefly explain some of the terminologies of information-flow security that we use in this paper:

**Security levels**. Variables are tagged with security levels, organized by a partial order $\sqsubseteq$ and a join $\sqcup$ operator, such that $L \sqsubseteq H$ and $L \sqcup H = H$. The $\sqcup$ operator returns the least upper bound of two security levels. Inside a class, declarations of fields, class parameters, and formal parameters may have statically declared initial security levels. These levels may change with statements. We define a new syntax for object creation to assign security levels to objects.

**Flow-sensitivity**. By a *dynamic flow-sensitive analysis*, security levels of variables propagate to other variables, and precise levels are evaluated during execution. Variables start with their declared security levels (the ones without levels are assumed as $L$), but levels may change after each statement. In an assignment, the left-hand-side level becomes high if *pc* is high, or there is a high variable on the right-hand-side. The left-hand-side level becomes low if *pc* is low, and there is no high variable on the right-hand-side [70]. Otherwise, the security level of a variable does not change. E.g., a flow-sensitive analysis accepts the program $h := 0$; **if** $h$ **then** $l := 1$ **fi**; **return** $l$; since the level of $h$ is updated to $L$ after the first assignment, hence there is no leakage. In **if** statements, in order to avoid implicit flows, when the guard is high, the security levels of variables appearing on the left-hand side of assignments in the taken and untaken branches are raised to high [70]. E.g., considering an initial environment $\Gamma = \{h \mapsto H, l_1 \mapsto L, l_2 \mapsto L\}$ and the program: **if** $h$ **then** $l_1 := 1$ **else** $l_2 := 0$ **fi** when the condition is true, $\Gamma$ changes to $\Gamma = \{h \mapsto H, l_1 \mapsto H, l_2 \mapsto H\}$ for a sound flow-sensitive analysis [70]. In a dynamic approach, in order to have a sound flow-sensitive analysis, the assigned variables in the untaken branches should be given to the analysis, which can be provided by static analysis of the program code [6, 70].

### III.2.1 Active object languages

Active object languages are based on a combination of the *actor model* [1] and object-oriented features [8]. Some well-known active object languages are Rebeca [76, 78], Scala/Akka [30, 84], Creol [39], ABS [43], Encore [9], and ASP/ProActive [13, 15]. In communication with futures, when a remote method call is made, a future object with a unique identity is created. Futures can be explicit with a specific type and access operations like in ABS or can be implicit with automatic creation and access [8]. E.g., in ABS, explicit futures are created as in $Fut[T]\ f := o!m(\overline{e}); v := f.\mathbf{get}$, where $f$ is a future variable of type $Fut[T]$,

and $T$ is the type of the future value. The symbol "!" indicates an asynchronous method call $m$ of object $o$ with actual parameters $\overline{e}$, and the future value is retrieved with a **get** construct when needed. The variable $f$ can be passed to other objects as a parameter (first-class futures). The caller may continue with other processes while the callee is computing the return value. The callee sends back the return value to the corresponding future, and then the future is called *resolved*. A synchronous call is denoted by $o.m(\overline{e})$, which blocks the caller until the return value is retrieved.

**Information-flow security with futures.** Static analysis is in general difficult for programs with futures, where the result of a call is no longer syntactically connected to the call, compared to the call/return paradigm in languages without futures [45]. For example, a future may be created in one module and received as a parameter in another. Thus, a future may not statically correspond to a unique call statement. One could overestimate all future values as high, but this would severely restrict the set of acceptable programs. It would be better to overestimate the set of possible call statements that corresponds to a given get statement, but this requires access to the whole program, which is often problematic for distributed systems. Moreover, the return values of these overestimated calls may have different security levels, which also results in overestimation.

A static analysis that assumes references as low, allows passing of future references. However, the exact security level of a future value is revealed when it becomes resolved, which goes beyond static analysis. For example, if a low-level object performs $x := f.\mathbf{get}$, and $f$ refers to a future with a high value, it is a leakage of information. A dynamic approach is required to control access to a future value at run-time when it is resolved, and if the value is high it needs protection. The futures concept makes static checking less precise, and the need for complementary run-time checking is greater, as provided in the present paper.

## III.3 Our core language

In order to exemplify our security approach, the security semantics (in Sec. III.4.2) is embedded in a simple, high-level core language. All remote calls are made by means of futures, where the method result is always returned to the corresponding future. Figure III.1 gives the syntax of statements. The statement $f := o!m(\overline{e})$ is an asynchronous call with futures, and $o!m(\overline{e})$ is an asynchronous call without waiting for the result and associating a future. We define an extended syntax for object creation $\mathbf{new}_{lev}\ c(\overline{e})$, where *lev* is the object's level (it can be $L$ or $H$).

Figure III.2 illustrates a health care service in our core language, involving futures for the sharing of secure medical records. Personnel and patients with lower-level access are not allowed to access medical records. High variables are emphasized based on user specifications, in this case reflecting patients' medical test results. The server, specified by the class *Service*, searches for a patient's test result, and the object *proxy* publishes the result to the patient and personnel. In Fig III.2, in line 10, a produce cycle is initiated between the *server* and *proxy*.

| *Basic constructs* | |
|---|---|
| $x :=$ **new**$_{lev}$ $c(\overline{e})$ | object creation with the security level $lev$ |
| **return** $e$ | creating a method result/future value |
| **if** $b$ **th** $s$ [**el** $s'$] **fi** | if statement ($b$ a Boolean condition) |
| $f := o!m(\overline{e})$ | remote asynchronous call, future variable $f$ |
| $x := f$.**get** | blocking access operation on future $f$ |
| $o!m(\overline{e})$ | simple asynchronous remote call |

Figure III.1: Statement syntax. Here $\overline{e}$ is an expression list. Brackets denote optional parts.

```
1  data type Result = ... // definition of medical data
2  interface ServiceI { Void produce() ... }
3  interface ProxyI { Void publish(Fut[Result] q, PatientI a, List[PersonnelI] d) ... }
4  interface LabI { Result_H search(PatientI a) ... }
5  interface PatientI { Void send(Result_H r) ... }
6  interface PersonnelI { Void send(Result_H r) ... }
7  interface DataBase { ... }
8  class Service(LabI lab, DataBase db) implements ServiceI {
9      ProxyI proxy = new_H Proxy(this);
10     this!produce();   // initial action, starting a produce cycle
11     Void produce() { Fut[Result] f;  PatientI a;   List[PersonnelI] d = Nil;
12         ....     // finding a patient and the associated personnel in a database
13         f:=lab!search(a); // searching for the test result of patient a
14         proxy!publish(f, a, d); } } //sending the future f, not waiting for the result
15
16 class Proxy(ServiceI s) implements ProxyI{ Result_H x;
17     Void publish(Fut[Result] f, PatientI a, List[PersonnelI] d) {
18         x:=f.get;   // waiting for future and assigning the value to x. x becomes H
19         a!send(x);   // x is now H
20         d!send(x);   // multicasting, x is H
21         s!produce(); } }
```

Figure III.2: Example of sharing *high* patients' test results by means of futures.

In line 13, the server searches for the test result of a patient with the userId $a$ by sending a remote asynchronous call to the laboratory $f :=$ *lab*!*search(a)*, where $f$ is the future variable. In line 14, the server calls *proxy*!*publish(f, a, d)* and passes the future $f$, userId $a$, and personnelId $d$ to the object *proxy*. Both *search* and *publish* are asynchronous calls, thus the server does not wait for the return values and is free to respond to any client request. In line 18, the object *proxy* waits for the test result and assigns the result to variable $x$ by performing $x := f$.**get**. Then *proxy* sends $x$ to the patient and personnel.

A static analysis over-approximates the security levels of test results as high, which leads to rejections of information passing. Note that the two *send* calls in the class *Proxy* would not be allowed if we only use static checking since we cannot tell which patients and personnel have a high enough level. A static analysis which considers references as low allows passing the future $f$ to the object *proxy* (line 14), but later when it is resolved, the future value can be high, and the *proxy* compromises security by sending this value to other objects.

## III.4 A framework for non-interference

Like Creol, our core language is equipped with interface encapsulation, which means that created objects are typed by interfaces, not classes [39]. As a result, remote access to fields or methods that are not declared in an interface is impossible. Therefore, observable behavior of an object is limited to its interactions through remote method calls. Illegal object interactions are the ones leading to an information-flow from high information to low level objects. An object can reveal confidential information in method calls by sending actual parameters with high security levels to low-level objects. If a future contains data with a high security level, low-level objects' access is illegal.

We exploit the notion of wrappers to perform dynamic checking for enforcing non-interference in object interactions. A wrapper blocks illegal interactions. Wrappers' security policies are based on run-time security levels. Inside an object, in order to compute the exact security levels of created messages or return values, the flow-sensitivity must be active. The operational semantics for the dynamic flow-sensitive analysis, is given in Sec. III.4.2 and for wrappers, in Sec. III.4.3.

We can be conservative and wrap all objects and correspondingly activate flow-sensitivity, but this will cost run-time overhead. In order to be more efficient at run-time, it is important to perform dynamic checking only for components where it is necessary. We benefit from static analysis to categorize a class definition as *safe* or *unsafe*. *A class is safe if it does not have any method calls with high actual parameters and return values. A class is unsafe if it has a method call with at least one high actual parameter or a high return value.* Objects created from unsafe classes are wrapped, and flow-sensitivity will be active inside these objects. Objects from safe classes do not need a wrapper or active flow-sensitivity. This will make the execution of objects of safe classes faster, as we avoid a potentially large number of run-time checks and wrappers.

### III.4.1 Static analysis

Our security approach can be combined with a sound static over-approximation for detecting security errors and safe classes, e.g., the one proposed in [60], which is more permissive (to classify a class as safe) than the static analysis indicated here, in that high communication is considered secure as long as the declared levels of parameters are respected. In a class, variables are declared with maximum security levels (the maximum level that can be assigned at run-time). The same for future variables at the time of declaration, for example, $Fut[T_H]\ x$ indicates that $x$ is a high future variable. Local variables without a declared security level start with the level $L$ (as default) but may change after each statement due to the flow-sensitivity. Dataflow typing rules inside an object can be defined similar to [60]; however, we change the typing rules for method calls and return values to classify unsafe and safe classes. A class is defined as safe if the confidentiality of each method is satisfied. The confidentiality of a method is satisfied if the typing rules for its return value and actual parameters

| | | |
|---|---|---|
| config | $::= \epsilon \mid$ object $\mid$ flowsen-obj $\mid$ msg $\mid$ future $\mid$ wrapper $\mid$ class $\mid$ config config | |
| object | $::= ob(o, a, p, lev)$ | $d ::= v \mid v_{lev}$ |
| flowsen-obj | $::= ob(o, a, p, lev, pcs)$ | $p ::= (l, s) \mid idle$ |
| msg | $::= invc(f, m, \overline{d}, o)_{lev} \mid Comp(d, f)_{lev}$ | |
| future | $::= fut(f, d)$ | |
| wrapper | $::= Wr\{wId, lev \mid$ config $\}$ | |
| class | $::= Cl(c \mid a', mm)_{lev}$ | |

Figure III.3: The components of a configuration.

are satisfied. The typing rules check that each occurrence of an actual parameter and a return value are not high; then, the class is safe; otherwise, it is unsafe and needs dynamic checking. The typing rule for getting a future, checks that if a future variable is high, then the class is classified as unsafe. Alternatively, we could have used another sound static analysis, for instance (the relevant parts of) the static analysis defined for ABS in [64], and adapt it to our setting.

We categorize safe and unsafe classes for the example in Fig. III.2. The interface laboratory *LabI* has a method with a high return value (*search*). Thus the object *lab* is unsafe and flow-sensitivity is active to compute the security level of the return value at run-time. The class *Proxy* is unsafe since it has at least one method call with a high actual parameter ($a!send(x)$), thus object *proxy* is active flow-sensitive and wrapped.

### III.4.2 Security semantics

We here discuss the operational semantics of our core language with the embedded notions of flow-sensitivity and security wrappers in Figs. III.4, III.5. The small-step operational semantics is defined by a set of rewrite rules [52]. In a rule, premises are above the line and one step rewrite is under the line. A rule is applied to a subset of a configuration if the premises are satisfied, and the subset is changed from the left-hand-side to the right-hand-side of the rewrite rule.

In Fig. III.3, an execution state is modeled as a configuration **config**, which is a multiset of objects (with or without active flow-sensitivity), messages, futures, wrappers, and classes. (Classes are included in a configuration to provide static information about fields and methods.) An **object** is represented as: $ob(o, a, p, lev)$, where $o$ is the object identity, $a$ is the field state, $p$ is the current *active process*, and $lev$ is the object's level ($lev \in \{L, H\}$). An active process $p$ is a pair $(l, s)$, where $l$ is the local variables state, and $s$ is a list of statements, or it is *idle* representing an empty local state and no statements. A state is a mapping (substitution) binding variables to values. A **flowsen-obj** represents an flow-sensitive object with an extra field *pcs* that denotes a stack of context security levels inside an object, where $pcs = emp$ denotes an empty stack.

A **class** is represented as: $Cl(c \mid a', mm)_{lev}$, where $c$ is the class name, $a'$ is the initial state of the class fields (attributes), $mm$ is a multiset of method declarations (with local variables and code), and $lev$ denotes the type of the class, i.e., if $lev = L$, the class is safe, and if $lev = H$, the class is unsafe. A **msg** represents an invocation message or a completion message. In an invocation message, $f$ is the future identity, $m$ is the method name, $\overline{d}$ is a list of actual parameters, and $lev$ is a level attached to the message at time of creation. If a message is created in a high context, then $lev = H$; otherwise, $lev = L$. A completion message contains a return value $d$ and a future identity $f$, and $lev$ represents the context level. The notation $d$ denotes a value $v$ or a value with security level $v_{lev}$. The **future** component shows a resolved future with identity $f$ and the value $d$, and $fut(f, \_)$ denotes an unresolved future. A **security wrapper** is represented as: $Wr\{wId, lev \mid config\}$, where $wId$ is the wrapper's identity, $lev$ is the level, and $config$ denotes the configuration inside the wrapper.

*Auxiliary functions.* Let $\Gamma$ be a mapping and $[x \mapsto d]$ be a binding, mapping $x$ to $d$. The notation $\Gamma[x \mapsto d]$ represents the update of $\Gamma$ with the binding. The look-up function is represented as $\Gamma(x)$, where $\Gamma[x \mapsto d](x) = d$. The map composition $a\#l$ indicates that the binding of a variable in the inner scope $l$ shadows any binding of that variable in the outer scope $a$. Thus $a\#l(x)$ gives $l(x)$ when defined, otherwise $a(x)$. Consider an object with attribute state $a$ and local state $l$. Then the composition $a\#l$ defines the *object state*. The notation $[\![e]\!]$ denotes the evaluation of expression $e$, where variables are evaluated according to the object state. The evaluation in $[\![e]\!]$ is strict in the sense that the resulting level is high if $e$ contains variables that have a high security level. Other auxiliary functions are given as follows:

- The function $level(d)$ returns the security level of $d$, such that $level(v_{lev}) = lev$, and for an untagged value $level(v) = L$. If $\overline{e}$ is a list of expressions, then $[\![\overline{e}]\!] = \overline{d}$ returns a list of data, and $level(\overline{d}) = \sqcup\ level(d_i), \forall d_i \in \overline{d}$ (the join of all data in $\overline{d}$).
- The function $level(o)$ returns the level of the object $o$.
- The function $level(pcs)$ returns the join of security levels in $pcs$, where if $pcs = emp$, $level(pcs) = L$, and if $pcs \neq emp$, $level(pcs) = H$.
- The function $update_H(s)$ raises the security levels of variables appearing in the left-hand-side of assignments in $s$ to high.
- The function $fresh()$ returns a unique identity for an object or a future.
- The function $bind(o, m, \overline{d}, f)$ returns a process, where the method $m$ in the class of the object $o$ is activated, and the method' parameters are bound to the actual ones $(\overline{d})$, and a reserved local variable $label$ is bound to $f$, denoting where to send the return value of the method [43].
- The function $bind(o, m, \overline{d})$ returns a process without the binding for the $label$, in case the method's result is not needed.

- The function $safe(Cl\,(c \mid a, mm)_{lev})$ returns true if $lev = L$ and false otherwise.

Figure III.4 represents the flow-sensitivity semantics of objects. The NEW rule shows the command $x := \textbf{new}_{lev} c'(\overline{e})$ in the active process of an object $o$, where $c'$ is an unsafe class. The rule creates an active flow-sensitive object $o'$ and a wrapper and assigns $o'$ to $x$. The active process of the new object $o'$ is initially *idle*, denoting an empty active process. The level of $o'$ is $lev$ as it is specified in the command $\textbf{new}_{lev} c'(\overline{e})$, if not, the level is assumed low. The stack of *pcs* is empty, denoted by *emp*. The wrapper has the same identity ($o'$) and the level ($lev$) of the object $o'$. The semantics of the actual class parameters is treated like parameters of an asynchronous call $x!init(\overline{e})$ (creating an invocation message by the rule CALL), where *init* is the name of the initialization method of a class. Note that if $\overline{e}$ contains high security level data, the wrapper does not send the corresponding invocation message to the new object if the new object is low-level (see rule WR-INVC-ERROR in Fig. III.5, which we explain later). The Rule ASSIGN-LOCAL shows an assignment $x := e$, where $x$ is in the local state $l$, $e$ is evaluated to $v_{lev'}$, and $x$ is updated in $l$ with the new value $v$ and the level $lev' \sqcup level(pcs)$. Therefore, the level of $x$ is updated with the right-hand-side level joined with that of *pcs*. In IF-LOW-TRUE, the guard's security level is low, and the guard is true ($true_L$), thus the corresponding branch $s'$ is taken. While in IF-LOW-FALSE, since the guard is false, the else branch $s''$ is taken. In IF-HIGH-TRUE and IF-HIGH-FALSE, since the guard's security level is high, similar to the approach in [70], the security levels of variables appearing in assignments in both branches are raised to high to avoid implicit flows. In the rules, the guard's security level $H$ is pushed to the *pcs* stack, resulting in a high security context, where all the messages created in a high context will have high security levels (see rules CALL-FUT, CALL). Moreover, assignments in the taken branch result in high security levels (see ASSIGN-LOCAL and ASSIGN-ATTRIBUTE). The added statement $endif(s'')$, where $s''$ is the untaken branch, marks the join point of the **if** structure and raises the assigned variables' levels in the untaken branch. In the ENDIF rule, the function $update_H(s'')$ raises the security levels of variables appearing in the left-hand-side of assignments in $s''$ to high, and these variables are updated in the local state. Moreover, the last element of *pcs* is removed ($pcs.pop()$), reflecting the previous context level.

In the rules, we do not cover local calls, which do not involve object interactions (therefore, less interesting here). The CALL-FUT rule deals with an asynchronous call $x := e!m(\overline{e})$, where $x$ is a future variable, and $e$ is the callee. The call generates a (not resolved) future with a unique identity $f$, where $f$ is assigned to $x$, and an invocation message containing $f$, $m$, actual parameters $\overline{d}$, and the callee $o'$. The invocation message's level is $level(pcs)$, which is needed to avoid indirect leakage from the caller. The rule CALL shows an asynchronous call $e!m(\overline{e})$ without an associated future, where the method's result is not needed. The call creates an invocation message containing $m$, $\overline{d}$, and the callee $o'$, and the message' level is $level(pcs)$. The START-FUT rule is applied when an object is *idle*, and there is an invocation message to the object. The object's active process

$$\textsc{new}\quad \frac{o' = \mathit{fresh}() \quad \mathit{false} = \mathit{safe}(C')}{\begin{array}{c}ob(o,a,(l,x := \mathbf{new}_{lev}\ C'(\overline{e});s),lev') \to ob(o,a,(l,x := o';x!\mathit{init}(\overline{e});s),lev') \\ Wr\{o',lev \mid ob(o',a'[\mathit{this} \mapsto o'],\mathit{idle},lev,emp)\}\end{array}}$$

$$\textsc{assign-local}\quad \frac{x \in dom(l) \quad v_{lev'} = [\![e]\!]}{ob(o,a,(l,x := e;s),lev,pcs) \to ob(o,a,(l[x \mapsto v_{lev' \sqcup level(pcs)}],s),lev,pcs)}$$

$$\textsc{assign-attribute}\quad \frac{x \in dom(a) \quad v_{lev'} = [\![e]\!]}{ob(o,a,(l,x := e;s),lev,pcs) \to ob(o,a[x \mapsto v_{lev' \sqcup level(pcs)}],(l,s),lev,pcs)}$$

$$\textsc{if-low-true}\quad \frac{\mathit{true}_L = [\![e]\!]}{ob(o,a,(l,\mathbf{if}(e)\ s'\ \mathbf{el}\ s''\ \mathbf{fi};s),lev,pcs) \to ob(o,a,(l,s';s),lev,pcs)}$$

$$\textsc{if-low-false}\quad \frac{\mathit{false}_L = [\![e]\!]}{ob(o,a,(l,\mathbf{if}(e)\ s'\ \mathbf{el}\ s''\ \mathbf{fi};s),lev,pcs) \to ob(o,a,(l,s'';s),lev,pcs)}$$

$$\textsc{if-high-true}\quad \frac{\mathit{true}_H = [\![e]\!]}{\begin{array}{c}ob(o,a,(l,\mathbf{if}(e)\ s'\ \mathbf{el}\ s''\ \mathbf{fi};s),lev,pcs) \to \\ ob(o,a,(l,s';\mathit{endif}(s'');s),lev,pcs.\mathit{push}(H))\end{array}}$$

$$\textsc{if-high-false}\quad \frac{\mathit{false}_H = [\![e]\!]}{\begin{array}{c}ob(o,a,(l,\mathbf{if}(e)\ s'\ \mathbf{el}\ s''\ \mathbf{fi};s),lev,pcs) \to \\ ob(o,a,(l,s'';\mathit{endif}(s');s),lev,pcs.\mathit{push}(H))\end{array}}$$

$$\textsc{call}\quad \frac{o' = [\![e]\!] \quad \overline{d} = [\![\overline{e}]\!]}{\begin{array}{c}ob(o,a,(l,e!m(\overline{e});s),lev,pcs) \\ \to ob(o,a,(l,s),lev,pcs) \\ \mathit{invc}(m,\overline{d},o')_{level(pcs)}\end{array}}$$

$$\textsc{start-fut}\quad \frac{p = \mathit{bind}(o,m,\overline{d},f)}{\begin{array}{c}ob(o,a,\mathit{idle},lev,pcs)\ \ \mathit{invc}(f,m,\overline{d},o)_{lev'} \\ \to ob(o,a,p,lev,pcs.\mathit{push}(lev'))\end{array}}$$

$$\textsc{call-fut}\quad \frac{f = \mathit{fresh}() \quad o' = [\![e]\!] \quad \overline{d} = [\![\overline{e}]\!]}{\begin{array}{c}ob(o,a,(l,x := e!m(\overline{e});s),lev,pcs) \\ \to ob(o,a,(l,x := f;s),pcs) \\ \mathit{fut}(f,\_)\mathit{invc}(f,m,\overline{d},o')_{level(pcs)}\end{array}}$$

$$\textsc{start}\quad \frac{p = \mathit{bind}(o,m,\overline{d})}{\begin{array}{c}ob(o,a,\mathit{idle},lev,pcs)\ \ \mathit{invc}(m,\overline{d},o)_{lev'} \\ \to ob(o,a,p,lev,pcs.\mathit{push}(lev'))\end{array}}$$

$$\textsc{endif}\quad \frac{l' = l[\mathit{update}_H(s')]}{\begin{array}{c}ob(o,a,(l,\mathit{endif}(s');s),lev,pcs) \\ \to ob(o,a,(l',s),lev,pcs.\mathit{pop}())\end{array}}$$

$$\textsc{return}\quad \frac{d = [\![e]\!] \quad f = l(\mathit{destiny})}{ob(o,a,(l,\mathbf{return}(e);),lev,pcs) \to ob(o,a,\mathit{idle},lev,pcs) \quad \mathit{Comp}(d,f)_{level(pcs)}}$$

Figure III.4: Flow-sensitive operational semantics, $lev, lev' \in \{l, H\}$.

is updated with $p$, which is the *bind*'s result, where method $m$ is activated, formal parameters are bound to the actual ones ($\overline{d}$), and the local variable *label* is bound to the future identity ($f$) for sending the method's result to the future by a **return** statement. The level of the received message $lev'$ is added to the object's stack *pcs*. This avoids implicit leakage from the sender. In the START rule, the invocation message does not contain a future identity, and the object starts execution the corresponding method, which is activated by the *bind* function without the binding for the *label* variable. The RETURN rule interprets a **return** statement, which creates a completion message to the corresponding future, which is looked up in the local state ($l(label)$), and the object becomes *idle*. The security level of the completion message is $level(pcs)$ to avoid indirect leakage from the callee to the recipients of the future value. We assume that each method body ends with a **return** statement. The rules for objects without active flow-sensitivity are similar but without security levels, *pcs*, and wrappers.

### III.4.3 Operational semantics of security wrappers

In this section, we discuss the operational semantics of security wrappers. As mentioned, a wrapper for an object is created in the rule NEW in Fig.III.4. A wrapper has the same identity as the wrapped component; thereby, the wrapper represents the component to the environment. Invocation messages generated by the CALL-FUT and CALL rules will first meet the object's wrapper for security checking before being sent to the callee. The WR-INVC rule in Fig. III.5, represents a wrapper with an invocation message inside, which is produced by the object $o$. If the join ($\sqcup$) of the message's level $lev'$ and the actual parameters' levels $level(\overline{d})$ is less than or equal to the destination object' level ($level(o')$), then the wrapper allows the message to go out. In WR-INVC-ERROR, since the recipient object's level is less than the message's level, the invocation message is deleted and the corresponding future value is replaced by an error value. This can be combined with an exception handling mechanism such that an exception is raised when a get operation tries to access an error value. However, as this is beyond the scope of this paper, we ignore the exception handling part. We simply indicate exceptions by assignments with **error** in the right-hand-side. The ERROR-FUT rule represents the case where a future value is **error**; the object performing the get command $x := e.\mathbf{get}$, where $e$ refers to the future, assigns an error to $x$. The rule ASSIGN-ATTRIBUTE shows an assignment, where $x$ is in the object's fields.

The INVC-WR rule represents a wrapper and an incoming invocation message to the object $o$. The notation $\Lambda[m, i]$ indicates the level of the $i$th formal parameter of the method $m$ as declared in the class. If the security level of each actual parameter ($lev_i$) is less than or equal to the security level of the corresponding formal parameter, then the wrapper allows the message to go through and adds it to its configuration inside. Otherwise, the invocation message is deleted in INVC-WR-ERROR. In LOW-FUT, an unresolved future gets the corresponding completion message containing $d$, hence the future becomes resolved with $d$. The join of the message's level $lev$ and $level(d)$ is low, thus no

$$\textsc{wr-invc}\quad \frac{lev' \sqcup level(\overline{d}) \sqsubseteq level(o')}{Wr\{o, lev \mid invc(f, m, \overline{d}, o')_{lev'}\ config\ \} \rightarrow Wr\{o, lev \mid config\ \}\ invc(f, m, \overline{d}, o')_{lev'}}$$

$$\textsc{high-fut}\quad \frac{H = level(d) \sqcup lev}{fut(f, \_)\ \ Comp(d, f)_{lev} \rightarrow Wr\{f, H \mid fut(f, d)\ \}}$$

$$\textsc{wr-invc-error}\quad \frac{lev' \sqcup level(\overline{d}) \sqsupset level(o')}{Wr\{o, lev \mid invc(f, m, \overline{d}, o')_{lev'}\ config\ \}\ fut(f, \_) \rightarrow Wr\{o, lev \mid config\ \}\ fut(f, \textbf{error})}$$

$$\textsc{low-fut}\quad \frac{L = level(d) \sqcup lev}{fut(f, \_)\ \ Comp(d, f)_{lev} \rightarrow fut(f, d)}$$

$$\textsc{error-fut}\quad \frac{f = [\![e]\!]}{fut(f, \textbf{error})\ \ ob(o, a, (l, x := e.\textbf{get}; s), lev, pcs) \rightarrow fut(f, \textbf{error})\ \ ob(o, a, (l, x := \textbf{error}; s), lev, pcs)}$$

$$\textsc{high-fut}\quad \frac{H = level(d) \sqcup lev}{fut(f, \_)\ \ Comp(d, f)_{lev} \rightarrow Wr\{f, H \mid fut(f, d)\}}$$

$$\textsc{invc-wr}\quad \frac{\forall lev_i \in level(\overline{d}) : lev_i \sqsubseteq \Lambda[m, i]}{Wr\{o, lev \mid config\}\ invc(f, m, \overline{d}, o)_{lev'} \rightarrow Wr\{o, lev \mid invc(f, m, \overline{d}, o)_{lev'}\ config\}}$$

$$\textsc{invc-wr-error}\quad \frac{\exists lev_i \in level(\overline{d}) : lev_i \sqsupset \Lambda[m, i]}{Wr\{o, lev \mid config\ \}\ invc(f,\ m, \overline{d}, o)_{lev'} \rightarrow Wr\{o, lev \mid config\}}$$

$$\textsc{error-high-fut-get}\quad \frac{f = [\![e]\!]\ \ lev \sqsubset H}{Wr\{f, H \mid fut(f, d)\}\ ob(o, a, (l, x := e.\textbf{get}; s), lev, pcs) \rightarrow Wr\{f, H \mid fut(f, d)\}\ ob(o, a, (l, x := \textbf{error}; s), lev, pcs)}$$

$$\textsc{high-fut-get}\quad \frac{f = [\![e]\!]\ \ lev \sqsupseteq H}{Wr\{f, H \mid fut(f, d)\}\ ob(o, a, (l, x := e.\textbf{get}; s), lev, pcs) \rightarrow Wr\{f, H \mid fut(f, d)\}\ ob(o, a, (l, x := d; s), lev, pcs)}$$

$$\textsc{low-fut-get}\quad \frac{f = [\![e]\!]}{fut(f, d)\ \ ob(o, a, (l, x := e.\textbf{get}; s), lev, pcs) \rightarrow fut(f, d)\ \ ob(o, a, (l, x := d; s), lev, pcs)}$$

Figure III.5: Operational semantics involving wrappers, $lev, lev' \in \{l, H\}$.

wrapper is created. In HIGH-FUT, $lev \sqcup level(d) = H$, thus the future becomes wrapped and resolved. Since the future is high, a wrapper is created to protect it, and the wrapper has the same identity and level as the future. The ERROR-HIGH-FUT-GET rule represents a wrapped future and an object that wants to get the future value. If the security level of the object ($lev$) asking for the value is less than the wrapper ($H$), then the wrapper sends an error value. In HIGH-FUT-GET, the object gets the value from the wrapped future since the object's level is greater than or equal to $H$. The LOW-FUT-GET rule shows that an object gets the value from an unwrapped future without security checking.

### III.4.4 Non-interference

We show that our security framework satisfies non-interference. Non-interference considers the observable behavior of different executions. The *observable behavior* of an object consists of invocation messages and completion messages. Even the observable behavior of object creation, by the NEW rule in Fig. III.4, is an asynchronous call $x!init(\overline{e})$, which creates an invocation message. Since object and future identities may change from execution to execution, we must compare executions relative to a correspondence of such identities in one execution to those in another execution. Corresponding objects must be of the same class.

A message is said to be low if it does not have a high tag nor contain any parameters with high tags. Two low messages are *indistinguishable*, $\simeq$, if the identities in the messages correspond to each other, and other values are equal. Two execution states of corresponding objects $o$ and $o'$ are said to be *indistinguishable* if the values of their local variables and attributes are indistinguishable and they have the same remaining statement lists, and also agree on other system variables, including flow sensitivity (with same values of $pcs$).

**Definition III.4.1.** Global non-interference means that for any two executions with corresponding objects and futures, such that the history of messages consumed or produced by an object in one execution state is indistinguishable from that of the corresponding object in a state of the other execution, and such that the next communication event of the first object is a low output, then the next low communication output event of the other object will be indistinguishable.

**Definition III.4.2.** Local non-interference means that for two executions with corresponding objects $o$ and $o'$, and for execution states where $o$ and $o'$ are non-idle and where the execution states of $o$ and $o'$ are indistinguishable, the next execution states of these objects will also be indistinguishable when both have executed the next statement, and in case the statement gives an output, both make indistinguishable output (or neither makes no low output).

Note that our security approach includes termination aspects. We next prove that each object is *locally deterministic*, in the sense that the next state of a statement, other than idle and get, is deterministic, i.e., depending only on the

prestate. The only source of non-determinism is get and the independent speed of the objects, which means that the ordering in the messages queues is in general non-deterministic. Thus only idle states and get cause local non-determinism.

**Lemma III.4.3.** *In our security model, each object is locally deterministic.*

*Proof.* According to our operational semantics, for each statement (other than idle and get) there is only one rule to apply, and for an **if** statement, the choice of the rule is given deterministically by testing the security level and value of the guard. There is no interleaving of processes inside an object as well. ■ ■

**Definition III.4.4.** Low-to-low determinism means that any low part of a state or output resulting from a statement, other than get, is determined by the low part of the prestate and the statement, when ignoring states where *pcs* is high.

**Lemma III.4.5.** *In our security model, each object is low-to-low deterministic.*

*Proof.* This can be proved by case analysis on the statements. For an **if** with a high test, the taken branch does not result in low state changes nor low outputs. In particular, any invocation message made has label $H$, and the execution of that method invocation by the same or another object, will start in a high context (see the START-FUT and START rules), and so will a new object created from the branch. This ensures that there is no implicit leakage from a high branch. However, the choice of branch could depend on high information, and lead to distinguishable states, but this is compensated by $endif(s'')$, which raises the level of variables updated in the untaken branch $s''$. For an **if** with low test, the choice of branch is given by the low part of the prestate and the test. For an assignment, the level of the left-hand-side becomes low if the level of the right-hand-side is low and *pcs* is low. Otherwise, the left-hand-side' level becomes high after the assignment. The cases for the other statements are straightforward. ■ ■

**Theorem III.4.6.** *Our security model guarantees local and global non-interference, and an attacker (i.e., a low object) will only receive low information.*

*Proof.* Local non-interference can be proved by induction of the number of execution steps considering two executions of an object. The low part of each state and the low outputs must be the same by the two previous lemmas, using the fact that future values of corresponding futures will be indistinguishable, since these are given by earlier outputs, which are indistinguishable by the induction hypothesis. Global non-interference can be proved by induction on the number of steps considering two executions. It follows by local non-interference for all objects. Since an attacker is a low object, the wrappers will prevent it from receiving high inputs. ■ ■

This theorem implies that an attacker will not be able to obtain high information explicitly or implicitly, nor obvserve difference of termination aspects.

## III.5 Related work

Starting with the work of Denning and Denning [22], a number of static techniques for lattice-based security information flow analysis have been suggested.

In [60], a secure type system has been suggested for Creol without futures to enforce noninterference in object interactions. Typing rules check that the security levels of variables respect the declared security levels in the interfaces. In [60], since the run-time security levels of objects, indicating the access rights, might not be available at static time, an if-test construct is added to check the security level of an object before sending data. Our approach is a dynamic technique, which is more permissive and precise and supports futures confidentiality. In [64], Pettai and Laud present a type system for ABS to ensure non-interference by means of over-approximation. E.g., a future's security level is the upper bound of the tasks' levels that the future refers to, while our run-time system does not use over-approximation (assuming the labels are exact). This work also deals with other concurrency features of ABS such as cogs and synchronization between tasks, where security issues are prevented by using the operational semantics and the type system. The cog feature of ABS is not relevant to our paper.

In [2], a dynamic information-flow-control approach is performed for the ASP language. Security levels are assigned to activities and communicated data. The security levels do not change when they are assigned. Dynamic checks are performed at activity creations, requests, and replies. Since future references are not confidential, they are passed between activities without dynamic checking, but getting a future value is checked by a reply transmission rule. In [2], the security model guarantees data confidentiality for multi-level security (MLS) systems. Our approach adds flow-sensitivity, which allows security levels of variables to change during execution of an object. It makes our approach more permissive and a wrapper deals with run-time security levels. In addition to enforcing the non-interference property in object interactions, our approach guarantees that an object will be given access only to the information that it is allowed to handle.

In [57], Nair et al. implement and design a run-time system, named Trishul, to track the flow of information within the Java virtual machine (JVM). This paper focuses on implicit and explicit flows through the Java control flows and the architecture and does not enforce non-interference. Due to the Trishul's modular nature, our security wrappers can be deployed to prevent illegal flows.

Russo and Sabelfeld [70] prove that a sound flow-sensitive dynamic information-flow enforcement is more permissive than static analysis. In [48], the notion of wrappers is used to control the behavior of JavaScript programs and enforce security policies to protect web pages from malicious codes. A policy specifies under which conditions a page performs a specific action, and a wrapper grants, rejects, or modifies these actions. Moreover, the notion of wrappers has been developed for the safety of objects [63], where the programmer needs to specify which objects should have a wrapper and to program what each wrapper should do based on any input/output. In contrast, we apply wrappers to security analysis, letting the runtime system automatically decide which components

should be wrapped, and also what the wrappers should do to prevent illegal flows.

## III.6 Conclusion

We have proposed a framework for enforcing secure information-flow and non-interference in active object languages based on the notion of security wrappers. We have considered a high-level core language supporting asynchronous calls and futures. In our model, due to encapsulation, there is no need for information-flow restrictions inside an object. Wrappers perform security checks for object interactions (with methods and futures) at run-time. Furthermore, wrappers control the access to futures with high values. Security rules of wrappers are defined based on security levels of communicated messages. Inside an object, the security levels of variables might change at run-time due to flow-sensitivity. Wrappers on unsafe objects and future components protect exchange of confidential values to low objects. Wrappers on objects protect outgoing method calls and prevent leakage of information through outgoing parameters. The wrappers are created automatically by the run-time system without the involved parties being aware of it. Their behavior is also defined by the runtime system. We define non-interference for our language and outline a proof of it. By combining results from static analysis, we can improve run-time efficiency by avoiding wrappers when they are superfluous according to the over-approximation of levels given by the static analysis.

**Acknowledgements.** We thank Christian Johansen for useful interactions. The Norwegian Research Council has funded us by project *IoTSec* (no. 248113/O70).

### Authors' addresses

**First Author** University of Oslo, Oslo, Norway, farzanka@ifi.uio.no

**Second Author** University of Oslo, Oslo, Norway, olaf@ifi.uio.no

**Third Author** Chalmers University of Technology, Gothenburg, Sweden, gerardo@cse.gu.se

# Bibliography


[1] Agha, G. A. *Actors: A model of concurrent computation in distributed systems*. Tech. rep. Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.

[2] Attali, I. et al. "Secured information flow for asynchronous sequential processes". In: *Electronic Notes in Theoretical Computer Science* vol. 180, no. 1 (2007), pp. 17–34.

[3] Bae, K., Escobar, S., and Meseguer, J. "The Maude LTL LBMC Tool Tutorial". In: ().

[4] Baier, C. and Katoen, J.-P. *Principles of model checking*. MIT press, 2008.

[5] Baker Jr, H. C. and Hewitt, C. "The incremental garbage collection of processes". In: *ACM SIGART Bulletin*, no. 64 (1977), pp. 55–59.

[6] Balliu, M., Schoepe, D., and Sabelfeld, A. "We are family: Relating information-flow trackers". In: *European Symposium on Research in Computer Security*. Springer. 2017, pp. 124–145.

[7] Basin, D., Debois, S., and Hildebrandt, T. "On purpose and by necessity: compliance under the GDPR". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2018, pp. 20–37.

[8] Boer, F. D. et al. "A survey of active object languages". In: *ACM Computing Surveys* vol. 50, no. 5 (2017), p. 76.

[9] Brandauer, S. et al. "Parallel objects for multicores: A glimpse at the parallel language Encore". In: *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Vol. 9104. *Lecture Notes in Computer Science*. Springer, 2015, pp. 1–56.

[10] Bruni, R. and Meseguer, J. "Generalized rewrite theories". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2003, pp. 252–266.

[11] Byun, J.-W., Bertino, E., and Li, N. "Purpose based access control of complex data for privacy protection". In: *Proceedings of the tenth ACM symposium on Access control models and technologies*. 2005, pp. 102–110.

[12] Byun, J.-W. and Li, N. "Purpose based access control for privacy protection in relational database systems". In: *The VLDB Journal* vol. 17, no. 4 (2008), pp. 603–619.

[13] Caromel, D. and Henrio, L. *A Theory of Distributed Objects: Asynchrony-Mobility-Groups-Components*. Springer, 2005.

[14] Caromel, D., Henrio, L., and Serpette, B. P. "Asynchronous and deterministic objects". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2004, pp. 123–134.

[15] Caromel, D. et al. "ProActive: an integrated platform for programming and running applications on Grids and P2P systems". In: *Computational Methods in Science & Technology* vol. 12.1 (2006), p. 16.

[16] Cavoukian, A. "Privacy by design: origins, meaning, and prospects for assuring privacy and trust in the information era". In: *Privacy protection measures and technologies in business organizations: aspects and standards*. IGI Global, 2012, pp. 170–208.

[17] Clarke, E. M. and Emerson, E. A. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Workshop on logic of programs*. Springer. 1981, pp. 52–71.

[18] Clavel, M. et al. *All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Springer, 2007.

[19] Dahl, O.-J. and Owe, O. *Formal Methods and the RM-ODP*. Research Report 261. (Full version of a paper presented at NWPT'98: Nordic Workshop on Programming Theory, Turku). Dept. of informatics, University of Oslo, Norway, May 1998, p. 18.

[20] Dedecker, J. et al. "Ambient-Oriented Programming in AmbientTalk". In: *European Conference on Object-Oriented Programming (ECOOP'06)*. Vol. 4067. *Lecture Notes in Computer Science*. Springer, 2006, pp. 230–254.

[21] Denning, D. E. "A lattice model of secure information flow". In: *Communications of the ACM* vol. 19, no. 5 (1976), pp. 236–243.

[22] Denning, D. E. and Denning, P. J. "Certification of programs for secure information flow". In: *Communications of the ACM* vol. 20, no. 7 (1977), pp. 504–513.

[23] Dijkstra, E. W. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Commun. ACM* vol. 18, no. 8 (Aug. 1975), pp. 453–457.

[24] Din, C. C., Dovland, J., and Owe, O. "Compositional Reasoning about Shared Futures". In: *Software Engineering and Formal Methods*. Ed. by Eleftherakis, G., Hinchey, M., and Holcombe, M. Vol. 7504. LNCS. Springer, 2012, pp. 94–108.

[25] Din, C. C. and Owe, O. "A sound and complete reasoning system for asynchronous communication with shared futures". In: *Journal of Logical and Algebraic Methods in Programming* vol. 83, no. 5-6 (2014), pp. 360–383.

[26] Durán, F. et al. "All about Maude: A high-performance logical framework". In: *LNCS* vol. 4350 (2007).

[27] Fernandez-Reyes, K., Clarke, D., and McCain, D. S. "ParT: An Asynchronous Parallel Abstraction for Speculative Pipeline Computations". In: *International Conference on Coordination Languages and Models*. Springer, 2016, pp. 101–120.

[28] Flanagan, C. and Felleisen, M. "The semantics of future and an application". In: *Journal of Functional Programming* vol. 9, no. 1 (1999), pp. 1–31.

[29] Goguen, J. A. and Meseguer, J. "Security policies and security models". In: *1982 IEEE Symposium on Security and Privacy*. IEEE. 1982, pp. 11–11.

[30] Haller, P. and Odersky, M. "Scala actors: Unifying thread-based and event-based programming". In: *Theoretical Computer Science* vol. 410, no. 2-3 (2009), pp. 202–220.

[31] Halstead Jr, R. H. "Multilisp: A language for concurrent symbolic computation". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* vol. 7, no. 4 (1985), pp. 501–538.

[32] Hayati, K. and Abadi, M. "Language-based enforcement of privacy policies". In: *International Workshop on Privacy Enhancing Technologies*. Springer. 2004, pp. 302–313.

[33] Hedin, D. and Sabelfeld, A. "A Perspective on Information-Flow Control." In: *Software Safety and Security* vol. 33 (2012), pp. 319–347.

[34] Henrio, L. and Rochas, J. "Multiactive objects and their applications". In: *Logical Methods in Computer Science* vol. Volume 13, Issue 4 (Nov. 2017), pp. 1–41.

[35] Henrio, L. et al. "First Class Futures: Specification and Implementation of Update Strategies." In: *Euro-Par Workshops*. Springer, 2010, pp. 295–303.

[36] Hewitt, C., Bishop, P., and Steiger, R. "Session 8 Formalisms for Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Advance Papers of the Conference*. Vol. 3. Stanford Research Institute. 1973, p. 235.

[37] Hoare, C. A. R. "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* vol. 12, no. 10 (1969), pp. 576–580.

[38] Hähnle, R. "The abstract behavioral specification language: a tutorial introduction". In: *International Symposium on Formal Methods for Components and Objects (FMCO 2012)*. Vol. 7866. *Lecture Notes in Computer Science*. Springer, 2012, pp. 1–37.

[39] Johnsen, E. B. and Owe, O. "An asynchronous communication model for distributed concurrent objects". In: *Software & Systems Modeling* vol. 6, no. 1 (2007), pp. 39–58.

[40] Johnsen, E. B., Owe, O., and Arnestad, M. "Combining Active and Reactive Behavior in Concurrent Objects". In: *Proc. of the Norwegian Informatics Conference (NIK'03)*. Tapir, Nov. 2003, pp. 193–204.

[41] Johnsen, E. B., Owe, O., and Yu, I. C. "Creol: A type-safe object-oriented model for distributed concurrent systems". In: *Theoretical Computer Science* vol. 365, no. 1-2 (2006), pp. 23–66.

[42] Johnsen, E. B. et al. "ABS: A core language for abstract behavioral specification". In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2010, pp. 142–164.

[43] Johnsen, E. B. et al. "ABS: A core language for abstract behavioral specification". In: *Formal Methods for Components and Objects*. Vol. 6957. *Lecture Notes in Computer Science*. Springer, 2011, pp. 142–164.

[44] Johnsen, E. B. et al. "Intra-Object versus Inter-Object: Concurrency and Reasoning in Creol". In: *Electronic Notes in Theoretical Computer Science* vol. 243 (2009). Proceedings of the 2nd International Workshop on Harnessing Theories for Tool Support in Software (TTSS'08), pp. 89–103.

[45] Karami, F., Owe, O., and Ramezanifarkhani, T. "An evaluation of interaction paradigms for active objects". In: *Journal of Logical and Algebraic Methods in Programming* vol. 103 (2019), pp. 154–183.

[46] Kashyap, V., Wiedermann, B., and Hardekopf, B. "Timing-and termination-sensitive secure information flow: Exploring a new approach". In: *2011 IEEE Symposium on Security and Privacy*. IEEE. 2011, pp. 413–428.

[47] Kumar, N. N. and Shyamasundar, R. "Realizing purpose-based privacy policies succinctly via information-flow labels". In: *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*. IEEE. 2014, pp. 753–760.

[48] Magazinius, J., Phung, P. H., and Sands, D. "Safe wrappers and sane policies for self protecting Javascript". In: *Nordic Conference on Secure IT Systems*. Springer. 2010, pp. 239–255.

[49] Markey, N. "Temporal logic with past is exponentially more succinct". In: *Bulletin-European Association for Theoretical Computer Science* vol. 79 (2003), pp. 122–128.

[50] Martí-Oliet, N. and Meseguer, J. "Rewriting logic as a logical and semantic framework". In: *Electronic Notes in Theoretical Computer Science* vol. 4 (1996), pp. 190–225.

[51] Martí-Oliet, N. and Meseguer, J. "Rewriting logic: roadmap and bibliography". In: *Theoretical Computer Science* vol. 285, no. 2 (2002), pp. 121–154.

[52] Meseguer, J. "Conditional rewriting logic as a unified model of concurrency". In: *Theoretical computer science* vol. 96, no. 1 (1992), pp. 73–155.

[53] Meseguer, J. "Membership algebra as a logical framework for equational specification". In: *International Workshop on Algebraic Development Techniques*. Springer. 1997, pp. 18–61.

[54] Meseguer, J. “Rewriting logic as a semantic framework for concurrency: a progress report”. In: *International Conference on Concurrency Theory*. Springer. 1996, pp. 331–372.

[55] Meseguer, J. “Twenty years of rewriting logic”. In: *The Journal of Logic and Algebraic Programming* vol. 81, no. 7-8 (2012), pp. 721–781.

[56] Myers, A. C. and Liskov, B. “Protecting privacy using the decentralized label model”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* vol. 9, no. 4 (2000), pp. 410–442.

[57] Nair, S. K. et al. “A virtual machine based information flow control system for policy enforcement”. In: *Electronic Notes in Theoretical Computer Science* vol. 197, no. 1 (2008), pp. 3–16.

[58] Owe, O. “Verifiable Programming of Object-Oriented and Distributed Systems”. In: *From Action Systems to Distributed Systems - The Refinement Approach*. Ed. by Petre, L. and Sekerinski, E. Chapman and Hall/CRC, 2016, pp. 61–79.

[59] Owe, O. and Dahl, O. “Generator Induction in Order Sorted Algebras”. In: *Formal Aspects Comput.* vol. 3, no. 1 (1991), pp. 2–20.

[60] Owe, O. and Ramezanifarkhani, T. “Confidentiality of Interactions in Concurrent Object-Oriented Systems”. In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Vol. 10436. lncs. Cham: Springer, 2017, pp. 19–34.

[61] Owe, O. and Ramezanifarkhani, T. “Confidentiality of interactions in concurrent object-oriented systems”. In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 2017, pp. 19–34.

[62] Owe, O. and Ryl, I. *OUN: A formalism for open, object oriented, distributed systems*. Research Report 270. Dept. of informatics, University of Oslo, Norway, Aug. 1999.

[63] Owe, O. and Schneider, G. “Wrap your objects safely”. In: *Electronic Notes in Theoretical Computer Science* vol. 253, no. 1 (2009), pp. 127–143.

[64] Pettai, M. and Laud, P. “Securing the Future — An Information Flow Analysis of a Distributed OO Language”. In: *SOFSEM 2012: Theory and Practice of Computer Science*. Springer, 2012, pp. 576–587.

[65] Pnueli, A. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. ieee. 1977, pp. 46–57.

[66] Queille, J.-P. and Sifakis, J. “Specification and verification of concurrent systems in CESAR”. In: *International Symposium on programming*. Springer. 1982, pp. 337–351.

[67] Razavi, N. et al. “Sysfier: Actor-based formal verification of systemc”. In: *ACM Transactions on Embedded Computing Systems (TECS)* vol. 10, no. 2 (2010), p. 19.

[68] Regulation, G. D. P. "Regulation EU 2016/679 of the European Parliament and of the Council of 27 April 2016". In: *Official Journal of the European Union. Available at: http://ec. europa. eu/justice/data-protection/reform/files/regulation_oj_en. pdf (accessed 20 September 2017)* (2016).

[69] "Regulation 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)". In: *Official Journal of the European Union* (2016), L119:1–88.

[70] Russo, A. and Sabelfeld, A. "Dynamic vs. static flow-sensitive security analysis". In: *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE. 2010, pp. 186–199.

[71] Sabelfeld, A. and Myers, A. C. "Language-based information-flow security". In: *IEEE Journal on selected areas in communications* vol. 21, no. 1 (2003), pp. 5–19.

[72] Schneider, G. "Is Privacy by Construction Possible?" In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 471–485.

[73] Schäfer, J. and Poetzsch-Heffter, A. "JCoBox: Generalizing Active Objects to Concurrent Components". In: *ECOOP 2010–Object-Oriented Programming. Lecture Notes in Computer Science* vol. 6183 (2010), pp. 275–299.

[74] Sen, S. et al. "Bootstrapping privacy compliance in big data systems". In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 327–342.

[75] Serbanescu, V. et al. "Towards type-based optimizations in distributed applications using ABS and JAVA 8". In: *International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. Springer. 2014, pp. 103–112.

[76] Sirjani, M., Movaghar, A., and Mousavi, M. R. "Compositional Verification of an Object-Based Model for Reactive Systems". In: *Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'01), Oxford, UK*. Citeseer. 2001, pp. 114–118.

[77] Sirjani, M. et al. "Extended Rebeca: A component-based actor language with synchronous message passing". In: *Application of Concurrency to System Design, 2005. ACSD 2005. Fifth International Conference on*. IEEE. 2005, pp. 212–221.

[78] Sirjani, M. et al. "Modeling and verification of reactive systems using Rebeca". In: *Fundamenta Informaticae* vol. 63, no. 4 (2004), pp. 385–410.

[79] Soundarajan, N. "Axiomatic Semantics of Communicating Sequential Processes". In: *ACM Transactions on Programming Languages and Systems* vol. 6 (1984), pp. 646–662.

[80] Tokas, S. and Owe, O. “A formal framework for consent management”. In: *Formal Techniques for Distributed Objects, Components, and Systems: 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15–19, 2020, Proceedings 40*. Springer. 2020, pp. 169–186.

[81] Tokas, S., Owe, O., and Ramezanifarkhani, T. “Static checking of GDPR-related privacy compliance for object-oriented distributed systems”. In: *Journal of Logical and Algebraic Methods in Programming* vol. 125 (2022), p. 100733.

[82] Voigt, P. and Bussche, A. von dem. *The EU general data protection regulation (GDPR): A Practical Guide*. Springer, 2017.

[83] Winskel, G. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[84] Wyatt, D. *Akka concurrency*. Artima Incorporation, 2013.

[85] Yonezawa, A., ed. *ABCL: An Object-oriented Concurrent System*. Cambridge, MA, USA: MIT Press, 1990.

[86] Ölveczky, P. C. *Designing Reliable Distributed Systems*. Springer, 2017.

[87] Ölveczky, P. C. *Designing Reliable Distributed Systems: A Formal Methods Approach Based on Executable Modeling in Maude*. Springer, 2018.

# List of Figures

# List of Tables