Static Analysis of Unbounded Structures in Object-Oriented Programs

Proefschrift

ter verkrijging van

de graad van Doctor aan de Universiteit Leiden

op gezag van Rector Magnificus prof. mr. P.F. van der Heijden,

volgens besluit van het College voor Promoties

te verdedigen op woensdag 19 december 2012

klokke 10:00 uur

door

Immo Grabe

geboren te Kiel, Duitsland, in 1979

The work in this thesis has been carried out at the Christian-Albrechts--Universität zu Kiel, the Centrum Wiskunde & Informatica (CWI), and the Universiteit Leiden. The research was partially funded by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services*, the EU-project FP7-231620 *HATS: Highly Adaptable and Trustworthy Software using Formal Methods*, and the German-Norwegian DAAD-NWO exchange project *Avabi* (Automated validation for behavioral interfaces of asynchronous active objects).

Cover design by Immo Grabe.
Printed by: Proefschriftmaken.nl || Uitgeverij BOXPress
Published by: Uitgeverij BOXPress, Oisterwijk
ISBN: 978-90-8891-154-75

Contents

List of Figures

Chapter 1

Introduction

As indicated by the title our interest lies with the analysis of unbounded structures for object-oriented languages. In particular we address two sources of unboundedness that are commonly found in object-oriented languages: object creation, i.e. structural complexity, and multi-threading, i.e. behavioural complexity.

Construction and validation of programs requires a deep understanding of the sources of these complexities and ways to address them in the validation process.

Thesis est omnis divisa in partes tres. All parts address the aforementioned complexities in different ways and settings. Since the setting and approach presented in each section is introduced individually we restrain ourselves here to an overview of the overall thesis to motivate the individual parts.

First we present our approach to address structural complexity by an abstraction of the underlying representation of objects and object creation. In object-oriented programming languages like Java, objects can be dynamically created by the constructor methods provided by their class. Using constructors for object creation is an abstraction from the underlying representation of objects and the implementation of object creation. For practical purposes it is important to be able to specify and verify properties of objects at the abstraction level of the programming language. We give such a representation in terms of a weakest precondition calculus for abstract object creation in dynamic logic, the logic underlying the KeY theorem prover [19]. This representation allows to both specify and verify properties of objects at the abstraction level of the programming language. Based on this weakest precondition calculus we show how to symbolically execute abstract object creation in the KeY theorem prover.

The work presented in the first part was published as [10].

Second we present a technique to address behavioural complexity introduced by multi-threading. Multi-threaded programs can show complex and unexpected behaviour due to the interleaving of the activities of the individual processes or threads. An example of such an unexpected and undesired behaviour is a system reaching a deadlock, i.e. a situation in which processes are blocked due to mutual waiting. Sources of such mutual waiting can be for example locks either by explicit or implicit lock handling, e.g. monitor concept. Reasoning about locks in an object-oriented language the concept of reentrance comes into picture, i.e. a process that has acquired a lock (to an object) is free to pass the same lock several times (further method invocations on the object locked by the process). The lock can only be freed as soon as the initial method invocation (acquiring the lock) and all consecutive method invocations of the process to methods protected by the lock (e.g. at the object in case of a monitor) have terminated. This complicates the analysis of deadlock behaviour as it introduces the need to look at the call stack of the processes. We present an abtraction of multi-threaded programs (with respect to data and control flow) that allows us to detect deadlocks. Our technique allows for reentrant method calls. To the best of our knowledge this is the first automata based approach tailored to deadlock detection of the abstract control flow of method calls and returns of multithreaded reentrant programs.

The work presented in the second part was published as [43].

Third we extend a calculus to reason about active objects with futures and promises. We present an open semantics for the core of the *Creol* language including first-class futures and promises. A future acts as a proxy for, or reference to, the delayed result of a computation. As the consumer of the result can proceed its own execution until it actually needs the result, futures provide a natural, lightweight, and transparent mechanism to introduce parallelism into a language. A promise is a generalization of a future as it allows for delegation with respect to which process performs the computation. The formalization is given as a typed, imperative object calculus to facilitate the comparison with the multi-threaded concurrency model of object-oriented languages, e.g. *Java*.

We close the third part of this thesis by presenting a technique to detect deadlocks in concurrent systems of active objects. Our technique is based on a translation of the system to analyse into a P/T net and the application of a technique to detect termination in such P/T nets. We illustrate our techique by application to an Actor-like subset of the *Creol* language featuring asynchronous calls using futures as means of communication. The so-called discipline of cooperative multi-tasking within an object as found in *Creol* can lead to deadlock. Our technique can be applied to detect such deadlocks.

The work presented in the third part was published as [4] and [44].

Part I

Object Creation

Chapter 2

Abstract Object Creation in Dynamic Logic[1]

In this chapter we give a representation of a weakest precondition calculus for abstract object creation in dynamic logic, the logic underlying the KeY theorem prover. This representation allows to both specify and verify properties of objects at the abstraction level of the (object-oriented) programming language. Objects which are not (yet) created never play any role, neither in the specification nor in the verification of properties. Further, we show how to symbolically execute abstract object creation.

2.1 Introduction

In object-oriented programming languages like Java, objects can be dynamically created by the constructor methods provided by their class. Using constructors for object creation is an abstraction from the underlying representation of objects and the implementation of object creation. At the abstraction level of the programming language, objects are described as instances of their classes, i.e., the classes provide the only operations which can be performed on objects. For practical purposes it is important to be able to specify and verify properties of objects at the abstraction level of the programming language. Specification languages like the Java Modeling Language (JML) [73] and the Object Constraint Language (OCL) [83] abstract from the underlying representation of objects. In [40], a Hoare logic is presented to verify properties of

[1]The work presented in this chapter was published as [10].

an object-oriented programming language at the abstraction level of the programming language itself. This Hoare logic is based on a weakest precondition calculus for object creation which abstracts from the implementation of object creation.

In this chapter we give a representation of a weakest precondition calculus for abstract object creation in dynamic logic, the logic underlying the KeY theorem prover [19]. This representation allows to both specify and verify properties of objects at the abstraction level of the programming language. Objects which are not (yet) created never play any role, neither in the specification nor in the verification of properties.

The generalization of Hoare logic to dynamic logic is of particular interest because it allows for the specification of properties of dynamic object structures which cannot be expressed in first-order logic, like reachability. In Hoare logic such properties require quantification over (finite) sequences or recursively defined predicates in the specification language which seriously complicates both the weakest precondition calculus and the underlying logic. In dynamic logic we can restrict to first-order quantification and use the modalities to express for example reachability properties.

An interesting consequence of the abstraction level of the specification language studied in this chapter is the *dynamic scope* of the quantification over objects because it is restricted to the created objects and as such is also affected by object creation. However, we show that the standard logic of first-order quantification also applies in the presence of (object) quantifiers with a dynamic scope.

Further, we show how to symbolically execute abstract object creation in KeY. In general, symbolic execution in KeY accumulates in a simultaneous substitution of the assignments generated by a computation. This accumulation involves a pre-processing of the substitution which in general simplifies its actual application. However, we cannot simply accumulate abstract object creation because its side-effects can only be processed by the actual application of the corresponding substitution. We show how to solve this problem by the introduction of fresh logical variables which are used as temporary place holders for the newly created objects. The use of these place holders together with the fact that we can always anticipate object creation allows to symbolically execute abstract object creation.

Related work

Most formalisations of object-oriented programs, like embeddings into the logic of higher-order theorem provers PVS [100] and Isabelle [70], or dynamic logic as employed in the KeY theorem prover, use an explicit representation of objects. Object creation is then formalized in terms of the information about which objects are in fact created. Such an explicit representation of objects additionally requires an axiomatization of certain consistency requirements, e.g., the global invariant that the values of the fields of created objects only refer to created objects. These requirements pervade the correctness proofs with the basic case distinction between “to be or not to be created” and adds considerably to the length of the proofs, as we illustrate in Section 2.5.

The contribution of this chapter is the formalization of object creation in dynamic logic which abstracts from an explicit representation of objects and the corresponding implementation of object creation. Proofs in this formalization only refer to created objects and as such are not pervaded by irrelevant implementation details.

Outline

In Section 2.2 we introduce a dynamic logic for a simple WHILE-language with object creation. This language allows us to focus on object creation. We present the axiomatization of the language in terms of the sequent calculus given in Section 2.3. Please observe that this calculus can be extended to other programming constructs of existing object-oriented languages like Java as described in [21]. With the calculus at hand symbolic execution of programs is described in Section 2.4. After a discussion of the state of the art in symbolic execution with respect to object creation and a look into the expressiveness of our approach in Section 2.5 we conclude with Section 2.6.

2.2 Dynamic Logic

To focus on the abstract object creation we restrict ourselves to a simple WHILE-language as our object-oriented programming language. The language contains data of three types Object, Integer, and Boolean. In [21] Becker and Platzer present a similar dynamic logic for Java Card called ODL. ODL covers the type system of Java. Besides the type system, dynamic dispatch, side-effects of expressions, and exception handling are presented in terms of program transformations. However ODL models object creation in terms of an explicit representation of objects. To obtain a logic covering Java that follows our theory of abstract object creation this representation can be replaced by our theory or our theory can be extended analogous to [21].

2.2.1 Syntax

We assume the sets F of fields and GVar of global variables to be given. Fields are the instance variables of objects. We assume a partitioning of the global variables into a set PVar of program variables and a set LVar of logical variables. Logical variables do not change during program execution, i.e. there are no assignments to logical variables. Logical variables are used to express invariant properties and for (first-order) quantification. All fields and variables are typed. As mentioned before we restrict to the types Object, Integer, and Boolean. We omit explicit declarations. The grammar for statements and expressions of the simple WHILE-language are presented in Figure 2.1.

$$\begin{array}{lll l}
s & ::= & \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od} \mid \mathsf{if}\, e_1\, \mathsf{then}\, s_2\, \mathsf{else}\, s_3\, \mathsf{fi} \mid s_1; s_2 \mid \mathsf{skip} \mid & \\
 & & u := \mathsf{new} \mid e_1.x := e_2 \mid u := e & \text{statements} \\
e & ::= & u \mid e.x \mid \mathsf{null} \mid e_1 = e_2 \mid \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3\, \mathsf{fi} \mid f(e_1, ..., e_n) & \text{expressions}
\end{array}$$

Figure 2.1: Grammar rules for the simple WHILE-language

The statement while denotes the usual looping. Conditional branching is denoted by if–then–else. The condition for both looping and branching is given by a Boolean expression. A semicolon denotes sequential composition. By skip we denote the empty statement. Object creation is denoted by $u := \mathsf{new}$, where u is a program variable. An assignment to a program variable is denoted by $u := e$. A dot denotes dereferencing, i.e., $e_1.x := e_2$ denotes an assignment to the field x of the object referenced by e_1. For technical convenience only we do not have assignments $e.x := \mathsf{new}$. In order to separate object creation from the aliasing problem we reason about such assignments in terms of the statement $u := \mathsf{new}; e.x := u$, where u is a fresh program variable.

The expression null of type Object denotes the undefined reference. The Boolean expression $e_1 = e_2$ denotes the test for equality between the values of the expressions e_1 and e_2, e.g., e_1 and e_2 refer to the same object in case e_1 and e_2 are variables of type Object. A conditional expression is denoted by if–then–else. The function $f(e_1, ..., e_n)$ denotes an arithmetic or Boolean operation of arity n. We assume every statement and expression to be well-typed. It is important to note that object expressions, i.e., expressions of type Object, can only be compared for equality, dereferenced, or appear as argument of a conditional expression.

Formulas. Dynamic logic is a variant of *modal logic*. Different parts of a formula are evaluated in different worlds (states), which vary in the interpretation of, in our case, program variables and fields. Dynamic logic extends full first-order logic with two additional (mix-fix) operators: $\langle\,.\,\rangle\,.$ (diamond) and $[\,.\,]\,.$ (box). In both cases, the first argument is a *program* (fragment), whereas the second argument is another dynamic logic formula. A formula $\langle p\rangle\phi$ is true in a state s if execution of p terminates when started in s and results in a state where ϕ is true. As for the box-operator, a formula $[p]\phi$ is true in a state s if execution of p, when started in s, does *either* not terminate *or* results in a state where ϕ is true. In other words, the difference between the operators is the difference between total and partial correctness.[2] Dynamic logic is closed under all logical connectives.

For instance, the formula $\forall\ l.\,(\langle p\rangle\ (l = u)\ \leftrightarrow\ \langle q\rangle\ (l = u))$ states equivalence of p and q w.r.t. the program variable u.

Example 2.2.1 (Object Creation) *We give an example of a formula involving object creation: $\forall l.\langle u := \mathsf{new}\rangle\neg(u = l)$. It states that every new object indeed is new because the logical variable l ranges over all the objects that exist* before *the object creation $u := \mathsf{new}$. Consequently, after the execution of $u := \mathsf{new}$ we have that the new object is not equal to any object that already existed before, i.e., $\neg(u = l)$, when l refers to an "old" object. Note that the formula $\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)$ has a completely different meaning. In fact the formula is false (cf. Section 2.3.3). These examples illustrate a further advantage of dynamic logic over Hoare logic: the presence of explicit quantifiers in both formulas clarify the difference in meaning.*

All major program logics (Hoare logic, weakest precondition calculus, dynamic logic) have in common that the resolving of assignments requires substitutions in the formula, in one way or the other. In the KeY approach, the effect of substitutions is delayed, by having *explicit substitutions* in the logic, called *'updates'*. In this chapter, elementary updates have the form $u := \mathsf{new}$, $e_1.x := e_2$, or $u := e$. Updates are introduced to the logic via the update

[2] Just as in standard modal logic, the diamond resp. box operators quantify existentially resp. universally over states (reached by the program). In case of deterministic programs, however, the only difference between the two is whether termination is claimed or not.

modality $\{\,.\,\}\,.\,$, connecting arbitrary updates with arbitrary formulas, like in $0 < v \rightarrow \{u := v\}\, 0 < u$.

A full account of KeY style dynamic logic can be found in [20].

2.2.2 Semantics

To define the semantics of our DL we assume given an arbitrary (infinite) set O of *object identities*, with typical element o. We define null itself to be an element of O, i.e., the value of the expression null is null itself. By $\mathit{dom}(T)$ we denote the domain of values of type T, e.g., $\mathit{dom}(\text{Object}) = O$.

States. A state $\Sigma = (\sigma, \tau)$ is a pair consisting of a heap σ and an environment τ. The heap σ is a partial function such that $\sigma(o)$ for every $o \in O$, if defined, denotes the internal state of object o. That is, the value of a field x of an object o, for which $\sigma(o)$ is defined, is given by $\sigma(o)(x) \in \mathit{dom}(T)$. The domain $\mathit{dom}(\sigma)$ of objects that exist in a heap σ is given by the set of objects o for which $\sigma(o)$ is defined. In order to describe unbounded object creation we require the domain of a heap to be finite. The environment τ assigns values to the global variables. The value of a variable v is given by $\tau(v)$.

We require every state $\Sigma = (\sigma, \tau)$ to be consistent, i.e.,

- $\mathsf{null} \in \mathit{dom}(\sigma)$,
- $\sigma(o)(x) \in \mathit{dom}(\sigma)$ for every $o \in \mathit{dom}(\sigma)$ and field x of type Object,
- $\tau(v) \in \mathit{dom}(\sigma)$ for every global variable v of type Object.

In words, null is an existing object, the fields of type Object of existing objects refer to existing objects and all global variables of type Object refer to existing objects.

Semantics of Expressions and Statements. The semantics of an expression e of type T is a partial function $[\![e]\!] : \Sigma \rightharpoonup \mathit{dom}(T)$. As an example, if $[\![e]\!]$ is defined and does not evaluate to null then

$$[\![e.x]\!](\sigma, \tau) = \sigma([\![e]\!](\sigma, \tau))(x),$$

otherwise $[\![e.x]\!]$ is undefined. For a general treatment of failures we assume given a predicate $\mathit{def}(e)$ which defines the conditions under which the expression e is defined. For example, we have that $\mathit{def}(u.x) \equiv \neg(u = \mathsf{null})$.

The semantics of a statement s is a partial function $[\![s]\!] : \Sigma \rightharpoonup \Sigma$. We focus on the semantics of object creation. In order to formally describe the initialisation of newly created objects, we first introduce for each type T an initial value of type T, i.e., $\mathit{init}_{\text{Object}} = \mathsf{null}$, $\mathit{init}_{\text{Integer}} = 0$, and $\mathit{init}_{\text{Boolean}} = \mathsf{false}$. We define init to be the initial state, i.e., the state that assigns to each field x of type T its initial value init_T. For the selection of a new object we use a choice function ν on heaps to get a fresh object, i.e., $\nu(\sigma) \notin \mathit{dom}(\sigma)$.

We now define

$$[\![u := \mathsf{new}]\!](\sigma, \tau) = (\sigma[o := \mathit{init}], \tau[u := o]),$$

where $o = \nu(\sigma)$. The heap $\sigma[o := \mathit{init}]$ assigns the local state init to the new object o and the environment $\tau[u := o]$ assigns this object to the program variable u.

Semantics of Formulas. A formula ϕ in dynamic logic is valid if $\Sigma \models \phi$ holds for every consistent state Σ. For a logical variable l of type Object, we have the following semantics of universal quantification

$$(\sigma, \tau) \models \forall l.\phi \text{ iff for all } o \in \mathit{dom}(\sigma) : (\sigma, \tau[l := o]) \models \phi,$$

where the consistency of $(\sigma, \tau[l := o])$ implies that the object o exists in σ. Consequently, quantification is restricted to the existing objects. Note that null is always included in the scope of the quantification (i.e., the scope of the quantification is non-empty).

Returning to the above example, we have

$$\begin{array}{l}(\sigma, \tau) \models \forall l.\langle u := \mathsf{new}\rangle \neg(u = l) \\ \text{iff} \\ (\sigma, \tau[l := o]) \models \langle u := \mathsf{new}\rangle \neg(u = l)\end{array}$$

for all $o \in \mathit{dom}(\sigma)$. Let $o' = \nu(\sigma)$. By the semantics of the diamond modality of dynamic logic and the above semantics of object creation we conclude that

$$\begin{array}{l}(\sigma, \tau[l := o]) \models \langle u := \mathsf{new}\rangle \neg(u = l) \\ \text{iff} \\ (\sigma[o' := \mathit{init}], \tau[l := o]) \models \neg(u = l) \\ \text{iff} \\ o \neq o'\end{array}$$

Note that since $o' \notin \mathit{dom}(\sigma)$ by definition of $\nu(\sigma)$ indeed $o \neq o'$ for all $o \in \mathit{dom}(\sigma)$.

2.3 Axiomatization

In this section, we introduce a proof system for dynamic logic with object creation which abstracts from the explicit representation of objects in the semantics defined above. As a consequence the rules of the proof system are purely defined in terms of the logic itself and do not refer to the semantics. It is characteristic for dynamic logic, in contrast to Hoare logic or weakest precondition calculi, that program reasoning is fully interleaved with first-order logic reasoning, because diamond, box or update modalities can appear both outside and inside the logical connectives and quantifiers. It is therefore important to realise that in the following proof rules, ϕ, ψ and alike, match *any* formula of our logic, possibly containing programs or updates.

2.3.1 Sequent Calculus

We follow [21, 19] in presenting the proof system for dynamic logic as a sequent calculus. A sequent is a pair of sets of formulas (each formula closed for logical variables) written as $\phi_1, ..., \phi_m \vdash \psi_1, ..., \psi_n$. The intuitive meaning is that, given all of $\phi_1, ..., \phi_m$ hold, at least one of $\psi_1, ..., \psi_n$ must hold. We use capital Greek letters to denote (possibly empty) sets of formulas. For instance, by $\Gamma \vdash \phi \rightarrow \psi, \Delta$ we mean a sequent containing at least an implication formula on the right side. Sequent calculus rules always have one sequent as conclusion and zero, one or many sequents as premises:

$$\frac{\Gamma_1 \vdash \Delta_1 \ \ldots \ \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

Semantically, a rule states that the validity of all n premises implies the validity of the conclusion ("top-down"). Operationally, rules are applied bottom-up, reducing the provability of the conclusion to the provability of the premises, starting from the initial sequent to be proved. Rules with no premise close the current proof branch. In Figure 2.2 we present some of the rules dealing with propositional connectives and quantifiers (see [55] for the full set). We omit the rules for the left hand side, the rules to deal with negation and the rule to cover conditional expressions. $\phi[l/e]$ denotes standard substitution of l with e in ϕ.

When it comes to the rules dealing with programs, most of them are not sensitive to the side of the sequent and can moreover be applied to subformulas even. For instance, $\langle s_1; s_2\rangle\phi$ can be split up into $\langle s_1\rangle\langle s_2\rangle\phi$ regardless of where

$$\textsf{impRight}\quad \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \to \psi, \Delta} \qquad \textsf{andRight}\quad \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \land \psi, \Delta}$$

$$\textsf{allRight}\quad \frac{\Gamma \vdash \phi[l/c], \Delta}{\Gamma \vdash \forall l.\phi, \Delta}\ \text{with } c \text{ a new constant} \qquad \textsf{allLeft}\quad \frac{\Gamma, \forall l.\phi, \phi[l/e] \vdash \Delta}{\Gamma, \forall l.\phi \vdash \Delta}\ \text{with } e \text{ an expression}$$

$$\textsf{close}\quad \frac{}{\Gamma, \phi \vdash \phi, \Delta}$$

$$\textsf{ind}\quad \frac{\Gamma \vdash \phi[l/0], \Delta \qquad \Gamma \vdash \forall l.(\phi \to \phi[l/l+1]), \Delta}{\Gamma \vdash \forall l.\phi, \Delta}\ \text{with } l \text{ of type Integer}$$

Figure 2.2: Sequent rules - first-order logic rules

it occurs. For that we introduce the following syntax

$$\frac{\lfloor \phi' \rfloor}{\lfloor \phi \rfloor}$$

for a schema rule where the premise is constructed from the conclusion via replacing an occurrence of ϕ by ϕ'.

In Figure 2.3 we present the rules dealing with statements. The schematic modality $\langle\!\![\cdot]\!\!\rangle$ can be instantiated with both $[\cdot]$ and $\langle\cdot\rangle$, though consistently within a single rule application. The extension of these rules with the predicate *def*(e) to reason about failures is standard and therefore omitted.

Total correctness formulas of the form $\langle \mathsf{while}\ldots\rangle\phi$ are proved by first applying the induction rule ind (possibly after generalising the formula) and applying the unwind rule within the induction step. For space reasons, we omit the invariant rule dealing with formulas of the form $[\mathsf{while}\ldots]\phi$ (see [21, 20]).

$$\mathsf{split}\ \frac{\lfloor\, \langle\!\langle s_1 \rangle\!\rangle \langle\!\langle s_2 \rangle\!\rangle \phi \,\rfloor}{\lfloor\, \langle\!\langle s_1 ; s_2 \rangle\!\rangle \phi \,\rfloor} \qquad \mathsf{if}\ \frac{\lfloor\, (e \rightarrow \langle\!\langle s_1 \rangle\!\rangle \phi) \wedge (\neg e \rightarrow \langle\!\langle s_2 \rangle\!\rangle \phi) \,\rfloor}{\lfloor\, \langle\!\langle \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi} \rangle\!\rangle \phi \,\rfloor}$$

$$\mathsf{unwind}\ \frac{\lfloor\, \langle\!\langle \mathsf{if}\ e\ \mathsf{then}\ s; \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi} \rangle\!\rangle \phi \,\rfloor}{\lfloor\, \langle\!\langle \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od} \rangle\!\rangle \phi \,\rfloor}$$

$$\mathsf{assignVar}\ \frac{\lfloor\, \{u := e\} \phi \,\rfloor}{\lfloor\, \langle\!\langle u := e \rangle\!\rangle \phi \,\rfloor} \qquad \mathsf{assignField}\ \frac{\lfloor\, \{e_1.x := e_2\} \phi \,\rfloor}{\lfloor\, \langle\!\langle e_1.x := e_2 \rangle\!\rangle \phi \,\rfloor}$$

$$\mathsf{createObj}\ \frac{\lfloor\, \{u := \mathsf{new}\} \phi \,\rfloor}{\lfloor\, \langle\!\langle u := \mathsf{new} \rangle\!\rangle \phi \,\rfloor}$$

Figure 2.3: Sequent rules - dynamic logic rules

2.3.2 Application of General Updates

Updates are essentially delayed substitutions.[3] They are resolved by application to the succeeding formula, e.g., $\{u := e\}(u > 0)$ leads to $e > 0$. Update application is only allowed on formulas *not* starting with either a diamond, box or update modality. The last restriction is dropped for symbolic execution, see Section 2.4.

We now define update application on formulas in terms of a rewrite relation $\{\mathcal{U}\}\phi \rightsquigarrow \phi'$ on formulas. As a technical vehicle, we extend the update operator to expressions, such that $\{\mathcal{U}\}e$ is an expression, for all updates $\mathcal{U}$ and expressions e. Accordingly, the rewrite relation $\rightsquigarrow$ carries over to such expressions: $\{\mathcal{U}\}e \rightsquigarrow e'$.

In Figure 2.4 we define $\rightsquigarrow$ for all standard cases (see also [92, 19]). The symbol $\mathcal{U}$ matches all updates, whereas $\mathcal{U}_{nc}$ ('*n*on-*c*reating') excludes statements of the form $u := \mathsf{new}$. Furthermore, Lit is the set of literals of all types, in our context $\{\mathsf{null}, \mathsf{true}, \mathsf{false}\} \cup \{\ldots, -1, 0, 1, \ldots\}$. (Recall LVar is the set of logical variables.)

The aliasing analysis performed by the last rule is the motivation to add

[3]The benefit of delaying substitutions in the context of symbolic execution is illustrated in Section 2.4.

$$\frac{\{\mathcal{U}\}\phi_1 * \{\mathcal{U}\}\phi_2 \rightsquigarrow \phi'}{\{\mathcal{U}\}(\phi_1 * \phi_2) \rightsquigarrow \phi'} \quad \text{with } * \in \{\wedge, \vee, \rightarrow\}$$

$$\frac{\neg\{\mathcal{U}\}\phi \rightsquigarrow \phi'}{\{\mathcal{U}\}(\neg\phi) \rightsquigarrow \phi'}$$

$$\frac{Q\, l.\, \{\mathcal{U}_{nc}\}\phi \rightsquigarrow \phi'}{\{\mathcal{U}_{nc}\}(Q\, l.\, \phi) \rightsquigarrow \phi'} \quad \text{with } Q \in \{\forall, \exists\},\ l \text{ not in } \mathcal{U}_{nc}$$

$$\{\mathcal{U}\}\alpha \rightsquigarrow \alpha \quad \text{with } \alpha \in \mathrm{LVar} \cup \mathrm{Lit}$$

$$\frac{\{\mathcal{U}_{nc}\}e_1 = \{\mathcal{U}_{nc}\}e_2 \rightsquigarrow e'}{\{\mathcal{U}_{nc}\}(e_1 = e_2) \rightsquigarrow e'}$$

$$\frac{f(\{\mathcal{U}\}e_1, ..., \{\mathcal{U}\}e_n) \rightsquigarrow e'}{\{\mathcal{U}\}f(e_1, ..., e_n) \rightsquigarrow e'}$$

$$\frac{(\{u := e_1\}e_2).x \rightsquigarrow e'}{\{u := e_1\}(e_2.x) \rightsquigarrow e'}$$

$$\frac{(\{e.x := e_1\}e_2).y \rightsquigarrow e'}{\{e.x := e_1\}(e_2.y) \rightsquigarrow e'} \quad x, y \text{ different fields}$$

$$\{u_1 := e\}u_2 \rightsquigarrow u_2 \quad u_1,\ u_2 \text{ different variables}$$

$$\{u := e\}u \rightsquigarrow e$$

$$\frac{\mathsf{if}\ (\{e.x := e_1\}e_2) = e\ \mathsf{then}\ e_1\ \mathsf{else}\ (\{e.x := e_1\}e_2).x\ \mathsf{fi}\ \rightsquigarrow\ e'}{\{e.x := e_1\}(e_2.x) \rightsquigarrow e'}$$

Figure 2.4: Application of Updates - standard cases

conditional expressions to our language. Object creation of the form $u := \mathsf{new}$ is only covered as far as it behaves like any other update. The cases where object creation makes a difference are discussed separately in Section 2.3.3. The relation $\rightsquigarrow$ is defined in a big-step manner, such that updates are resolved completely in a single $\rightsquigarrow$ step.

Note that $\rightsquigarrow$ is not defined for formulas of the form $\{\mathcal{U}\}\langle s\rangle\phi$, $\{\mathcal{U}\}[s]\phi$ or $\{\mathcal{U}\}\{\mathcal{U}'\}\phi$, i.e., they are not subject to update application. We return to formulas with nested updates, like $\{\mathcal{U}\}\{\mathcal{U}'\}\phi$, in Section 2.4.

The following rule links the rewrite relation $\rightsquigarrow$ with the sequent calculus:

$$\textsf{applyUpd} \qquad \frac{\lfloor \phi' \rfloor}{\lfloor \{\mathcal{U}\}\phi \rfloor} \quad \text{with } \{\mathcal{U}\}\phi \rightsquigarrow \phi'$$

2.3.3 Contextual Application of Object Creation

To define update application on expressions $\{u := \textsf{new}\}e$, simple substitution is not sufficient, i.e., replacing u in e by some expression, because we cannot refer to the newly created object in the state prior to its creation. However, since object expressions can only be compared for equality, or dereferenced, and do not appear as arguments of any other function, we define update application by a contextual analysis of the occurrences of u in e.

We define application of $u := \textsf{new}$ inductively. Some cases are already covered in Section 2.3.2, Figure 2.4 (the rules dealing with unrestricted $\mathcal{U}$). The other cases are discussed in the following.

If u_1 and u_2 are different variables, then

$$\{u_1 := \textsf{new}\}u_2 \rightsquigarrow u_2$$

Since the fields of a newly created object are initialised we have

$$\{u := \textsf{new}\}u.x \rightsquigarrow init_T$$

where T is the type of x.

If e is neither u nor a conditional expression then

$$\frac{(\{u := \textsf{new}\}e).x \rightsquigarrow e'}{\{u := \textsf{new}\}(e.x) \rightsquigarrow e'}$$

Otherwise, if e is a conditional expression then

$$\frac{\textsf{if}\ \{u := \textsf{new}\}b\ \textsf{then}\ \{u := \textsf{new}\}(e_1.x)\ \textsf{else}\ \{u := \textsf{new}\}(e_2.x)\ \textsf{fi} \rightsquigarrow e'}{\{u := \textsf{new}\}(\textsf{if}\ b\ \textsf{then}\ e_1\ \textsf{else}\ e_2\ \textsf{fi}\ .x) \rightsquigarrow e'}$$

Note that we use here the valid equation:
$\textsf{if}\ b\ \textsf{then}\ e_1\ \textsf{else}\ e_2\ \textsf{fi}\ .x = \textsf{if}\ b\ \textsf{then}\ e_1.x\ \textsf{else}\ e_2.x\ \textsf{fi}$.

The only other possible context of u is that of an equality $e = e'$. We distinguish the following cases.

If neither e nor e' is u or a conditional expression then they cannot refer to the newly created object and we define[4]

$$\frac{(\{u := \mathsf{new}\}e) = (\{u := \mathsf{new}\}e') \leadsto e''}{\{u := \mathsf{new}\}(e = e') \leadsto e''}$$

If e is u and e' is neither u nor a conditional expression (or vice versa) then after $u := \mathsf{new}$ the expressions e and e' cannot denote the same object (because one of them refers to the newly created object and the other one refers to an already existing object) and so we define

$$\{u := \mathsf{new}\}(e = e') \leadsto \mathsf{false}$$

On the other hand if both the expressions e and e' equal u we obviously have

$$\{u := \mathsf{new}\}(e = e') \leadsto \mathsf{true}$$

If e is a conditional expression of the form $\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}$ then

$$\frac{\mathsf{if}\ \{u := \mathsf{new}\}b\ \ \mathsf{then}\ \{u := \mathsf{new}\}(e_1 = e')\ \mathsf{else}\ \{u := \mathsf{new}\}(e_2 = e')\ \mathsf{fi} \leadsto e''}{\{u := \mathsf{new}\}(e = e') \leadsto e''}$$

And similarly for $e' = e$. Note that we use here the valid equation:

$$(\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} = e') = \mathsf{if}\ b\ \mathsf{then}\ e_1 = e'\ \mathsf{else}\ e_2 = e'\ \mathsf{fi}$$

Since object expressions can only be compared for equality, dereferenced or appear as argument of a conditional expression, it is easy to see that for every boolean expression e there exists an expression e' such that $\{u := \mathsf{new}\}e \leadsto e'$.

The following lemma states the semantic correctness of the rewrite relation $\{u := \mathsf{new}\}e \leadsto e'$: The value of e' in the state *before* the assignment $u := \mathsf{new}$ equals the value of e *after* the assignment.

Lemma 2.3.1
If $\{u := \mathsf{new}\}e \leadsto e'$ *and* $[\![u := \mathsf{new}]\!](\sigma, \tau) = (\sigma', \tau')$ *then* $[\![e']\!](\sigma, \tau) = [\![e]\!](\sigma', \tau')$.

The proof of this lemma involves a further elaboration of proofs given in [16].

Now we define the rewriting of $\{u := \mathsf{new}\}\phi$, where ϕ is a first-order formula in predicate logic (which does not contain modalities). The rules for this generalization are standard.

[4]To see why the shifting inwards of $\{u := \mathsf{new}\}$ is necessary, consider the case $\{u := \mathsf{new}\}(u.x = u.x)$.

Example 2.3.2 *We present a rule for quantification as an example:*

$$\frac{(\{u := \mathsf{new}\}\phi[l/u]) \wedge \forall l.(\{u := \mathsf{new}\}\phi) \rightsquigarrow \psi}{\{u := \mathsf{new}\}\forall l.\phi \rightsquigarrow \psi}$$

where l is a logical variable. This rewrite rule takes care of the changing scope *of the quantified variable l by distinguishing the following cases: p holds for the new object is expressed by the first conjunct $\{u := \mathsf{new}\}\phi[l/u]$ which is obtained by application of the update to $\phi[l/u]$ and p holds for all 'old' objects is expressed by the second conjunct $\forall l.(\{u := \mathsf{new}\}\phi)$.*

Example 2.3.3 *As an example, we derive $\{u := \mathsf{new}\}\forall l.\neg(u = l) \rightsquigarrow \neg(\mathsf{true}) \wedge \forall l.\neg\mathsf{false}$:*

$$\dfrac{\dfrac{\dfrac{\{u := \mathsf{new}\}(u = u) \rightsquigarrow \mathsf{true}}{\{u := \mathsf{new}\}\neg(u = u) \rightsquigarrow \neg(\mathsf{true})} \qquad \dfrac{\dfrac{\{u := \mathsf{new}\}(u = l) \rightsquigarrow \mathsf{false}}{\{u := \mathsf{new}\}\neg(u = l) \rightsquigarrow \neg\mathsf{false}}}{\forall l.\{u := \mathsf{new}\}\neg(u = l) \rightsquigarrow \forall l.\neg\mathsf{false}}}{\{u := \mathsf{new}\}\neg(u = u) \wedge \forall l.\{u := \mathsf{new}\}\neg(u = l) \rightsquigarrow \neg(\mathsf{true}) \wedge \forall l.\neg\mathsf{false}}}{\{u := \mathsf{new}\}\forall l.\neg(u = l) \rightsquigarrow \neg(\mathsf{true}) \wedge \forall l.\neg\mathsf{false}}$$

The resulting formula is equivalent to false. *We use this to prove the formula $\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)$, which states that u is different from all objects existing* after *the update (including u itself), invalid. In fact we have the following derivation for $\neg\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)$.*

$$\text{notRight}\,\dfrac{\text{assignVar}\,\dfrac{\text{applyUpd}\,\dfrac{\text{andLeft}\,\dfrac{\text{notLeft}\,\dfrac{\text{closeTrue}\,\dfrac{}{\forall l.\neg\mathsf{false} \vdash \mathsf{true}}}{\neg(\mathsf{true}), \forall l.\neg\mathsf{false} \vdash}}{\neg(\mathsf{true}) \wedge \forall l.\neg\mathsf{false} \vdash}}{\{u := \mathsf{new}\}\forall l.\neg(u = l) \vdash}}{\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)) \vdash}}{\vdash \neg\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)}$$

On the other hand, we have the following derivation of

$$\forall l.\langle u := \mathsf{new}\rangle\neg(u = l)$$

which expresses in an abstract and natural way that u indeed is a new object different from objects existing before *the update.*

$$\text{allRight}\,\dfrac{\text{assignVar}\,\dfrac{\text{applyUpd}\,\dfrac{\text{notRight}\,\dfrac{\text{closeFalse}\,\dfrac{}{\mathsf{false} \vdash}}{\vdash \neg\mathsf{false}}}{\vdash \{u := \mathsf{new}\}\neg(u = c)}}{\vdash \langle u := \mathsf{new}\rangle\neg(u = c)}}{\vdash \forall l.(\langle u := \mathsf{new}\rangle\neg(u = l))}$$

The second example shows that the standard rules for quantification apply to the quantification over the existing objects.

2.4 Symbolic Execution

2.4.1 Simultaneous Updates for Symbolic State Representation

The proof system presented so far allows for classical backwards reasoning, in a weakest precondition manner. We now generalise the notion of updates, to allow for the *accumulation* of substitutions, thereby delaying their application. In particular, this can be done in a *forward manner*, giving the proofs a *symbolic execution* nature. We illustrate this principle by example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u < v \vdash u < v}\,\mathsf{close}}{u < v \vdash \{w := u \,|\, u := v \,|\, v := u\}v < u}\,\mathsf{applyUpd}}{u < v \vdash \{w := u \,|\, u := v\}\{v := w\}v < u}\,\mathsf{mergeUpd}}{u < v \vdash \{w := u \,|\, u := v\}\langle v := w\rangle v < u}\,\mathsf{assignVar}}{u < v \vdash \{w := u\}\{u := v\}\langle v := w\rangle v < u}\,\mathsf{mergeUpd}}{\dfrac{u < v \vdash \{w := u\}\langle u := v; v := w\rangle v < u}{u < v \vdash \langle w := u; u := v; v := w\rangle v < u}\,\mathsf{split,assignVar}}\,\mathsf{split,assignVar}$$

The first application of the update rule mergeUpd introduces what is called the simultaneous update $w := u \,|\, u := v$. After applying the second mergeUpd, note that the w from the inner update was turned into a u in the simultaneous update. This is achieved by *applying* the outer update to the inner one:

$$\mathsf{mergeUpd}\qquad \dfrac{\lfloor \{\mathcal{U}_1 \,|\, \ldots \,|\mathcal{U}_n \,|\mathcal{U}'\}\phi \rfloor}{\lfloor \{\mathcal{U}_1 \,|\, \ldots \,|\mathcal{U}_n\}\{U\}\phi \rfloor}\quad \text{with } \{\mathcal{U}_1 \,|\, \ldots \,|\mathcal{U}_n\}\mathcal{U} \rightsquigarrow \mathcal{U}'$$

For this, we need to extend the rewrite relation $\rightsquigarrow$ towards defining application of updates to updates:

$$\dfrac{u := \{\mathcal{U}_{nc}\}e \rightsquigarrow \mathcal{U}'}{\{\mathcal{U}_{nc}\}(u := e) \rightsquigarrow \mathcal{U}'} \qquad \dfrac{(\{\mathcal{U}_{nc}\}e_1).x := \{\mathcal{U}_{nc}\}e_2 \rightsquigarrow \mathcal{U}'}{\{\mathcal{U}_{nc}\}(e_1.x := e_2) \rightsquigarrow \mathcal{U}'}$$

What remains is the definition of the application of simultaneous updates to *expressions*. For space reasons, we will not include the full definition here, but only one interesting special case, where two left-hand sides both write the field x which is accessed in $e.x$.

$$\frac{\mathsf{if}\ ((\mathcal{U}e_2) = e)\ \mathsf{then}\ e'_2\,\mathsf{else}\ \mathsf{if}\ ((\mathcal{U}e_1) = e)\ \mathsf{then}\ e'_1\,\mathsf{else}\ \mathcal{U}(e).x\ \mathsf{fi}\ \mathsf{fi}\ \rightsquigarrow\ e'}{\mathcal{U}(e.x) \rightsquigarrow e'}$$

with $\mathcal{U} = \{e_1.x := e'_1 \,|\, e_2.x := e'_2\}$

This already illustrates two principles: a recursive alias analysis has to be performed on all left-hand sides, and moreover, in case of a clash, the rightmost update will 'win'. The latter is exactly what reflects the destructive semantics of imperative programming. Most cases are, however, much simpler. Most of the time, it is sufficient to think of an application of a simultaneous update as an application of a standard substitution (of more than one variable). For a full account on simultaneous updates, see [92].

The idea to use simultaneous updates for symbolic execution was developed in the KeY project [19], and turned out to be a powerful concept for the validation of real world (Java) programs. A simultaneous update forms a representation of the symbolic state which is reached by "executing" the program in the proof up to the current proof node. The program is "executed" in a forward manner, avoiding the backwards execution of (pure) weakest precondition calculi, thereby achieving better readability of proofs. The simultaneous update is only applied to the post-condition as a final, single step. The KeY tool uses these updates not only for verification, but also for test case generation with high code based coverage [46] and for symbolic debugging.

2.4.2 Symbolic Execution and Abstract Object Creation

A motivation to choose the setting of dynamic logic with updates is to allow for abstract object creation in symbolic execution style verification. To do so, we have to answer the question of how symbolic execution and abstract object creation can be combined. The problem is that there is no natural way of merging object creation $\{u := \mathit{new}\}$ with other updates. Consider, for instance, the following formulas, only the first of which is valid.

$$\langle u := \mathsf{new}; v := u\rangle(u = v) \qquad\qquad \langle u := \mathsf{new}; v := \mathsf{new}\rangle(u = v)$$

Symbolic execution generates the following formulas:

$$\{u := \mathsf{new}\}\{v := u\}(u = v) \qquad\qquad \{u := \mathsf{new}\}\{v := \mathsf{new}\}(u = v)$$

Merging the updates naively results in both cases in:

$$\{u := \mathsf{new} \,|\, v := \mathsf{new}\}(u = v)$$

Whichever semantics one gives to a simultaneous update with two object creations, the formula cannot be both valid and invalid.

The proposed solution is twofold: not to merge an object creation with other updates at all, but to create a second reference to the new object, to be used for merging. For this, we introduce a *fresh* auxiliary variable to store the newly created object, and generate *two* updates according to the following rule:

$$\textsf{createObj}\qquad \frac{\lfloor \{a := \textsf{new}\}\{u := a\}\phi \rfloor}{\lfloor \langle u = \textsf{new}\rangle \phi \rfloor} \qquad \text{with } a \text{ a fresh program variable}$$

The inner update $\{u := v\}$ can be merged with other updates resulting from the analysis of ϕ. The next point to address is the "disruption" of the symbolic state, caused by object creation being unable to merge with their "neighbours", thereby strictly separating state changes happening before and after object creation. The key idea to overcome this is to gradually move all object creations to the very front (as if all objects were allocated up front) and perform standard symbolic execution on the remaining updates. We achieve this by the following rule:

$$\textsf{pullCreation}\qquad \frac{\lfloor \{u := \textsf{new}\}\mathcal{U}_{nc}\phi \rfloor}{\lfloor \mathcal{U}_{nc}\{u := \textsf{new}\}\phi \rfloor} \qquad \text{with } u \text{ not appearing in } \mathcal{U}_{nc}$$

We illustrate symbolic execution with abstract object creation by an example.

$$\textsf{split, assignVar}\ \dfrac{\textsf{split, createObj}\ \dfrac{\textsf{pullCreation}\ \dfrac{\textsf{mergeUpd, assignVar}\ \dfrac{\textsf{mergeUpd}\ \dfrac{\textsf{applyUpd}\ \dfrac{\textsf{applyUpd}\ \dfrac{\textsf{notRight, closeFalse}\ \dfrac{}{\vdash \neg\textsf{false}}}{\vdash \{a := \textsf{new}\}\neg(v = a)}}{\vdash \{a := \textsf{new}\}\{u := v \,|\, v := a \,|\, w := u\}\neg(w = v)}}{\vdash \{a := \textsf{new}\}\{u := v \,|\, v := a\}\{w := u\}\neg(w = v)}}{\vdash \{a := \textsf{new}\}\{u := v\}\{v := a\}\langle w := u\rangle\neg(w = v)}}{\vdash \{u := v\}\{a := \textsf{new}\}\{v := a\}\langle w := u\rangle\neg(w = v)}}{\vdash \{u := v\}\langle v := \textsf{new}; w := u\rangle\neg(w = v)}}{\vdash \langle u := v; v := \textsf{new}; w := u\rangle\neg(w = v)}$$

2.5 Discussion

2.5.1 Object Creation vs. Object Activation

Proof systems for object-oriented languages(cf. [2]) usually achieve the uniqueness of objects via an injective mapping, here called obj, from the natural numbers to object identities. Only the object identities $\mathsf{obj}(i)$ up to a maximum index i are considered to stand for actually created objects. In each state, the successor of this maximum index is stored in a ghost variable, here called next. (In case of Java, next would be a `static` field, for each class). Object creation increases the value of next, which conceptually is more an activation than a creation. Quantifiers cover the entire co-domain of obj, including "not yet created" objects. In order to restrict a certain property ϕ to the "created" objects, the following pattern is used: $\forall l.(\psi \to \phi)$, where ψ restricts to the created objects. Formulas of the form $\exists n.\,(n < \mathsf{next} \land \mathsf{obj}(n) = l)$ are the approach taken in ODL [21]. To avoid the extra quantifier, ghost instance variable of boolean type, here called $\mathsf{created}$, can be used to indicate for each object whether or not it has already been "created", see [20]. In this case we set the $\mathsf{created}$ status of the "new" object (identified by next) and increase next. The assertion $\forall n.(\mathsf{obj}(n).\mathsf{created} \leftrightarrow n < \mathsf{next})$ retains the relation between the $\mathsf{created}$ status and the object counter next on the level of the proofs. In both case, we need further assertions to state that fields of created objects always refer to created objects.

To state in this setting that a new object indeed is new we need to argument the formula introduced in Section 2.3, i.e. $\forall l.\,(l.\mathsf{created} \to \langle u := \mathsf{new}\rangle \neg(u = l))$. In fact the formula in Section 2.3 is not valid in this setting. An object activation style proof of this is given in Figure 2.5 (abbreviating $\mathsf{created}$ by cr). Many steps in this proof are caused by the particular details of the explicit representation of objects and the simulation of object creation by object activation.

2.5.2 Expressiveness

Many interesting properties of dynamic object structures, like reachability in dynamic linked data structures, cannot be expressed in first-order predicate logic. There are approaches to simulate reachability by an overapproximation of the reachable states [74]. In first-order dynamic logic however we can use the modalities to express such properties. For example, if a linked list is given in terms of a field *next* and the data is stored in a field *data* then the following formula in dynamic logic states that the object denoted by v is reachable from

$$
\mathsf{allRight}\,\dfrac{\mathsf{impRight}\,\dfrac{\mathsf{createObj}\,\dfrac{\mathsf{applyUpd}\,\dfrac{\mathsf{notRight}\,\dfrac{\mathsf{assumption(1)}\,\dfrac{\mathsf{allLeft}\,\dfrac{\mathsf{(2\ rules)}\,\dfrac{\mathsf{notLeft}\,\dfrac{\mathsf{equality}\,\dfrac{\mathsf{close}\,\dfrac{}{c.\mathsf{cr}, \mathsf{obj}(\mathsf{next})=c \vdash c.\mathsf{cr}}}{c.\mathsf{cr}, \mathsf{obj}(\mathsf{next})=c \vdash \mathsf{obj}(\mathsf{next}).\mathsf{cr}}}{\neg\mathsf{obj}(\mathsf{next}).\mathsf{cr}, c.\mathsf{cr}, \mathsf{obj}(\mathsf{next})=c \vdash}}{(\mathsf{obj}(\mathsf{next}).\mathsf{cr} \leftrightarrow \mathsf{next} < \mathsf{next}), c.\mathsf{cr}, \mathsf{obj}(\mathsf{next})=c \vdash}}{\forall n.(\mathsf{obj}(n).\mathsf{cr} \leftrightarrow n < \mathsf{next}), c.\mathsf{cr}, \mathsf{obj}(\mathsf{next})=c \vdash}}{c.\mathsf{cr}, \mathsf{obj}(\mathsf{next})=c \vdash}}{c.\mathsf{cr} \vdash \neg(\mathsf{obj}(\mathsf{next})=c)}}{c.\mathsf{cr} \vdash \{u{:=}\mathsf{obj}(\mathsf{next}) \mid \mathsf{obj}(\mathsf{next}).\mathsf{cr}{:=}\mathsf{true} \mid \mathsf{next}{:=}\mathsf{next}{+}1\}\neg(u=c)}}{c.\mathsf{cr} \vdash \langle u := \mathsf{new}\rangle\neg(u=c)}}{\vdash c.\mathsf{cr} \to \langle u := \mathsf{new}\rangle\neg(u=c)}}{\vdash \forall l.\,(l.\mathsf{cr} \to \langle u := \mathsf{new}\rangle\neg(u=l))}
$$

Figure 2.5: Object activation style proof

the object denoted by u:

$$\langle \mathsf{while}\ u \neq v\ \mathsf{do}\ u := u.next\ \mathsf{od}\rangle(\mathsf{true})$$

Note that in DL such formulas can be used to express properties themselves.

2.6 Conclusion

In this chapter we gave a representation of a weakest precondition calculus for abstract object creation in dynamic logic and the KeY theorem prover. Abstract object creation is formalized in terms of an inductively defined rewrite relation. The standard sequent calculus for dynamic logic is extended with a schema rule which allows to substitute formulas in sequents and thus provides a general mechanism to import for example specific rewrite relations. The resulting logic abstracts from an explicit representation of objects and the corresponding implementation of object creation. As such it abstracts from irrelevant implementation details which in general complicate proofs. Moreover, it treats the dynamic scope of quantified object variables in a standard manner. Finally, we have shown how to symbolically execute abstract object creation in KeY.

Part II

Multi-threading

Chapter 3

Deadlock Detection for Reentrant Call-graphs[1]

In this chapter we investigate the synchronization of multithreaded call graphs with reentrance similar to call graphs in Java programs. We model the individual threads as Visibly Pushdown Automata (VPA) and analyse the reachability of a state in the product automaton by means of a Context Free Language (CFL) which captures the synchronized interleaving of threads. We apply this CFL-reachability analysis to detect deadlock.

3.1 Introduction

Due to behavioural complexity formal methods are needed when it comes to reasoning about programs. This is particularly true for concurrent (or multithreaded) programs. Such programs in general involve the synchronization of different processes (or threads) which may lead to undesirable deadlock situations.

A group of activities competing for a number of resources can block each other if each of them holds resources another one needs. A typical example of such a resource is exclusive access to a part of the system, i.e. a class or object, guarded by a lock. Modern programming languages like Java opt for implicit lock handling, i.e. instead of explicitly grabbing a lock via the execution of a lock operation a region is declared to be subject to a lock and the lock handling is done by the execution platform rather than the programmer.

[1]The work presented in this chapter was published as [43].

These languages also allow for reentrance, i.e. a thread is allowed to enter each region guarded by a lock several times if it holds the lock. In such a setting the number of times a thread has entered a lock-guarded region has to be counted to decide when to release the lock again. There are techniques, e.g. preemption or global lock orders, to address the problem of deadlock in such a setting. But these techniques do not provide a general solution to the deadlock problem, i.e. most concurrent programming languages have to deal with deadlocks and how to avoid them on the programming level.

The combination of multithreading, implicit lock handling, and reentrance makes the detection of deadlocks hard. This explains the need for methods and tools to do automatic deadlock analysis.

Contribution In order to develop an automated method for deadlock detection applicable to Java-like languages we abstract from data and focus on the control flow of method calls and returns. The unsynchronized interleaving of a finite number of reentrant (abstract) threads is naturally modelled as a multistack Visibly Pushdown Automaton [14]. In order to analyse the synchronization between threads we apply Context-Free-Language(CFL)-reachability as introduced in [90] to the underlying finite state automaton. Information about the ownership of the locks is included in the CFL to model synchronized sequences of calls and returns and to identify deadlock states.

In general however CFLs are not closed under arbitrary interleavings. We resolve this lack of expressive power of CFL languages in this particular setting by showing that for every (synchronized) interleaving there exists a *rescheduling* which does not affect the synchronization and is included in the CFL language. In fact, the CFL language only restricts the scheduling of the returns and we can anticipate returns of synchronised method calls without affecting the synchronization.

To the best of our knowledge this is the first automata based approach tailored to deadlock detection of the abstract control flow of method calls and returns of multithreaded reentrant programs. A sketch of an implementation is described in the concluding section. In this implementation the programmer only needs to indicate a finite number of threads, i.e., for each thread class the number of threads involved in the deadlock analysis.

Related Work In [91] Rinard gives an overview of recent techniques to analyse multithreaded programs. Deadlock detection is only covered for languages communicating via pairwise rendezvous. Ramalingam (see [89]) has shown that the analysis of synchronization problems is not decidable even for only two threads if a CCS-style pairwise rendezvous is used to synchronize among the threads.

In [68] Kahlon et al. give a compositional method to reason about programs synchronizing via locks based on Visibly Pushdown Automata. Visibly Pushdown Automata are a kind of pushdown automata tailored to the generation of nested words reflecting the structure of traces generated by languages with nested call and return structures. The languages generated by these automata are closed under intersection. The result from [89] is generalized by showing that the reachability problem is not decidable for two threads communicating via non-nested locks. The language presented is non-reentrant and uses explicit acquire and release primitives. The automata are extended by so called acquisition histories to record relevant locking information to synchronize the threads. These acquisition histories can be used together with the explicit acquire primitives to identify deadlock situations. As soon as reentrance is allowed the setting gets more complicated. Due to reentrance the number of calls to synchronised methods has to be counted to decide whether or not to release a lock. Note that Java provides a nested call and return structure (with respect to one thread) which implies a nested acquire and release of locks.

Kidd et al. [69] introduce a technique called language strength reduction to reduce reentrant locks to non-reentrant locks in the context of Visibly Pushdown Languages. They check for atomic-set serializability violations, i.e. an illegal data access pattern. Due to this goal they take data into consideration. They create a CFL for each thread and for each lock. These languages are approximated by regular languages. Additionally a language describing a violation of a data access pattern is defined and the intersection of all languages is checked for emptiness. Up to our understanding there is no natural way to express a deadlock in this setting.

Lammich and Müller–Olm [72] present a model that can deal with thread creation and reentrant monitors. Their analysis is also focused on atomic-set serializability violations. Their approach is based on a fixpoint construction. They also use acquisition histories but only for synchronization purposes again. To reduce the number of executions to analyse they reduce the executions to a restricted subset involving the notion of a macrostep, i.e. a number of steps by one thread such that the stack only grows that is there are at most new locks taken after a macrostep but none are freed. In general, their analysis answers whether a given program location and a stack of method calls can be reached by a computation. However solutions to this reachability problem do not solve the more abstract problem of checking the reachability of a deadlock configuration.

In [34] Carotenuto et al. introduce Visibly Pushdown Automata with a finite number of stacks. The languages generated by these automata are also closed under intersection. However the emptiness problem is not decidable for these automata.

A variety of other synchronization and communication mechanisms in concurrent programs with recursion have been studied (cf. for example [24], [35]). In [25] Bouajjani et al. present a formalism to compute abstractions of multithreaded call-graphs.

Outline This chapter is organized as follows. We start with a section on the syntax and semantics of synchronised multithreaded programs. In Section 3.3 we introduce Thread Automata to model the individual threads. Based on Thread Automata we introduce a technique based on CFL-reachability for the analysis of the product automaton in Section 3.4. In Section 3.5 we prove soundness and completeness of our method. We conclude in Section 3.6.

3.2 Synchronized Multithreaded Programs

We abstract from data, which includes object identities. In our setting locks are bound to classes. A system consists of a finite number of given classes and a finite number of given threads synchronizing via locks.

3.2.1 Syntax

We assume a given set of method names M with typical element m. Methods are specified by the following regular expressions. Here τ denotes an internal step and m denotes a call.

$$r ::= \tau \mid m \mid r;r \mid r+r \mid r^*$$

Figure 3.1: Grammar rule for the simple multithreaded language

We denote by M_s the synchronised methods and by M_u the unsynchronised ones. Every method is either synchronised or not:

$$M = M_s \cup M_u \text{ and } M_s \cap M_u = \emptyset.$$

We assume a given set D of method definitions. A method definition consists of a method name m and a method body given by a regular expression $m ::= r$. Furthermore we assume a finite partitioning C of the method names into classes with typical element c.

For every class c, we denote by

- M^c its methods,
- M^c_s its synchronised methods,
- M^c_u its unsynchronised methods.

We assume that every method belongs to exactly one class.

The set of method definitions D and the set of classes C define a program P. The behaviour of a program is defined in terms of a given set of threads T with typical element t. Each thread t has an initial (run) method denoted by run(t).

3.2.2 Operational Semantics

The operational semantics of a multithreaded program is described by a labelled transition relation between configurations Θ which consist of pairs (t, θ), where θ is a stack of labelled expressions $m@r$. We require Θ to contain for each thread $t \in T$ at most one such pair. The label $m@r$ indicates that r is the continuation of the execution of the body of method m. We record the name of the method to formalize synchronization as described below. The label at the top of the stack represents the method currently executed by the thread. By $\theta \cdot m@r$ we denote the result of pushing the label $m@r$ unto the stack θ.

To describe operationally the return of a method we extend the syntax of expressions by return expression ret. We identify the method body r in a declaration $m ::= r$ with $r; ret$.

Method Calls We have the following transition for unsynchronised methods:

A call of the method m' thus pushes the corresponding label on the stack. Note that upon return of m' the execution of m continues with r.

For synchronised methods we additionally require that no other thread is executing a synchronised method subject to the same lock:

unsync
$$\Theta \cup \{(t, \theta \cdot m@m'; r)\} \to \Theta \cup \{(t, \theta \cdot m@r \cdot m'@r')\}$$
with $m' ::= r' \in D$ and $m' \in M_u^c$ for some class c.

sync
$$\Theta \cup \{(t, \theta \cdot m@m'; r)\} \to \Theta \cup \{(t, \theta \cdot m@r \cdot m'@r')\}$$
with $m' ::= r' \in D$, $m' \in M_s^c$ for some class c, and there does not exist $(t', \theta') \in \Theta$ such that $t \neq t'$ and θ' contains a continuation $m''@r''$ of method $m'' \in M_s^c$.

ret
$$\Theta \cup \{(t, \theta \cdot m@ret)\} \to \Theta \cup \{(t, \theta)\}$$

Figure 3.2: Operational rules

Return Returning from a method is described by The top of the stack thus is simply popped upon return.

The rules for the choice and iteration operators are obtained by a straightforward lifting of the corresponding transitions for regular expressions as described in the next section.

The above transition relation maintains the following synchronization invariant:

Corollary 3.2.1 *For every class c there is at most one* $(t, \theta) \in \Theta$ *such that* θ *contains a continuation* $m@r$ *of a synchronised method* $m \in M_s^c$.

This characterization of threads and locks can be modelled in a straightforward manner as a multistack pushdown automaton with counters for each class. Reachability is not decidable in this general setting. Therefore we model the system differently. Each thread is modelled as a Visibly Pushdown Automata (VPA, for short). We show that the product of these automata are amenable to analysis via a technique based on CFL-reachability. We give a grammar to steer this analysis.

3.3 Thread Automata

In this section we model and analyse the operational semantics of multithreaded programs described above in terms of thread automata. For each thread t a Thread Automaton TA(t) is defined as a VPA, in terms of a *call* alphabet $\Sigma_{\text{call}}^t = \{t_m \mid m \in M\}$ and a *return* alphabet $\Sigma_{ret}^t = \{t_{ret}\}$. By Σ^t we denote the visible alphabet $\Sigma^t_{\text{call}} \cup \Sigma^t_{ret}$ of thread t. A call of method m by thread t is indicated by t_m. The return of thread t from a method call is indicated by t_{ret}. The idea is that the call alphabet generates push operations, whereas the return alphabet generates pop operations. For each thread t its local alphabet is defined by $\{\tau\}$, used to describe internal steps.

States

The set of states of TA(t) is the set R_t of regular expressions reachable from the run method run(t). Here reachability is defined in terms of the following standard transition relation describing the behaviour of (regular) expressions:

- $m; r \to r$
- $r_1 + r_2; r \to r_i; r \quad \text{for } i \in \{1, 2\}$
- $r^*; r' \to r; r^*; r'$
- $r^*; r' \to r'$

Transitions

The external transitions of TA(t) are of the form (r, a, r', s), where r and r' are states as introduced above, a is an action of the visible alphabet of t, and s a stack symbol. The stack alphabet Γ^t of a Thread Automaton TA(t) is given by the set $\{t_r \mid t \in T,\ r \in R_t\}$, where the regular expression r denotes the return "address" of t. Method calls push a stack symbol upon the stack. This symbol encodes the location to return to later. Method returns pop a symbol from the stack. The location to return to can be derived from this symbol.

Internal transitions are of the form (r, τ, r') with r and r' states in terms of regular expressions, and τ to denote the internal step.

Method call For every state $m; r'$ we have the transition $(m; r', t_m, r, t_{r'})$, where $m ::= r \in D$. This transition models a move of control from state $m; r'$ to state r and a push of token $t_{r'}$ on the stack when reading t_m . The states encode the actual code to execute whereas the stack symbol encodes the location to return to when the method call terminates, i.e. a return is received.

Return For every state r, returning from a method is described by the transition (ret, t_{ret}, r, t_r) which models a move of control from state *ret* to state r and a pop of token t_r from the stack when reading t_{ret}. For each caller of the method a *return* transition exists. The location of return to is determinate by the token popped of the stack. Because the return location being determined by the stack symbol an unspecific return action t_{ret} is sufficient.

Internal transitions The choice construct is described by the transitions $(r_1 + r_2; r, \tau, r_i; r)$ for $i \in \{1, 2\}$ and iteration is described by a transition modelling looping $(r^*; r', \tau, r; r^*; r')$ and a transition modelling termination $(r^*; r', \tau, r')$. Note that internal transitions do not involve an operation on the stack.

Unsynchronised Product

We model the system by the product of the above automata for the individual threads. This automaton does not take synchronization between the individual threads into account. We add this synchronization by means of a grammar in Section 3.4.

Let $T = \{t^1, \ldots, t^n\}$. By $\text{TA}(T)$ we denote the *unsynchronised* product of the automata $\text{TA}(t^i)$. This product is described by a multistack VPA with call alphabet $\Sigma_{\text{call}} = \{t_m \mid t \in T,\ m \in M\}$, return alphabet $\Sigma_{\text{return}} = \{t_{ret} \mid t \in T\}$ and for each thread t a stack over the alphabet Γ^t. We denote by q_0 the initial state $q_0 = \langle \text{run}(t_1), \ldots, \text{run}(t_n) \rangle$.

States

The states of the product automaton are of the form $\langle r_1, \ldots, r_n \rangle$ where r_i denotes the state of t^i.

Transitions

We lift the transitions of the individual threads to transitions of the product in the obvious manner. Note that this lifting still does not provide any synchronization between the threads.

Reachability

Similar to the operational semantics for the definition of reachability in $\text{TA}(T)$ we give a declarative characterization of the synchronization between threads in terms of arbitrary sequences of calls and returns.

This characterization involves the following language theoretic properties:

- Calls and returns in a sequence are *matched* according to formal language theory, i.e. a bracketed grammar.
- A call without a matching return is called *pending*.
- A return without a matching call is called *pending*.
- A sequence is *well-formed* if it does not contain any pending returns.

Note that the words generated by the unsynchronised product are already well-formed.

Now we define synchronised sequences of calls and returns:
A sequence is called synchronised if for each call t_m to a synchronised method m ($m \in M_s^c$) by thread t there exists no pending call $t'_{m'}$ to a synchronised method m' of c by a thread $t' \neq t$ in the prefix of the sequence up to t_m.

We conclude this section with the definition of reachability and a definition of deadlock freedom in TA(T).

A state $q = \langle r_1, \ldots, r_n \rangle$ of TA(T) is *reachable* in TA(T) if there exists a computation in TA(T)

$$(q_0, \{\perp\}^n) \xrightarrow{W} (q, \bar{\theta})$$

for a synchronised sequence of calls and returns W and a tuple of stacks $\bar{\theta} = \langle \theta_1, \ldots, \theta_n \rangle$. Where $\perp$ denotes the empty stack.

This notion of reachability of a state does not provide enough information for deadlock detection. Therefore we extend the definition in the obvious manner to configurations: A configuration $(q, \bar{\theta})$ of TA(T) is *reachable* in TA(T) if there exists a computation in TA(T)

$$(q_0, \{\perp\}^n) \xrightarrow{W} (q, \bar{\theta})$$

for a synchronised sequence. Furthermore a configuration $(q, \bar{\theta})$ is a deadlock configuration iff $(q, \bar{\theta}) \not\rightarrow$, which indicates there is no transition possible, and at least one thread is not yet terminated, i.e. there exists an i such that $r_i \neq \mathit{ret}$ or $\theta_i \neq \perp$.

Finally we define the automaton TA(T) to be deadlock free iff there does not exist a reachable deadlock configuration.

3.4 CFL-Reachability

For the proof of the decidability of the reachability problem and deadlock freedom we apply CFL-reachability to the finite state automaton FA(T) embodied in TA(T). We first focus on the unsynchronised product and introduce synchronization later.

CFL-Modelling of Unsynchronised Interleavings

The the finite state automaton FA(T) contains all internal transitions (q, τ, q') of TA(T). To model the push and pop transitions of TA(T) we introduce the set of actions

$$\Sigma = \{t_r^m, t_r \mid t \in T,\ m \in M,\ r \in R\}$$

where t_r^m denotes a call of m by t with return expression r and t_r indicates that t returns to the regular expression r. We then model the transitions (q, t_m, q', t_r) and (q, t_{ret}, q', t_r) in TA(T) by (q, t_r^m, q') and (q, t_r, q'), respectively.

In order to compensate for the loss of information we introduce next for each thread t the following context free grammar which describes the structure of recursive call/return sequences.

$$\begin{array}{lcl} S^t & ::= & \epsilon \mid B^t \mid t_r^m S^t \mid S^t S^t \\ B^t & ::= & \epsilon \mid t_r^m r^t \mid B^t B^t \\ r^t & ::= & B^t t_r \end{array}$$

Sequences generated by the non-terminal S^t can contain pending calls, whereas sequences generated by B^t do not contain pending calls. Sequences generated by the non-terminal r^t $(r \in R_t)$ describe a return from a method call to the expression r. In these sequences the call itself does not appear, e.g., these sequences contain a return t_r without a matching call. Note that the non-terminal r^t should be distinguished from the corresponding terminal t_r.

Starting with S^t or B^t the grammar produces well-formed sequences.

We lift this grammar to the definition of another CFL grammar describing the *unsynchronised* interleavings of the individual threads. The non-terminals of this grammar are sets G, where G contains for each thread t one of its non-terminals S^t, B^t, and r^t. The above rules are lifted to this grammar as follows.

$$\begin{array}{lcll} G & ::= & \epsilon & (G \subseteq \{S^t, B^t \mid t \in T\}) \\ G \cup \{S^t\} & ::= & G \cup \{B^t\} \ \mid \ t_r^m \, G \cup \{S^t\} & \\ G \cup \{B^t\} & ::= & t_r^m \, G \cup \{r^t\} & \\ G \cup \{r^t\} & ::= & G \cup \{B^t\} \, t_r & \\ G_1 \circ G_2 & ::= & G_1 \, G_2 & \end{array}$$

where the composition $G_1 \circ G_2$ contains for every thread a non-terminal U^t for which there exist a rule $U^t ::= V_1^t V_2^t$, with V_1^t in G_1 and V_2^t in G_2. Note that only sets G which contain for each thread t either S^t or B^t can be split (in other words, the non-terminal r^t cannot be split). Note also that the non-terminal r^t cannot be generated by a split.

We denote by $G_0 = \{S^t \mid t \in T\}$ the initial configuration of a derivation.

Note that not all possible interleavings can be derived by this grammar (see the following example). But for any possible interleaving an equivalent one (with respect to synchronization) exists which can be derived by the grammar. Since the non-terminal r^t can not be split the location of a method return is restricted. This does not affect the reachability or deadlock analysis. The method returns can be shuffled within certain limits (a return can be brought forward ignoring steps of other threads and can be delayed by steps of other threads on other locks). This holds also for the synchronised case as we show later. In the synchronized case this property is ensured by the requirements with respect to the lock sets.

Example 3.4.1 *We give an example of a sequence that can not be derived directly. The sequence $t_r^m, t'^{m'}_{r'}, t_r, t'^{m}_{r''}, t'_{r''}, t'_{r'}$ with $m \in M_s^c$ and $m' \in M_s^{c' \neq c}$. It is not possible to find a direct derivation for $G_0 \Rightarrow^* t_r^m, t'^{m'}_{r'}, t_r, t'^{m}_{r''}, t'_{r''}, t'_{r'}$. Since the projection on t resp. t' contains a matching return for every call it can only be derived by a rule starting from B^t resp. $B^{t'}$. We have to start with B^t to get the t_r^m in the front position of the sequence. The next step has to be a $B^{t'}$ step to get $t'^{m'}_{r'}$ to the second position. Now $G = \{r^t\} \cup \{r'^{t'}\}$. The next step has to be a $r'^{t'}$ to get $t'_{r'}$ to the end of the sequence. Now we get $G = \{r^t\} \cup \{B^{t'}\}$. Here we are stuck. To get the t_r in front of the $t'^{m}_{r''}, t'_{r''}$ we could only use the composition rule but this one is forbidden for Gs containing a r^t. Instead we can derive $t_r^m, t_r, t'^{m'}_{r'}, t'^{m}_{r''}, t'_{r''}, t'_{r'}$. By reordering the returns we can get the original sequence..*

According to the technique of CFL-reachability we define inductively transitions of the form (q, G, q'), where q and q' are states of FA(T) and G is a set of non-terminals. Such a transition indicates that q' is reachable from q by a sequence generated by G.

- For every rule $G ::= \epsilon$ and state q we add a transition (q, G, q).
- For transitions (q, τ, q') and (q', G, q'') we add a transition (q, G, q''). Similarly, for transitions (q, G, q') and (q', τ, q'') we add a transition (q, G, q'').
- Given a transition (q, G, q'), an application of a rule $G' ::= G$ generates a transition (q, G', q).
- Given transitions (q_0, t_r^m, q) and (q, G, q_1), an application of a rule $G' ::= t_r^m\ G$ generates a transition (q_0, G', q_1).
- Given transitions (q_0, G, q) and (q, t_r, q_1), an application of a rule $G' ::= G\ t_r$ generates a transition (q_0, G', q_1).
- Given transitions (q_0, G_1, q) and (q, G_2, q_1), an application of rule $G_1 \circ G_2 ::= G_1 G_2$ generates a transition $(q_0, G_1 \circ G_2, q_1)$.

Reachability of a state q in FA(T) from the initial state q_0 by a word $G_0 \Rightarrow^* W$ then can be decided by checking the existence of a transition (q_0, G_0, q).

CFL-Modelling of Synchronised Interleavings

We now extend the above grammar for unsynchronised interleavings of threads with input/output information about the locks. This information is represented by pairs (I, L), where $I, L \subseteq T \times C$. The set of locks I are taken by some threads at the beginning of a derivation (step), whereas L is the set of locks that are taken by some threads at the end of a derivation (step). We denote an element of $T \times C$ by t_c which indicates that t holds the lock of class c. We implicitly restrict to subsets of $T \times C$ where for each class c at most one thread holds its lock. The non-terminals of this new grammar are annotated sets $(I, L) : G$, where $I \subseteq L$ and G contains for each thread t one of its non-terminals S^t, B^t, and r^t.

We have the following rules (here $I_c = \{t \in T \mid\ t_c \in I\}$ and

$L_c = \{t \in T \mid t_c \in L\})$.

$$
\begin{array}{llll}
(I,I){:}G & ::= & \epsilon & (G\subseteq\{S^t,B^t|t\in T\}) \\
(I,L){:}G\cup\{S^t\} & ::= & (I,L){:}G\cup\{B^t\} & \\
 & | & t_r^m\ (I,L){:}G\cup\{S^t\} & (m \notin M_s^c \text{ or } t_c \in I \cap L) \\
 & | & t_r^m\ (I\cup\{t_c\},L){:}G\cup\{S^t\} & (m\in M_s^c,\ I_c{=}\emptyset,\ t_c{\in}L) \\
(I,L){:}G\cup\{B^t\} & ::= & t_r^m\ (I,L){:}G\cup\{r^t\} & (m \notin M_s^c \text{ or } t_c \in I \cap L) \\
 & | & t_r^m\ (I\cup\{t_c\},L\cup\{t_c\}){:}G\cup\{r^t\} & (m \in M_s^c,\ I_c = L_c = \emptyset) \\
(I,L){:}G\cup\{r^t\} & ::= & (I,L){:}G\cup\{B^t\}\ t_r & \\
(I,L){:}G_1{\circ}G_2 & ::= & (I,L'){:}G_1\ (L',L){:}G_2 &
\end{array}
$$

The above grammar generates synchronised sequences. The conditions of the rules reflect in a natural manner the locking mechanism. To characterize the language generated by the above grammar we denote for a sequence of calls and returns W the set of locks still taken at the end of W by Lock(W): $t_c \in \text{Lock}(W)$ iff there exists a pending call to a method m by thread t with $m \in M_s^c$.

Theorem 3.4.2 *For every sequence W generated by $(I, L) : G$ we have the following properties:*

- *W is synchronised*
- *$t_c \in L$ iff $t_c \in I \cup \text{Lock}(W)$.*

Proof: The theorem is proven by induction on the length of the derivation of W. Details of the proof can be found in appendix A.1. □

To check reachability we add inductively transitions $(q, \alpha : G, q')$ to FA(T) analogous to the unsynchronised case above.

3.5 Soundness and Completeness of CFL-Reachability

Soundness and completeness of our method follows from the general technique of CFL-reachability and the following properties of our specific grammars together with the properties for sequences generated by the grammar established in Theorem 3.4.2.

We define an equivalence relation $W' \approx W$ as follows: For every thread t

- the projection of W' on t equals that of W on t

- $\mathrm{Lock}(W') = \mathrm{Lock}(W)$.

Lemma 3.5.1 *For every well-formed synchronised sequence W there exists a well-formed synchronised sequence W' such that $G_0 \Rightarrow^* W'$ with $G_0 = \{S^t \mid t \in T\}$ and $W' \approx W$.*

Proof: The lemma is proven by induction on the length of the word W. Details of the proof can be found in appendix A.2. □

We extend the notion of a synchronised sequence to a sequence synchronised with respect to a lock set I. A sequence W is synchronised with respect to I if for each $t_c \in I$ W does not contain any calls or returns of a thread $t' \neq t$ to a synchronised method of class c.

Lemma 3.5.2 *If $G_0 \Rightarrow^* W$ with W synchronised then $(\emptyset, \mathrm{Lock}(W)) : G_0 \Rightarrow^* W$.*

Proof: Instead of proving the lemma directly we prove a more general statement: If $G_0 \Rightarrow^* W$ with W synchronised with respect to I, then $(I, I \cup \mathrm{Lock}(W)) : G_0 \Rightarrow^* W$.

The statement is proven by induction on the length of the derivation $G_0 \Rightarrow^* W$. Details of the proof can be found in appendix A.3. □

Theorem 3.5.3 *The reachability problem of* $\mathrm{TA}(T)$ *is decidable.*

Our method for checking reachability of a state q in $\mathrm{TA}(T)$ consists of checking the existence of a transition $(q_0, (\emptyset, L) : G_0, q)$ in $\mathrm{FA}(T)$.

Decidability follows from soundness and completeness proven above.

Theorem 3.5.4 *The problem of deadlock freedom of* $\mathrm{TA}(T)$ *is decidable.*

In this case our method consists of checking reachability of $(q_0, (\emptyset, L) : G_0, q)$ for some state q for which there exists a subset of threads $T' \subseteq T$ such that in q each thread $t \in T'$ is about to execute a synchronised method $m \in M_s^c$ of a class c the lock of which is already held by a different thread $t' \in T'$, i.e., $t'_c \in L$.

Note that this notion of deadlock is a refinement of the notion presented in Section 3.3. With this notion we do not only cover a deadlock of the whole system but also of parts of the system.

3.6 Conclusion

We generalized the technique of CFL-reachability to the analysis of the synchronized interleavings of multithreaded Java programs. By means of this technique we can decide whether a state in the finite state automaton underlying the product of the individual thread automata is reachable by a synchronized interleaving. We also can decide deadlock freedom.

Future Work We are working on an implementation of our approach using the Meta Environment tools (see [101]). This work first involves the development of a suitable ASF specification to rewrite the parse tree of a Java program to the call graphs which form the basis of our analysis. The next step will be to provide a Meta Environment tool to perform the actual CFL-reachability analysis. Once this implementation for Java is finished it will be interesting to extend the method with further static analysis of the control flow graphs and dataflow in Java programs.

Part III

Active Objects

Chapter 4

Behavioral Interface Description of an Object-Oriented Language with Futures and Promises[1]

This chapter formalizes the observable interface behavior of a concurrent, object-oriented language with futures and promises. The calculus captures the core of *Creol*, a language, featuring in particular asynchronous method calls and, since recently, first-class futures.

The focus of the chapter are *open* systems and we formally characterize their behavior in terms of interactions at the interface between the program and its environment. The behavior is given by transitions between typing judgments, where the absent environment is represented abstractly by an assumption context. A particular challenge is the safe treatment of promises: The erroneous situation that a promise is fulfilled twice, i.e., bound to code twice, is prevented by a resource aware type system, enforcing linear use of the write-permission to a promise. We show subject reduction and the soundness of the abstract interface description.

[1]The work presented in this chapter was published as [4].

4.1 Introduction

How to marry concurrency and object-orientation has been a long-standing issue; (see e.g., [15]) for an early discussion of different design choices. The thread-based model of concurrency, prominently represented by languages like *Java* and C$^\sharp$ has been recently criticized, especially in the context of *component-based* software development. As the word indicates, components are (software) artifacts intended for composition, i.e., open systems, interacting with a surrounding environment. To compare different concurrency models for open systems on a solid mathematical basis, a semantic description of the interface behavior is needed, and this is what we provide in this work. We present an *open semantics* for the core of the *Creol* language [39, 66], an object-oriented, concurrent language, featuring in particular asynchronous method calls and, since recently [42], first-class futures. An open semantics means, it describes the interface behavior of a program or a part of a program, interacting with its environment.

Futures and promises

A *future,* very generally, represents a result yet to be computed. It acts as a proxy for, or reference to, the delayed result from some piece of code (e.g., a method or a function body in an object-oriented, resp. a functional setting). As the consumer of the result can proceed its own execution until it actually needs the result, futures provide a natural, lightweight, and (in a functional setting) transparent mechanism to introduce parallelism into a language. Since their introduction in *Multilisp* [58, 18], futures have been used in various languages like Alice ML [71, 11, 93], E [45], the ASP-calculus [30], *Creol*, and others. A *promise* is a generalization[2] insofar as the reference to the result on the one hand, and the code to calculate the result on the other, are not created at the same time; instead, a promise can be created and only later, after possibly passing it around, be bound to the code (the promise is *fulfilled*).

The notion of futures goes back to functional programming languages. In that setting, futures are annotations to side-effect-free expressions[3], that can

[2] The terminology concerning futures, promises, and related constructs is not too consistent in the literature. Sometimes, the two words are used as synonyms. Interested in the observable differences between futures and promises, we distinguish the concepts and thus follow the terminology as used e.g., in $\lambda(fut)$and Alice ML.

[3]Though in e.g. *Multilisp* also expressions with side-effects can be computed in parallel, but still under the restriction that the observable behavior equals that of the sequential counterpart.

be computed in parallel to the rest of the program. If some program code needs the result of a future, its execution blocks until the evaluation of the future is completed and the result value is automatically fetched back (*implicit* futures). An important property of future-based functional programs is, that future annotations do not change the functionality: the observable behavior of an annotated program equals the observable behavior of its non-annotated counterpart. This property is no longer assured in the object-oriented setting.

To facilitate parallelization, futures were introduced in the `java.util.concurrent` package of the Java 2 Platform Standard Edition 5.0. Here, futures are no annotations: they may execute program code with side-effects, leading to non-determinism and to a different observable behavior. The result of a *Java* future must be explicitly claimed when needed for further computation (*explicit* futures)[4].

Basically, *Java* objects are *passive* objects, i.e., they store data and define methods to manipulate them, but they do not execute any code; the active, executing part are the *threads*. Consequently, *Java* futures consist of the code to be executed in parallel (definable via the Callable interface), some proxy for later access to the result (the Future interface), and the executing threads (Executor interface). Due to this distinction, *Java* futures are not bound to a certain thread. Furthermore, the *Java* setting allows to use promises: a future object can be first created, and bound to some code and to some thread later on.

Interface behavior

An open program, in this setting, interacts with its environment via message exchange. Besides message passing, of course, different communication and synchronization mechanisms exists (shared variable concurrency, multi-cast, black-board communication, publish-and-subscribe and many more). We concentrate here, however, on basic message passing using asynchronous method calls. In that setting, the interface behavior of an open program C can be characterized by the set of all those message sequences (traces) t, for which there *exists* an environment E such that C and E exchange the messages recorded in t. Thereby we abstract away from any concrete environment, but consider only environments that are compliant to the language restrictions (syntax, type system, etc.). Consequently, interactions are not arbitrary traces $C \overset{t}{\Longrightarrow}$;

[4]Explicit futures/promises may be implemented as a library, whereas implicit futures/promises require language support.

instead we consider behaviors $C \parallel E \overset{t}{\underset{\bar{t}}{\Longrightarrow}} \acute{C} \parallel \acute{E}$ where E is a *realizable* environment and trace $\bar{t}$ is complementary to t, i.e., each input is replaced by a matching output and vice versa. The notation $C \parallel E$ indicates that the component C runs in parallel with its environment or observer E. To account for the abstract environment ("there exists an E s.t. ..."), the open semantics is given in an *assumption-commitment* way:

$$\Delta \vdash C : \Theta \overset{t}{\Longrightarrow} \acute{\Delta} \vdash \acute{C} : \acute{\Theta} \ ,$$

where Δ (as an abstract version of E) contains the *assumptions* about the environment, and dually Θ the *commitments* of the component. Abstracting away also from C gives a language characterization by the set of all possible traces between any component and any environment.

Such a behavioral interface description is relevant and useful for the following reasons. 1) The set of possible traces given this way is more restricted (and realistic) than the one obtained when ignoring the environments. When reasoning about the trace-based behavior of a component, e.g., in compositional verification, with a more precise characterization one can carry out stronger arguments. 2) When using the trace description for *black-box testing*, one can describe test cases in terms of the interface traces and then synthesize appropriate test drivers from it. Clearly, it makes no sense to specify impossible interface behavior, as in this case one cannot generate a corresponding tester. 3) A representation-independent behavior of open programs paves the way for a compositional semantics, a two-level semantics for the nested composition of program components. It allows furthermore optimization of components: only if two components show the same external, observable behavior, one can replace one for the other without changing the interaction with any environment. 4) The formulation gives insight into the semantic nature of the language, here, the externally observable consequences of futures and promises. This helps to compare alternatives, e.g., the *Creol* concurrency model with *Java*-like multi-threading.

Contribution

The chapter formalizes the abstract interface behavior for concurrent object-oriented languages with futures and promises. The contributions are the following.

Concurrent object calculus with futures and promises We formalize a class-based concurrent language featuring futures and promises. The formalization is given as a typed, imperative object calculus in the style of [1] resp. one of its concurrent extensions. The operational semantics for components distinguishes unobservable component-internal steps from external steps which represent observable component-environment interactions. We present the semantics in a way that facilitates comparison with the multi-threading concurrency model of *Java*, i.e., the operational semantics is formulated so that the multi-threaded concurrency as (for instance) in *Java* and the one here based on futures are represented similarly.

Linear type system for promises The calculus extends the semantic basis of *Creol* as given for example in [42] with promises. Promises can refer to a computation with code bound to it later, where the binding is done at most once. To guarantee such a *write-once* policy when passing around promises, we refine the type system introducing two type constructors

$$[T]^{+-} \quad \text{and} \quad [T]^{+} \ .$$

The first one represents a reference to a promise that can still be written (and read) with result type T, the second one where only a *read*-permission is available. The write permission constitutes a resource which is consumed when the promise is fulfilled. The resource-aware type system is therefore formulated in a *linear* manner wrt. the write permissions. Linear type systems [102] or linear logics [56] are, roughly speaking, instances of so-called sub-structural type systems resp. logics. In constrast to ordinary such derivation systems, where a hypothesis can be used as many times as needed for carrying out a proof, derivation systems built upon linear use of assumptions work differently: using an assumption in a proof step "*consumes*" it. That feature allows in a natural way to reason about "resources": In our setting, the write-permission is such a resource, and using the corresponding type to derive well-typedness of one part of the program consumes that right such that it is not longer available for type-checking the rest of the program, assuring the intended write-once discipline. The type system resembles in intention the one in [82] for a functional calculus with references. Our work is more general, in that it tackles the problem in an object-oriented setting (which, however, conceptually does not pose much complications), and, more importantly, in that we do not consider closed systems, but open components. Also this aspect of openness is not dealt with in [42]. Additionally, the type system presented here is simpler than in [82], as it avoids the representation of the promise-concept by so-called *handled futures.*

Soundness of the abstractions We show soundness of the abstractions, which includes

- *subject reduction,* i.e., preservation of well-typedness under reduction. Subject reduction is not just proven for a closed system (as usual), but for an open system interacting with its environment. Subject reduction implies furthermore:
- *absence of run-time errors* like "message-not-understood", also for open systems.
- *soundness* of the interface behavior characterization, i.e., all possible interaction behavior is included in the abstract interface behavior description.
- for promises: absence of *write-errors,* i.e. the attempt to fulfill a promise twice.

Related work

The general concept of "delayed reference" to a result of a computation to be yet completed is quite old. The notion of futures was introduced by Baker and Hewitt [18], where (`future` e) denotes an expression executed in a separate thread, i.e., concurrently with the rest of the program. As the result of the expression e is not immediately available, a *future variable* (or future) is introduced as placeholder, which will eventually contain the result of e. In the meantime, the future can be passed around, and when it is accessed for reading ("touched" or "claimed"), the execution suspends until the future value is available, namely when e is evaluated. The principle has also been called *wait-by-necessity* [28, 29]. Futures provide, at least in a purely functional setting, an elegant means to introduce concurrency and *transparent* synchronization simply by accessing the futures. They have been employed for the parallel *Multilisp* programming language [58].

Indeed, quite a number of calculi and programming languages have been equipped with concurrency using future-like mechanisms and asynchronous method calls. Flanagan and Felleisen [50, 48, 49] present an operational semantics, based on evaluation contexts, for a λ-calculus with futures. The formalization is used for analysis and optimization to eliminate superfluous dereferencing ("touches") of future variables. The analysis is an application of a set-based analysis and the resulting transformation is known as touch optimization. Moreau [80] presents a semantics of Scheme equipped with futures and control operators. *Promises* is a mechanism quite similar to futures and actually the two notions are sometimes used synonymously. They have been proposed in [77]. A language featuring both futures and promises as separate concepts, is *Alice ML* [11, 23, 71, 93].

[82] presents a concurrent call-by-value λ-calculus with reference cells (i.e., a non-purely functional calculus with an imperative part and a heap) and with futures ($\lambda(fut)$), which serves as the core of *Alice ML*. Certain aspects of that work are quite close to the material presented here. In particular, we were inspired by using a type system to avoid fulfilling a promise twice (in [82] called handle error). There are some notable differences, as well. The calculus incorporates futures and promises into a λ-calculus, such that functions can be executed in parallel. In contrast, the notion of futures here, in an object-oriented setting, is coupled to the asynchronous execution of methods. Furthermore, the object-oriented setting here, inspired by *Creol*, is more high-level. In contrast, $\lambda(fut)$ relies on an atomic test-and-set operation when accessing the heap to avoid atomicity problems. Besides that, [82] formalizes promises using the notion of *handled* futures, i.e., the two roles of a promise, the writing- and the reading part, are represented by two different references, where the *handle* to the futures represents the writing-end. Apart from that, [82] is not concerned with giving an open semantics as here. On the other hand, [82] investigates the role of the heap and the reference cells, and gives a formal proof that the *only* source of non-determinism by race conditions in their language actually are the reference cells and without those, the language becomes (uniformly) confluent.[5] Recently, an observational semantics for the (untyped) $\lambda(fut)$-calculus has been developed in [81]. The observational equivalence is based on may- and must-program equivalence, i.e., two program fragments are considered equivalent, if, for all observing environments, they exhibit the same necessary and potential convergence behavior.

[5]Uniform confluence is a strengthening of the more well-known notion of (just ordinary) confluence; it corresponds to the diamond property of the *one-step* reduction property. For standard reduction strategies of a purely functional λ-calculus, only confluence holds, but not uniform confluence. However, the non-trivial "diamonds" in the operational semantics of $\lambda(fut)$ are caused not by different redexes within one λ-term (representing one thread), but by redexes from different threads running in parallel, where the reduction strategy per thread is deterministic (as in our setting, as well).

Futures have also been investigated in the object-oriented paradigm. For instance, the object-oriented language Scala [84] has recently been extended [57] by actor-based concurrency, offering futures and promises as part of the standard library. The futures and promises are inspired by their use in *Alice ML*. In *Java* 5, *futures* have been introduced as part of the `java.util.concurrent` package. As *Java* does not support futures as a core mechanism for parallelism, they are introduced in a library. Dereferencing of a future is done explicitly via a `get`-method (similarly to this chapter). A recent paper [103] introduces *safe* futures for *Java*. The safe concept is intended to make futures and the related parallelism *transparent* and in this sense goes back to the origins of the concept: introducing parallelism via futures does not change the meaning of a program. While straightforward and natural in a functional setting, safe futures in an object-oriented and thus state-based language such as *Java* require more considerations. [103] introduces a semantics which guarantees safe, i.e., transparent, futures by deriving restrictions on the scheduling of parallel executions and uses object versioning. The futures are introduced as an extension of Featherweight *Java* (*FJ*) [59], a core object calculus, and is implemented on top of *Jikes* RVM [12, 26]. Pratikakis et. al. [87] present a constraint-based static analysis for (transparent) futures and proxies in *Java*, based on type qualifiers and qualifier inference [51]. Also this analysis is formulated as an extension of *FJ* by type qualifiers. Similarly, Caromel et. al. [32, 31, 30] tackle the problem to provide *confluent*, i.e., effectively deterministic system behavior for a concurrent object calculus with futures (asynchronous sequential processes, *ASP*, an extension of Abadi and Cardelli's imperative, untyped object calculus impς [1]) and in the presence of imperative features. The *ASP* model is implemented in the *ProActive Java*-library [33]. The fact, that *ASP* is derived from some (sequential, imperative) object-calculus, as in the formalization here, is more a superficial or formal similarity, in particular when being interested in the interface behavior of concurrently running objects, where the inner workings are hidden anyway. Apart from that there are some similarities and a number of differences between the work presented here and *ASP*. First of all, both calculi are centered around the notion of first-class futures, yielding active objects. The treatment, however, of getting the value back, is done differently in [30]. Whereas here, the client must explicitly claim a return value of an asynchronous method, if interested in the result, the treatment of the future references is done *implicitly* in *ASP*, i.e., the client blocks if it performs a strict operation on the future (without explicit syntax to claim the value). Apart from that, the object model is more sophisticated, in that the calculus distinguishes between active and passive objects. Here, we simple have objects, which can behave actively or passively (reactively), depending on the way they are used. In *ASP*, the units of concurrency are the explicitely activated active objects, and each passive one is owned and belongs to exactly one active one. Especially, passive objects do not directly communicate with each other across the boundaries of concurrent activity, all such communication between concurrent activities is mediated and served by the active objects.

Related to that, a core feature of *ASP*, not present here, is the necessity to specify (also) the *receptive behavior* of the active object, i.e., in which order it is willing to process or *serve* incoming messages. The simplest serve strategy would be the willingness to accept all messages and treat them in a first-come, first-serve manner, i.e., a input-enabled FIFO strategy on the input message queue. The so-called *serve*-method is the dedicated activity of an active object to accept and schedule incoming method calls. Typically, as for instance in the FIFO case, it is given as a non-terminating process, but it might also terminate, in which case the active object together with the passive objects it governs, becomes superfluous: an active object which does no service any longer does not become a passive data structure, but can no longer react in any way.

As extension of the core *ASP* calculus, [30, Chapter 10] treats *delegation* that bears some similarities with the promises here. By executing the construct $delegate(o.l(\vec{v}))$ (using our notational conventions), a thread n hands over the permission and obligation to provide eventually a value for the future reference n to method l of object o, thereby losing that permission itself. That corresponds to executing $\mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n$. Whereas in our setting, we must use a yet-unfulfilled promise n for that purpose, the delegation operator in *ASP* just (re-)uses the current future for that. Consequently, *ASP* does not allow the creation of promises independently from the implicit creation when asynchronously calling a method, as we do with the $\mathsf{promise}\, T$ construct. In this sense, the promises here are more general, as they allow to profit from delegation and have the promise as first-class entity, i.e., the programmer can pass it around, for instance, as argument of methods. This, on the other hand, requires a more elaborate type system to avoid write-errors on promises. This kind of error, fulfilling a promise twice, is avoided in the delegate-construct of *ASP* not by a type system, but by construction, in that the delegate-construct must be used only at the end of a method, so that the delegating activity cannot write to the future/promise after it has delegated the permission to another activity.

Further uses of futures for *Java* are reported in [67, 78, 88, 98, 97]. Futures are also integral part of *Io* [60] and *Scoop* (simple concurrent object-oriented programming) [17, 38, 79], a concurrent extension of *Eiffel*. Both languages are based on the active objects paradigm.

Benton et. al. [22] present polyphonic C$^\sharp$, adding concurrency to C$^\sharp$, featuring asynchronous methods and based on the join calculus [52, 53]. Polyphonic C$^\sharp$ allows methods to be declared as being asynchronous using the async keyword for the method type declaration. Besides that, polyphonic C$^\sharp$ supports so-called *chords* as synchronization or join pattern. With similar goals, *Java* has been extended by join patterns in [61, 62].

In the context of *Creol*, de Boer et. al. [42] present a formal, operational semantics for the language and extend it by *futures* (but not promises). Besides the fact, that both operational semantics ultimately formalize a comparable set of features, there are, at a technical level, a number of differences. For once, here, we simplified the language slightly mainly in two respects (apart from making it more expressive in adding promises, of course). We left out the "interleaving" operators $|||$ and $///$ of [42] which allow the user to express interleaving concurrency *within* one method body. Being interested in the observable interface behavior, those operations are a matter of internal, hidden behavior, namely leading to non-deterministic behavior at the interface. Since objects react non-deterministically anyhow, namely due to race conditions present independently of $|||$ and $///$, those operators have no impact on the possible traces at the interface. The operators might be useful as abstractions for the programmer, but without relevance for the interface traces, and so we ignore them here. Another simplification, this time influencing the interface behavior, is how the programmer can claim the value of a future. This influences, as said, the interface behavior, since the component may fetch the value of a future being part of the environment, or vice versa. Now, the design of the *Creol*-calculus in [42] is more liberal wrt. what the user is allowed to do with future references. In this chapter, the interaction is rather restricted: if the client requests the value using the claim-operation, there are basically only two reactions. If the future computation has already been completed, the value is fetched and the client continues; otherwise it blocks until, if ever, the value is available. The bottom line is, that the client, being blocked, can never *observe* that the value is yet absent. The calculus of [42], in contrast, permits the user to *poll* the future reference directly, which gives the freedom to decide, *not* to wait for the value if not yet available. Incorporating such a construct into the language makes the *absence* of the value for a future reference observable and would complicate the behavioral interface semantics to some extent. This is also corroborated by the circumstance that the expressive power of explicit polling quite complicates the proof theory of [42] (see also the discussion in the conclusion of [42]). This is not a coincidence, since one

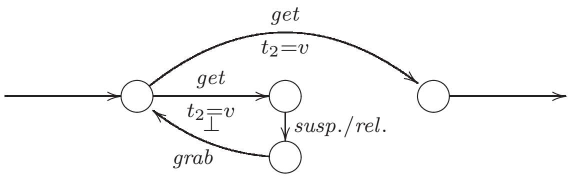

Figure 4.1: Claiming a future (busy wait)

crux of the complete Hoare-style proof systems such as in [42] is to internalize the (ideally observable) behavior into the program state by so-called auxiliary variables. In particular recording the past interaction behavior in history variables is, of course, an internalization of the interface behavior, making it visible to the Hoare-assertions. As a further indication that allowing to poll a future quite adds expressivity to the language is the observation that adding a poll-operation to *ASP*, destroys a central property of *ASP*, namely confluence, as is discussed in [30, Chapter 11].

Apart from that, the combination of claiming a futures, the possibility of polling a future, and a general await-statement complicates the semantics of claiming a future: in [42], this is done by *busy-waiting*, which in practice one intends to avoid. So instead of the behavior described in Figure 4.3, the formalization in [42] behaves as sketched in Figure 4.1.

After an unsuccessful try to obtain a value of future, the requesting thread is suspended and loses the lock. In order to continue executing, the blocked thread needs two resources: the value of the future, once it is there, plus the lock again. The difference of the treatment in Figure 4.3 and the one of Figure 4.1 for [42] is the order in which the requesting thread attempts to get hold of these two resources: our formalization first check availability of the future and afterwards re-gains the lock to continue, whereas [42] do it vice versa, leading to busy wait. The reason why it is sound to copy the future value into the local state space without already having the lock again (cf. Figure 4.3) is, of course, that, once there, the future value remains stable and available.

In addition, our work differs also technically in the way, the operational semantics is represented. [42] formulated the (internal) operational semantics using evaluation contexts (as do, e.g., [82] for $\lambda(\mathit{fut})$), whereas we rely on a "reduction-style" semantics, making use of an appropriate notion of structural congruence. While largely a matter of taste, it seems to us that, especially in the presence of complicated synchronization mechanisms, for instance the ready queue representation of [42], the evaluation contexts do not give rise to an immediately more elegant specification of the reduction behavior. Admittedly, we ignored here the internal interleaving operators $|||$ and $///$, which quite contribute to the complexity of the evaluation contexts. Another technical difference concerns the way, the futures, threads, and objects are *represented* in the operational semantics, i.e., in the run-time syntax of the calculus. Different from our representation, their semantics makes the active-objects paradigm of *Creol* more visible: The activities are modeled as part of the object. More precisely, an object contains, besides the instance state, an explicit representation of the current activity (there called "process") executing "inside" the object plus a representation of the ready-queue containing all the activities, which have been *suspended* during their execution inside the object. The scheduling between the different activities is then done by juggling them in and out of the ready-queue at the processor release points. Here, in contrast, we base our semantics on a separate representation of the involved semantics concepts: 1) classes as *generators* of objects, 2) objects carrying in the instance variables the persistent *state* of the program, thus basically forming the heap, and 3), the *parallel* activities in the form of threads. While this representation makes arguably the active-object paradigm less visible in the semantics, it on the other hand separates the concepts in a clean way. Instead of an explicit local scheduler inside the objects, the access to the shared instance states of the objects is regulated by a simple, binary lock per object. So, instead of having two levels of parallelism —locally inside the objects and inter-object parallelism— the formalization achieves the same with just one conceptual level, namely: parallelism is between threads (and the necessary synchronization is done via object-locks). Additionally, our semantics is rather close to the object-calculi semantics for multi-threading as in *Java* [63, 64, 95]. This allows to see the differences and similarities between the different models of concurrency, and the largely similar representation allows a more formal comparison between the interface behaviors in the two settings.

The language *Cool* [36, 37] (concurrent, object-oriented language) is defined as an extension of C^{++} [96] for task-level parallelism on shared memory multi-processors. Concurrent execution in *Cool* is expressed by the invocation of parallel functions executing *asynchronously.* Unlike the work presented here, *Cool* contains future types, which correspond to the types of the form $[T]$ used here. Further languages supporting futures include ACT-1 [75, 76], concurrent *Smalltalk* [104, 107], and of course the influential actor model [7, 8, 9], *ABCL/1*[105, 106] (in particular the extension *ABCL/f*[99]).

We have characterized the behavioral semantics of open systems, similarly to the one presented here for futures and promises, in earlier papers, especially for object-oriented languages based on *Java*-like multithreading and synchronous method calls, as in *Java* or C$^\sharp$. [6] deals with thread classes and [5] with re-entrant monitors. In [95] the proofs of full abstraction for the sequential and multithreaded cases of a class-based object-calculus can be found. Poetzsch-Heffter and Schäfer [86] present a behavioral interface semantics for a class-based object-oriented calculus, however without concurrency. The language, on the other hand, features an ownership-structured heap.

Outline The chapter is organized as follows. Section 4.2 defines the syntax, the type system, and the operational semantics, split into an internal one, and one for the interface behaviour of open systems. Section 4.3 describes the interface behavior. Section 4.4 concludes with related and future work. For more details see [3].

4.2 Calculus

This section presents the calculus, based on a version of the *Creol*-language with first-class futures [42] and extended with promises. It is a concurrent variant of an imperative, object-calculus in the style of the calculi from [1]. Our calculus covers first-class futures, which can be seen as a generalization of asynchronous method calls and promises.

We start with the abstract syntax in Section 4.2.1. After discussing the type system in Section 4.2.2, we present the operational semantics in Section 4.2.3.

4.2.1 Syntax

The abstract syntax is given in Figure 4.2. It distinguishes between *user* syntax and *run-time* syntax (the latter underlined). The user syntax contains the phrases in which programs are written; the run-time syntax contains syntactic constituents additionally needed to express the behavior of the executing program in the operational semantics. The latter are not found in a program written by the user, but generated at run-time by the rules of the operational semantics.

The basic syntactic category of names n, which count among the values v, represents references to classes, to objects, and to futures/promises. To facilitate reading, we allow ourselves to write o and its syntactic variants for names referring to objects, c for classes, and n when being unspecific. Technically, the disambiguation between the different roles of the names is done by the

$$
\begin{array}{rcll}
C & ::= & \mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).C} \mid n[\![O]\!] \mid \underline{n[n,F,L]} \mid \underline{n\langle t\rangle} \mid \underline{n\langle\bullet\rangle} & \text{component} \\
O & ::= & F, M & \text{object} \\
M & ::= & l = m, \ldots, l = m & \text{method suite} \\
F & ::= & l = f, \ldots, l = f & \text{fields} \\
m & ::= & \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t & \text{method} \\
f & ::= & \varsigma(n{:}T).\lambda().v \mid \varsigma(n{:}T).\lambda().\bot_{n'} & \text{field} \\
t & ::= & v \mid \mathsf{stop} \mid \mathsf{let}\, x{:}T = e \,\mathsf{in}\, t & \text{thread} \\
e & ::= & t \mid \mathsf{if}\, v = v \,\mathsf{then}\, e \,\mathsf{else}\, e \mid \mathsf{if}\, \mathit{undef}(v.l()) \,\mathsf{then}\, e \,\mathsf{else}\, e & \text{expr.} \\
 & \mid & \mathsf{promise}\ T \mid \mathsf{bind}\, n.l(\vec{v}) : T \hookrightarrow n \mid v.l() \mid v.l := \varsigma(s{:}n).\lambda().v & \\
 & \mid & \mathsf{new}\, n \mid \mathsf{claim@}(n,n) \mid \underline{\mathsf{get@}n} \mid \mathsf{suspend}(n) \mid \underline{\mathsf{grab}(n)} \mid \underline{\mathsf{release}(n)} & \\
v & ::= & x \mid n \mid () & \text{values} \\
L & ::= & \bot \mid \top & \text{lock status}
\end{array}
$$

Figure 4.2: Abstract syntax

type system and the abstract syntax of Figure 4.2 uses the non-specific n for names. The unit value is represented by $()$ and x stands for variables, i.e., local variables and formal parameters, but not instance variables.

A *component* C is a collection of classes, objects, and (named) threads, with $\mathbf{0}$ representing the empty component. The sub-entities of a component are composed using the parallel-construct $\parallel$. The entities executing in parallel are the named threads $n\langle t\rangle$, where t is the code being executed and n the name of the thread. In the given setting, threads are always promises (with the exception of initial threads, see Section 4.2.3), with their name being the reference under which the result value of t, if any[6], will be available. A class $c[\![O]\!]$ carries a name c and defines its methods and fields in O. An object $o[c, F, L]$ with identity o keeps a reference to the class c it instantiates, stores the current value F of its fields, and maintains a *binary lock* L indicating whether any code is currently active inside the object (in which case the lock is taken) or not (in which case the lock is free). The symbols $\top$, resp., $\bot$, indicate that the lock is taken, resp., free. From the three kinds of entities at component level —threads $n\langle t\rangle$, classes $c[\![O]\!]$, and objects $o[c, F, L]$— only the threads are *active*, executing entities, being the target of the reduction rules. The objects, in contrast, store the state in their fields or instance variables, whereas the classes are constant entities specifying the methods.

The named threads $n\langle t\rangle$ are incarnations of method bodies "in execution". Incarnations insofar, as the formal parameters have been replaced by actual ones, especially the method's self-parameter has been replaced by the identity of the target object of the method call. The term t is basically a sequence

[6]There will be no result value in case of non-terminating methods.

of expressions, where the let-construct is used for sequencing and for local declarations.[7] During execution, $n\langle t\rangle$ contains in t the currently running code of a method body. When evaluated, the thread is of the form $n\langle v\rangle$ and the value can be accessed via n, the future reference, or future for short.

Each thread belongs to one specific object "inside" which it executes, i.e., whose instance variables it has access to. Object locks are used to rule out unprotected concurrent access to the object states: though each object may have more than one method body incarnation partially evaluated, at each time point at most one of those bodies (the lock owner) can be active inside the object. In the terminology of *Java*, all methods are implicitly considered "synchronized". A crucial difference between a concurrency model based on multi-threading like *Java* and a concurrency model based on active objects like *Creol* is the way method calls are issued at the caller site. In *Java* and similar languages, method calls are *synchronous* in the sense that the calling activity blocks to wait for the return of the result and thus the control is transferred to the callee. Here, method calls are issued *asynchronously,* i.e., the calling thread continues executing and the code of the method being called is computed concurrently in a new thread located in the callee object. In that way, a method call never transfers control from one object, the caller, to another one, the callee. In other words, no thread ever crosses the boundaries of an object, which means, the boundaries of an object are at the same time boundaries of the threads and thus, the objects are at the same time units of concurrency. Thus, the objects are harnessing the activities and can be considered as bearers of the activities. This is typical for object-oriented languages based on *active objects*.

The final construct at the component level is the ν-operator for hiding and dynamic scoping, as known from the π-calculus. In a component $C = \nu(n{:}T).C'$, the scope of the name n (of type T) is restricted to C' and unknown outside C. ν-binders are introduced when dynamically creating new named entities, i.e., when instantiating new objects or new promises. The scope is dynamic, i.e., when the name is communicated by message passing, it is enlarged.

Besides components, the grammar specifies the lower level syntactic constructs, in particular, methods, expressions, and (unnamed) threads, which are basically sequences of expressions. A method $\varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t$ provides the method body t abstracted over the ς-bound "self" parameter s and the formal

[7] $t_1; t_2$ (sequential composition) abbreviates $\mathsf{let}\, x{:}T = t_1 \,\mathsf{in}\, t_2$, where x does not occur free in t_2.

parameters $\vec{x}$. For uniformity, fields are represented as methods without parameters (with the exception of the standard self-parameter). The "body" of a field is either a value or yet undefined. Note that the methods are stored in the classes but the fields are kept in the objects, of course. In freshly created objects, the lock is free, and all fields carry the undefined reference $\perp_c$, where class name c is the (return) type of the field.

We use f for instance variables or fields and $l = \varsigma(s{:}T).\lambda().v$, resp. $l = \varsigma(s{:}T).\lambda().\perp_c$ for field variable definition. Field access is written as $v.l()$ and field update as $v'.l := \varsigma(s{:}T).\lambda().v$. By convention, we abbreviate the latter constructs by $l = v$, $l = \perp_c$, $v.l$, and $v'.l := v$. Note that the construct $v.l()$ is used for field access only, but not for method invocation. We will also use $v_\perp$ to denote either a value v or a symbol $\perp_c$ for being undefined. Note that the syntax does not allow to set a field back to undefined. Direct access (read or write) to fields across object boundaries is forbidden by convention, and we do not allow method update. Instantiation of a new object from class c is denoted by $\mathsf{new}\, c$.

Expressions especially include syntax to deal with promises and futures. The expression $\mathsf{promise}\, T$ creates a new promise, i.e., a reference or name for a result yet to come. At the point of creation, only the name exists, but no code has been determined and attached to the reference to calculate the result. Binding code to the promise is done by $\mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n$, stipulating that the eventual result is calculated using the method l of object o with actual parameters $\vec{v}$. Executing the binding operation is also known as *fulfilling* the promise. Some languages do not allow to *independently* create a name for the eventual result, i.e., creation and binding are done inseparately by one single command. In that situation, one does not speak of promises, but (just) of futures, even if in the literature, sometimes no distinction is drawn between futures and promises. In a certain way, futures and promises can be seen as two different roles of a reference n: the promise-role means, a client can *write* to the name using the bind-operation, and the future-role represents the possibility to *read* back an eventual result using the reference. In this way, we will use both the term future and promise when referring to the same reference, depending on the role it is playing when used.

The expression $\mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n$ binds a *method body* to the promise n. Thus, there is a close connection to asynchronous method calls, central to the concurrency model of *Creol*. Indeed, in comparison with [42], which introduces the concept of futures (but not promises) into *Creol*, asynchronous calls are syntactic sugar for creating a new promise and immediately binding $o.l(\vec{v})$ to it. This behaves as an asynchronous method call, as the one creating the promise and executing the bind can continue without being blocked waiting for the result.

The further expressions claim, get, suspend, grab, and release deal with communication and synchronization. As mentioned, objects come equipped with binary locks, responsible for assuring mutual exclusion. The two basic, complementary operations on a lock are grab and release. The first allows an activity to acquire access in case the lock is free ($\bot$), thereby setting it to $\top$, and release(o) conversely relinquishes the lock of the object o, giving other threads the chance to be executed in its stead, when succeeding to grab the lock via grab(o). The user is not allowed to directly manipulate the object locks. Thus, both expressions belong to the run-time syntax, underlined in Figure 4.2, and are only generated and handled by the operational semantics as auxiliary expressions at run-time. Instead of using directly grab and release, the lock-handling is done automatically when executing a method body: before starting to execute, the lock has to be acquired and upon termination, the lock is released again. Besides that, lock-handling is involved also when futures are claimed, i.e., when a client code executing in an object, say o, intends to read the result of a future. The expression claim@(n, o) is the attempt to obtain the result of a method call from the future n while in possession of the lock of object o. There are two possibilities in that situation: either the value of the future has already been determined, i.e., the method calculating the result has terminated, in which case the client just obtains the value *without* loosing its own lock. In the alternative case, where the value is not yet determined, the client trying to read the value gives up its lock via release and continues executing only after the requested value has been determined (using get to read it) and after it has re-acquired the lock. Unlike claim, the get-operation is not part of the user-syntax. Both expressions are used to read back the value from a future and the difference in behavior is that get unconditionally attempts to get the value, i.e., blocks until the value has arrived, whereas claim gives up the lock temporarily, if the value has not yet arrived, as explained. This behavior is sketched in Figure 4.3. Note the order in which get and grab are executed after releasing the lock: the value is read in via get *before* the lock has actually been re-acquired! That this order is acceptable rests on the fact that a future, once evaluated, does not change the value later and reading the value in by itself has no side-effect. Reversing the order —first re-aquiring the lock and afterwards checking for availability of the future's value— would result in equivalent behavior but amount to busy waiting. Finally, executing suspend(o) causes the activity to relinquish and re-grab the lock of the object o. We assume by convention, that when appearing in methods of classes, the

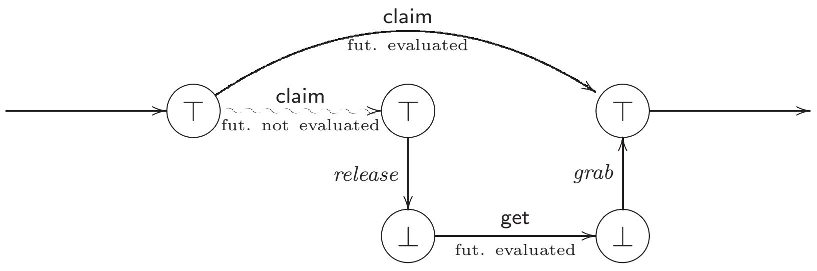

Figure 4.3: Claiming a future

claim- and the suspend-command only refer to the self-parameter *self*, i.e., they are written claim@(n, self) and suspend(self).

Before continuing with the type system, let us explain how and why we exclude a specific potential deadlock situation in the semantics of the claim-statement (though the language does not generally exclude the presence of deadlocks, i.e., it is possible to write a deadlocking program in the language). Remember that if a thread is about to execute a claim-statement in an object, it always owns the lock of the object. If the claimed result is not yet available, then the claiming thread blocks. During blocking, if we would not release the lock previously, no other thread could execute in the object, since it would require the lock of the object. Consequently, if the computation of the claimed result needs execution in the object, the threads would deadlock. Such deadlocks could not be easily excluded syntactically, since release and grab are only auxiliary syntax, i.e., they cannot be used to write programs, and we do not support checking if a thread already finished its computation. Thus we release the lock before blocking, i.e., waiting for the claimed result, and re-grab the lock after the thread got the result.

4.2.2 Type system

The language is typed and the available types are given in the following grammar:

$$\begin{array}{lll} T & ::= & B \mid \mathsf{Unit} \mid [T]^{+-} \mid [T]^{+} \mid [l{:}U,\ldots,l{:}U] \mid [\![l{:}U,\ldots,l{:}U]\!] \mid n \\ U & ::= & T \times \ldots \times T \to T \end{array}$$

Besides base types B (left unspecified; typical examples are booleans, integers, etc.), Unit is the type of the unit value (). Types $[T]^{+-}$ and $[T]^{+}$ represent the reference to a future which will return a value of type T, in case it eventually terminates. $[T]^{+-}$ indicates that the promise has not yet been fulfilled, i.e., it represents the write-permission to a promise, which implies read-permission at the same time. $[T]^{+}$ represents read-only-permission to a future. The read/write capability is more specific than read-only, which is expressed by the (rather trivial) subtyping relation generated by $[T]^{+-} \leq [T]^{+}$, accompanied by the usual subsumption rule. Furthermore, $[\_]^{+}$ acts monotonely, and $[\_]^{+-}$ invariantly wrt. subtyping. When not interested in the access permission, we just write $[T]$.

The name of a class serves as the type for its instances. We need as auxiliary type constructions (i.e., not as part of the user syntax, but to formulate the type system) the type or interface of unnamed objects, written $[l_1{:}U_1, \ldots, l_k{:}U_k]$ and the interface type for classes, written $[\![(l_1{:}U_1, \ldots, l_k{:}U_k)]\!]$. We allow ourselves to write $\vec{T}$ for $T_1 \times \ldots \times T_k$ etc. where we assume that the number of arguments match in the rules, and write $\mathsf{Unit} \rightarrow T$ for $T_1 \times \ldots \times T_k \rightarrow T$ when $k = 0$.

We are interested in the behavior of well-typed programs, only, and this section presents the type system to characterize those. As the operational rules later, the derivation rules for typing are grouped into two sets: one for typing on the level of components, i.e., global configurations, and secondly one for their syntactic sub-constituents (cf. also the two different levels in the abstract syntax of Figure 4.2).

In Figure 4.4 we define the typing on the level of global configurations, i.e., for "sets" of objects, classes, and named threads. On this level, the typing judgments are of the form

$$\Delta \vdash C : \Theta \ , \tag{4.1}$$

where Δ and Θ are *name contexts*, i.e., finite mappings from names (of classes, objects, and threads) to types. In the judgment, Δ plays the role of the typing assumptions about the *environment,* and Θ of the commitments of the *component,* i.e., the names offered to the environment. Sometimes, the words required and provided interface are used to describe their dual roles. Δ must contain at least all external names referenced by C and dually Θ mentions the names offered by C. Both contexts constitute the static interface information. A pair Δ and Θ of assumption and commitment context with disjoint domains is called *well-formed.*

The empty configuration $\mathbf{0}$ is well-typed in any context and exports no names (cf. rule T-EMPTY). Two configurations in parallel can refer mutually

$$\frac{}{\Delta \vdash \mathbf{0} : ()}\ \textsc{T-Empty} \qquad \frac{\Delta' \leq \Delta \quad \Theta \leq \Theta' \quad \Delta \vdash C : \Theta}{\Delta' \vdash C : \Theta'}\ \textsc{T-Sub}$$

$$\frac{\Delta_1, \Theta_2 \vdash C_1 : \Theta_1 \quad \Delta_2, \Theta_1 \vdash C_2 : \Theta_2}{\Delta_1 \oplus \Delta_2 \vdash C_1 \parallel C_2 : \Theta_1, \Theta_2}\ \textsc{T-Par} \qquad \frac{\Delta \vdash C : \Theta, n{:}T}{\Delta \vdash \nu(n{:}T).C : \Theta}\ \textsc{T-Nu}$$

$$\frac{;\Delta, c{:}T \vdash [\![O]\!] : T}{\Delta \vdash c[\![O]\!] : (c{:}T)}\ \textsc{T-NClass} \qquad \frac{;\Delta \vdash c : [\![T_F, T_M]\!] \quad ;\Delta, o{:}c \vdash [F] : [T_F]}{\Delta \vdash o[c, F, L] : (o{:}c)}\ \textsc{T-NObj}$$

$$\frac{;\Delta, n{:}[T]^{+} \vdash t : T}{\Delta \vdash n\langle t\rangle : (n{:}[T]^{+})}\ \textsc{T-NThread} \qquad \frac{}{\Delta \vdash n\langle \bullet \rangle : (n{:}[T]^{+-})}\ \textsc{T-NThread}'$$

Figure 4.4: Typing (component level)

to each other's commitments and together offer the (disjoint) union of their names (cf. rule T-Par). It is an invariant of the operational semantics that the identities of parallel entities are disjoint wrt. the mentioned names. Therefore, Θ_1 and Θ_2 in the rule for parallel composition are merged disjointly, indicated by writing Θ_1, Θ_2 (analogously for the assumption contexts). Also the treatment of the assumption context requires care wrt. the write-permissions. In general, C_1 and C_2 can rely on the same assumptions that also $C_1 \parallel C_2$ in the conclusion uses, as it represents the environment *common* to $C_1 \parallel C_2$. This, however, does not apply to the write-permissions: if $C_1 \parallel C_2$ do have write-permission on a promise n, which resides in the environment of $C_1 \parallel C_2$, this is represented as $n{:}[T]^{+-}$ in the assumptions of the parallel composition. Due to the linear nature of the write-permission, however, the binding $n{:}[T]^{+-}$ can occur only in the assumptions of either C_1 or of C_2 in the two premises of T-Par. In other words, the assumption context of $C_1 \parallel C_2$ must be split as far as the write-permissions to promises are concerned. To capture this intuition, we define:

Definition 4.2.1 *Let the symmetric operation $\oplus$ on well-formed name contexts be defined as follows:*

$$\begin{array}{lcll}
\mathbf{0} \oplus \Delta & = & \Delta & \\
n{:}[T]^{+}, \Delta_1 \oplus n{:}[T]^{+-}, \Delta_2 & = & n{:}[T]^{+-}, (\Delta_1 \oplus \Delta_2) & \\
n{:}T, \Delta_1 \oplus n{:}T, \Delta_2 & = & n{:}T, (\Delta_1 \oplus \Delta_2) & T \neq [T']^{+-} \textit{ for some } T' \\
\Delta_1 \oplus \Delta_2 & = & \text{undefined} & \textit{else}
\end{array}$$

We omit symmetric rules (e.g. for $\Delta \oplus \mathbf{0}$). The contexts are considered as *unordered, i.e., $n{:}T, \Delta$ does not mean, the binding $n{:}T$ occurs leftmost in a "list".*

In combination with the rest of the rules (in particular, rule T-Bind in Figure 4.6), this assures that a promise cannot be fulfilled by the component and the environment at the same time.

The ν-binder hides the bound object or the name of the future inside the component (cf. rule T-Nu). In the T-Nu-rule, we assume that the bound name n is new to Δ and Θ. Let-bound variables are *stack* allocated and checked in a stack-organized variable context Γ (see Figures 4.5 and 4.6). Object names created by new and future/promise names created by promise are *heap* allocated and thus checked in a "parallel" context (cf. again the assumption-commitment rule T-Par). The rules for named classes introduce the name of the class and its type into the commitment (cf. T-NClass). The code $[\![O]\!]$ of the class $c[\![O]\!]$ is checked in an assumption context where the name of the class is available. Note also that the premise of T-NClass (like those of T-NObj and T-NThread) is not covered by the rules for type checking at the component level, but by the rules for the lower level entities (in this particular case, by rule T-Obj from Figure 4.5). The judgments in Figures 4.5 and 4.6 use as assumption not only a name context, but additionally a stack-organized, variable context Γ in order to handle the let-bound variables. So in general, the assumption context at that level is of the form $\Gamma; \Delta$. The premise of T-NClass starts, however, with Γ being empty, i.e., with no assumptions about the type of local variables. This is written in the premise as $; \Delta, c{:}T \vdash [\![O]\!] : T$; similar for the premises of T-NObj and T-NThread. An instantiated object will be available in the exported context Θ by rule T-NObj.

Promises, that are not yet fulfilled, are present in the configuration as thread entities $n\langle\bullet\rangle$ (see Section 4.3); their type $[T]^{+-}$ can be derived by rule T-NThread$'$. Fulfilled promises $n\langle t\rangle$ are treated by rule T-NThread, where the type $[T]^{+}$ of the future reference n is matched against the result type T of thread t. As n is already fulfilled, its type exports read-permission, only. As t may refer to n, it is checked in the premise by Δ extended by the appropriate binding $n{:}[T]^{+}$. The last rule is a rule of subsumption, expressing a simple form of subtyping: we allow that an object respectively a class contains *at least* the members which are required by the interface. This corresponds to width subtyping. Note, however, that each named object has exactly one type, namely its class.

Definition 4.2.2 (Subtyping) *The relation $\leq$ on types is defined as identity for all types except for $[T]^{+-} \leq [T]^{+}$ (mentioned above) and object interfaces,* *where we have:*

$$[\![l_1{:}U_1,\ldots,l_k{:}U_k,l_{k+1}{:}U_{k+1},\ldots]\!] \leq [\![l_1{:}U_1,\ldots l_k{:}U_k]\!] \ .$$

For well-formed name contexts Δ_1 and Δ_2 , we define in abuse of notation $\Delta_1 \leq \Delta_2$, if Δ_1 and Δ_2 have the same domain and additionally $\Delta_1(n) \leq \Delta_2(n)$ for all names n.

The definition is applied, of course, also to name contexts Θ, used for the commitments. The relations $\leq$ are obviously reflexive, transitive, and anti-symmetric.

Next we formalize the typing for objects and threads and their syntactic sub-constituents. Again, the treatment of the write-permissions requires care: The capability to write to a promise is consumed by the bind-operation as it should be done only once. This is captured by a *linear* type system where the execution of a thread or an expression may change the involved types. The judgments are of the form

$$\Gamma;\Delta \vdash e : T :: \acute{\Gamma},\acute{\Delta}, \tag{4.2}$$

where the change from Γ and Δ to $\acute{\Gamma}$ and $\acute{\Delta}$ reflects the potential consumption of write-permissions when executing e. The consumption is only potential, as the type system statically overapproximates the run-time behavior, of course. The typing is given in Figures 4.5 and 4.6. For brevity, we write $\Delta;\Gamma \vdash e : T$ for $\Delta;\Gamma \vdash e : T :: \acute{\Gamma},\acute{\Delta}$, when $\acute{\Gamma} = \Gamma$ and $\acute{\Delta} = \Delta$. Besides assumptions about the provided names of the environment kept in Δ, the typing is done relative to assumptions about occurring free variables. They are kept separately in a variable context Γ, a finite mapping from variables to types. Apart from the technicalities, treating the write capabilities in a linear fashion is straightforward: one must assure that the corresponding capability is available at most once in the program and is not duplicated when passed around. A promise is no longer available for writing when bound to a variable using the let-construct, or when handed over as argument to a method call or a return.

Classes, objects, and methods resp. fields have no effect on Δ (see rules T-Class, T-Obj, T-Memb, and T-Undef). Note that especially in T-Memb, the name context Δ does not change. This does *not* mean, that a method cannot have a side-effect by fulfilling promises, but they are not part of the check of the method *declaration* here. Rule T-Class is the introduction rule for class types, the rule of instantiation of a class T-NewC requires reference to a class-typed name. In the rules T-Memb and T-FUpdate we use the meta-mathematical notation $T.l$ to pick the type in T associated with label l, i.e.,

$$\frac{\Gamma;\Delta \vdash c : [\!(l_1{:}U_1,\ldots,l_k{:}U_k)\!] \quad \Gamma;\Delta \vdash m_i : U_i \quad m_i = \varsigma(s_i{:}c).\lambda(\vec{x}_i{:}\vec{T}_i).t_i}{\Gamma;\Delta \vdash [\!(l_1 = m_1,\ldots,l_k = m_k)\!] : c}\ \textrm{T-C{\scriptsize LASS}}$$

$$\frac{\Gamma;\Delta \vdash c : [\!(l_1{:}U_1,\ldots,l_k{:}U_k)\!] \qquad \Gamma;\Delta \vdash f_i : U_i \qquad f_i = \varsigma(s_i{:}c).\lambda().v_\perp}{\Gamma;\Delta \vdash [l_1 = f_1,\ldots,l_k = f_k] : c}\ \textrm{T-O{\scriptsize BJ}}$$

$$\frac{\Gamma,\vec{x}{:}\vec{T};\Delta,s{:}c \vdash t : T' :: \acute{\Gamma};\acute{\Delta} \quad \Gamma;\Delta \vdash c : T \quad T = [\!(\ldots,l{:}\vec{T} \to T',\ldots)\!]}{\Gamma;\Delta \vdash \varsigma(s{:}c).\lambda(\vec{x}{:}\vec{T}).t : T.l}\ \textrm{T-M{\scriptsize EMB}}$$

$$\frac{\Gamma;\Delta,s{:}c \vdash c : [\!(\ldots,l : \mathsf{Unit} \to c',\ldots)\!]}{\Gamma;\Delta \vdash \varsigma(s{:}c).\lambda().\perp_{c'} : c'}\ \textrm{T-U{\scriptsize NDEF}}$$

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : T \quad \Gamma;\Delta \vdash v' : T.l}{\Gamma;\Delta \vdash v.l := v' : c}\ \textrm{T-FU{\scriptsize PDATE}} \qquad \frac{\Gamma;\Delta \vdash c : [\!(T)\!]}{\Gamma;\Delta \vdash \mathsf{new}\,c : c}\ \textrm{T-N{\scriptsize EW}C}$$

$$\frac{\Gamma_1;\Delta_1 \vdash e : T_1 :: \Gamma_2;\Delta_2 \qquad \Gamma_2,x{:}T_1;\Delta_2 \vdash t : T_2 :: \Gamma_3;\Delta_3}{\Gamma_1;\Delta_1 \vdash \mathsf{let}\,x{:}T_1 = e\,\mathsf{in}\,t : T_2 :: \Gamma_3;\Delta_3}\ \textrm{T-L{\scriptsize ET}}$$

$$\frac{\begin{array}{c}\Gamma_1;\Delta_1 \vdash v_1 : T_1 \quad \Gamma_1;\Delta_1 \vdash v_2 : T_1\\ \Gamma_1;\Delta_1 \vdash e_1 : T_2 :: \Gamma_2;\Delta_2 \quad \Gamma_1;\Delta_1 \vdash e_2 : T_2 :: \Gamma_2;\Delta_2\end{array}}{\Gamma_1;\Delta_1 \vdash \mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_2 :: \Gamma_2;\Delta_2}\ \textrm{T-C{\scriptsize OND}}$$

$$\frac{\begin{array}{c}\Gamma_1;\Delta_1 \vdash v : c \quad \Gamma_1;\Delta_1 \vdash c : [\!(\ldots,l{:}\mathsf{Unit} \to T,\ldots)\!]\\ \Gamma_1;\Delta_1 \vdash e_1 : T_2 :: \Gamma_2;\Delta_2 \quad \Gamma_1;\Delta_1 \vdash e_2 : T_2 :: \Gamma_2;\Delta_2\end{array}}{\Gamma_1;\Delta_1 \vdash \mathsf{if}\ \mathsf{undef}(v.l())\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_2 :: \Gamma_2;\Delta_2}\ \textrm{T-C{\scriptsize OND}}_\perp$$

$$\frac{}{\Gamma;\Delta \vdash \mathsf{stop} : T}\ \textrm{T-S{\scriptsize TOP}} \qquad \frac{}{\Gamma;\Delta \vdash () : \mathsf{Unit}}\ \textrm{T-U{\scriptsize NIT}}$$

Figure 4.5: Typing (objects and threads)

$T.l$ denotes U, when $T = [\dots, l{:}U, \dots]$ and analogously for $T = [\![\dots, l{:}U, \dots]\!]$. Rules T-Class and T-Obj check the definition of classes resp., of objects against the respective interface type $[\![l_1{:}U_1, \dots, l_k{:}U_k]\!]$. Note that the type of the self-parameter must be identical to the name of the class, the method resides in. The premises of rule T-Memb check the method body in the context Γ appropriately extended with the formal parameters x_i, resp. the context Δ extended by the ς-bound self-parameter (s in the rule). T-Undef works similarly, treating the case of an uninitialized field. The terminated expression stop and the unit value do not change the capabilities (cf. rules T-Stop and T-Unit). Note that stop has any type (cf. rule T-Stop) reflecting the fact that control never reaches the point *after* stop. Further constructs without side-effects are the three expressions to manipulate the monitor locks (suspension, lock grabbing, and lock release), object instantiation (T-NewC), and field update. Wrt. field update in rule T-FUpdate, the reason why the update has no effect on the contexts is that we do not allow fields to carry a type of the form $[T]^{+-}$. This effectively prevents the passing around of write-permissions via fields. The rule T-Let for let-bindings introduces a local scope. The change from Δ_1 to Δ_2 and further from Δ_2 to Δ_3 (and analogously for the Γs) reflects the sequential evaluation strategy: first e is evaluated and afterwards t. For conditionals, both branches must agree on their pre- and post Δ-contexts, which typically means, over-approximating the effect by taking the upper bound on both as combined effect. Note that the comparison of the values in T-Cond resp. the check for definedness in T-Cond$_\perp$ has no side-effect on the contexts. The rule for testing for definedness using undef (not shown) works analogously.

In Figure 4.6 we define the typing rules to deal with futures, promises, and especially the linear aspect of consuming and transmitting the write-permissions. The claim-command fetches the result value from a future; hence, if the reference n is of type $[T]^+$, the value itself carries type T (cf. rule T-Claim). The rule T-Get for get works analogously.

The expression promise T creates a new promise, which can be read or written and is therefore of type $[T]^{+-}$. Note, however, that the context Δ does *not* change. The reason is that the new name created by promise is hidden by a ν-binder immediately after creation and thus does not immediately extend the Δ-context (see the reduction rule Prom below). The binding of a thread t to a promise n is well-typed if the type of n still allows the promise to be fulfilled, i.e., n is typed by $[T]^{+-}$ and not just $[T]^+$. The expression claim dereferences a future, i.e., it fetches a value of type T from the reference of type $[T]^+$. Otherwise, the expression has no effect on Δ, as reading can

$$\frac{}{\Gamma;\Delta \vdash \mathsf{promise}\ T : [T]^{+-}}\ \textsc{T-Prom}$$

$$\frac{\Gamma;\Delta \vdash n : [T]^{+} \qquad \Gamma;\Delta \vdash o{:}c}{\Gamma;\Delta \vdash \mathsf{claim}@(n,o) : T}\ \textsc{T-Claim} \qquad \frac{\Gamma;\Delta \vdash n : [T]^{+}}{\Gamma;\Delta \vdash \mathsf{get}@n : T}\ \textsc{T-Get}$$

$$\frac{\begin{array}{c}\Gamma;\Delta, n{:}[T]^{+} \vdash o : c \quad \Gamma;\Delta, n{:}[T]^{+} \vdash c : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!] \\ \Gamma;\Delta, n{:}[T]^{+} \vdash \vec{v} : \vec{T} \quad \acute{\Gamma};\acute{\Delta} = \Gamma;\Delta \setminus (\vec{v} : \vec{T})\end{array}}{\Gamma;\Delta, n : [T]^{+-} \vdash \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n : [T]^{+} :: \acute{\Gamma};\acute{\Delta}, n{:}[T]^{+}}\ \textsc{T-Bind}$$

$$\frac{\Gamma(x) = T \qquad \acute{\Gamma} = \Gamma \setminus x : T}{\Gamma;\Delta \vdash x : T :: \acute{\Gamma};\Delta}\ \textsc{T-Var} \qquad \frac{\Delta(x) = T \qquad \acute{\Delta} = \Delta \setminus n : T}{\Gamma;\Delta \vdash n : T :: \Gamma;\Delta'}\ \textsc{T-Name}$$

$$\frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{suspend}(o) : \mathsf{Unit}}\ \textsc{T-Suspend}$$

$$\frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{grab}(o) : \mathsf{Unit}}\ \textsc{T-Grab} \qquad \frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{release}(o) : \mathsf{Unit}}\ \textsc{T-Release}$$

$$\frac{\Gamma_1;\Delta_1 \vdash t : T :: \Gamma_2;\Delta_2 \quad T \leq T'}{\Gamma_1;\Delta_1 \vdash t : T' :: \Gamma_2;\Delta_2}\ \textsc{T-Sub}$$

Figure 4.6: Typing (objects and threads)

be done arbitrarily many times. Note that in rule T-Claim, the type of o is not checked, as by convention, the claim-statement must be used in the form claim@$(n, \textit{self})$ in the user syntax, where *self* is the self-parameter of the surrounding methods. Reduction then preserves well-typedness so a re-check here is not needed. Similar remarks apply to the remaining rules. The treatment of get is analogous (cf. rules T-Claim and T-Get). For T-Bind, handing over a promise with read-/write-permissions as an actual parameter of a method call, the caller loses the right to fulfill the promise. Of course, the caller can only pass the promise to a method which assumes read-/write-permissions, if itself has the write-permission. The loss of the write-permission is specified by setting $\acute{\Delta}$ and $\acute{\Gamma}$ to $\Delta \setminus \vec{v} : \vec{T}$ resp. to $\Gamma \setminus \vec{v} : \vec{T}$. The *difference*-operator $\Delta \setminus n : [T]^{+-}$ removes the *write*-permission for n from the context Δ. In T-Bind, the premise $\Gamma;\Delta, n{:}[T]^{+} \vdash \vec{v} : \vec{T}$ abbreviates the following: assume $\vec{v} = v_1, \ldots v_n$ and $\vec{T} = T_1 \ldots T_n$ and let Ξ_1 abbreviate $\Gamma;\Delta, n{:}[T]^{+}$. Then $\Xi \vdash \vec{v} : \vec{T}$ means: $\Xi_i \vdash v_i : T_i$ and $\Xi_{i+1} = \Xi_i \setminus T_i$, for all $1 \leq i \leq n$.

Note that checking the type of the callee has no side-effect on the bindings. Mentioning a variable or a name removes the write-permission (if present) from the respective binding context (cf. T-Var and T-Name). The next three rules T-Suspend, T-Grab, and T-Release deal with the expressions for coordination and lock handling; they are typed by Unit. The last rule T-Sub is the standard rule of subsumption.

4.2.3 Operational semantics

The operational semantics is given in two stages, component internal steps and external ones, where the latter describe the interaction at the interface. Section 4.2.3 starts with component-internal steps, i.e., those definable without reference to the environment. In particular, the steps have no externally observable effect. The external steps, presented afterwards in Section 4.2.3, define the interaction between component and environment. They are defined in reference to assumption and commitment contexts. The static part of the contexts corresponds to the static type system from Section 4.2.2 on component level and takes care that, e.g., only well-typed values are received from the environment.

Internal steps

The internal semantics describes the operational behavior of a *closed* system, not interacting with its environment. The corresponding reduction steps are shown in Figure 4.7, distinguishing between confluent steps $\rightsquigarrow$ and other internal transitions $\xrightarrow{\tau}$, both invisible at the interface. The $\rightsquigarrow$-steps, on the one hand, do not access the instance state of the objects. They are free of imperative side-effects and thus confluent. The $\xrightarrow{\tau}$-steps, on the other hand, access the instance state, either by reading or by writing it, and may thus lead to race conditions. In other words, this part of the reduction relation is in general not confluent.

The first seven rules deal with the basic sequential constructs, all as confluent steps. The basic evaluation mechanism is substitution (cf. rule Red). Note that the rule requires that the leading let-bound variable is replaced only by *values* v. The operational behavior of the two forms of conditionals are axiomatized by the four Cond-rules. Depending on the result of the comparison in the first pair of rules, resp., the result of checking for definedness in the second pair, either the then- or the else-branch is taken. In rule Cond$_2$, we assume that v_1 does not equal v_2, as side condition. Evaluating stop terminates the future for good, i.e., the rest of the thread will never be executed as there is no reduction rule for the future $n\langle\mathsf{stop}\rangle$ (cf. rule STOP). The rule FLOOKUP deals with field look-up, where $F'.l(o)()$ stands for $\bot_c[o/s] = \bot_c$, resp., for $v[o/s]$, where $[c, F', L] = [c, \ldots, l = \varsigma(s{:}c).\lambda().\bot_c, \ldots, L]$, if the field is yet undefined, resp., $[c, F', L] = [c, \ldots, l = \varsigma(s{:}c).\lambda().v, \ldots, L]$. In rule FUPDATE, the meta-mathematical notation $F.l := v$ stands for $(\ldots, l = v, \ldots)$, when $F = (\ldots, l = v', \ldots)$. There will be no external variant of the rule for field look-up later in the semantics of open systems, as we do not allow field access across component boundaries. The same restriction holds for field update in rule FUPDATE. A new object as instance of a given class is created by rule NEWO$_i$. Note that initially, the lock is free and there is no activity associated with the object, i.e., the object is initially passive.

The expression $\mathsf{promise}\,T$ creates a fresh promise n'. A new thread $n'\langle\bullet\rangle$ is allocated with an "undefined" body, as so far nothing more than the name is known. The rule PROM mentions the types T and T'. The typing system assures that the type T is of the form $[S]^{+-}$ for some type S. A promise is fulfilled by the bind-command (cf. rule BIND$_i$), in that the new thread n' is put together with the code to be executed and run in parallel with the rest. In the configuration after the reduction step, the meta-mathematical notation $M.l(o)(\vec{v})$ stands for $t[o/s][\vec{v}/\vec{x}]$, when the method suite $[M]$ equals $[\ldots, l = \varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t, \ldots]$.

Upon termination, the result is available via the claim- and the get-syntax (cf. the CLAIM-rules and rule GET$_i$), but not before the lock of the object is given back again using $\mathsf{release}(o)$ (cf. rule RELEASE). If the thread is not yet terminated, the requesting thread suspends itself, thereby giving up the lock. The behavior of claim is sketched in Figure 4.3. Note the types of the involved let-bound variables: the future reference is typed by $[T]$, indicating that the value for x will not directly be available, but must be dereferenced first via claim.

The two operations grab and release take, resp., give back the lock of an object. They are not part of the user syntax, i.e., the programmer cannot directly manipulate the monitor lock. The user can release the lock using the suspend-command or by trying to get back the result from a call using claim.

The above reduction relations are used modulo *structural congruence,* which captures the algebraic properties of parallel composition and the hiding operator. The basic axioms for $\equiv$ are shown in Figure 4.8 where in the fourth axiom, n does not occur free in C_1. The congruence relation is imported into the reduction relations in Figure 4.9. Note that all syntactic entities are always tacitly understood modulo α-conversion.

For illustration of the operational semantics, we show the combination of creating a promise and binding a method body to it. The steps in the reduction sequence below are justified by PROM, LET, and BIND, in that order. In the sequence, we did not write the definition of the object plus the class,

$n\langle \mathsf{let}\ x{:}T{=}v\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle t[v/x]\rangle$ RED

$n\langle \mathsf{let}\ x_2{:}T_2{=}(\mathsf{let}\ x_1{:}T_1{=}e_1\ \mathsf{in}\ e)\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle \mathsf{let}\ x_1{:}T_1{=}e_1\ \mathsf{in}\ (\mathsf{let}\ x_2{:}T_2{=}e\ \mathsf{in}\ t)\rangle$ LET

$n\langle \mathsf{let}\ x{:}T{=}(\mathsf{if}\ v{=}v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle \mathsf{let}\ x{:}T{=}e_1\ \mathsf{in}\ t\rangle$ COND$_1$

$n\langle \mathsf{let}\ x{:}T{=}(\mathsf{if}\ v_1{=}v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle \mathsf{let}\ x{:}T{=}e_2\ \mathsf{in}\ t\rangle$ where $(v_1{\neq}v_2)$ COND$_2$

$n\langle \mathsf{let}\ x{:}T{=}(\mathsf{if}\ \mathsf{undef}(\bot_{c'})\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle \mathsf{let}\ x{:}T{=}e_1\ \mathsf{in}\ t\rangle$ COND$_1^{\bot}$

$n\langle \mathsf{let}\ x{:}T{=}(\mathsf{if}\ \mathsf{undef}(v)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle \mathsf{let}\ x{:}T{=}e_2\ \mathsf{in}\ t\rangle$ COND$_2^{\bot}$

$n\langle \mathsf{let}\ x{:}T{=}\mathsf{stop}\ \mathsf{in}\ t\rangle \rightsquigarrow n\langle \mathsf{stop}\rangle$ STOP

$o[c,F,L] \| n\langle \mathsf{let}\ x{:}T{=}o.l()\ \mathsf{in}\ t\rangle \xrightarrow{\tau} o[c,F,L] \| n\langle \mathsf{let}\ x{:}T{=}F.l(o)()\ \mathsf{in}\ t\rangle$ FLOOKUP

$o[c,F,L] \| n\langle \mathsf{let}\ x{:}T{=}o.l{:=}v\ \mathsf{in}\ t\rangle \xrightarrow{\tau} o[c,F.l{:=}v,L] \| n\langle \mathsf{let}\ x{:}T{=}o\ \mathsf{in}\ t\rangle$ FUPDATE

$c[\![F,M]\!] \| n\langle \mathsf{let}\ x{:}c{=}\mathsf{new}\ c\ \mathsf{in}\ t\rangle \rightsquigarrow c[\![F,M]\!] \| \nu(o{:}c).(o[c,F,\bot] \| n\langle \mathsf{let}\ x{:}c{=}o\ \mathsf{in}\ t\rangle)$ NEWO$_i$

$n\langle \mathsf{let}\ x{:}T'{=}\mathsf{promise}\ T\ \mathsf{in}\ t\rangle \rightsquigarrow \nu(n'{:}T').(n\langle \mathsf{let}\ x{:}T'{=}n'\ \mathsf{in}\ t\rangle \| n'\langle\bullet\rangle)$ PROM

$c[\![F',M]\!] \| o[c,F,L] \| n_1\langle \mathsf{let}\ x{:}T{=}\mathsf{bind}\ o.l(\vec{v}){:}T_2{\hookrightarrow}n_2\ \mathsf{in}\ t_1\rangle \| n_2\langle\bullet\rangle \xrightarrow{\tau}$
$c[\![F',M]\!] \| o[c,F,L] \| n_1\langle \mathsf{let}\ x{:}T{=}n_2\ \mathsf{in}\ t_1\rangle \| n_2\langle \mathsf{let}\ x{:}T_2{=}\mathsf{grab}(o);M.l(o)(\vec{v})\ \mathsf{in}\ \mathsf{release}(o);x\rangle$ BIND$_i$

$n_1\langle v\rangle \| n_2\langle \mathsf{let}\ x{:}T{=}\mathsf{claim@}(n_1,o)\ \mathsf{in}\ t\rangle \rightsquigarrow n_1\langle v\rangle \| n_2\langle \mathsf{let}\ x{:}T{=}v\ \mathsf{in}\ t\rangle$ CLAIM$_i^1$

$$\frac{t_2{\neq}v}{n_2\langle t_2\rangle \| n_1\langle \mathsf{let}\ x{:}T{=}\mathsf{claim@}(n_2,o)\ \mathsf{in}\ t_1'\rangle \rightsquigarrow n_2\langle t_2\rangle \| n_1\langle \mathsf{let}\ x{:}T{=}\mathsf{release}(o);\mathsf{get@}n_2\ \mathsf{in}\ \mathsf{grab}(o);t_1'\rangle}\ \text{CLAIM}_i^2$$

$n_1\langle v\rangle \| n_2\langle \mathsf{let}\ x{:}T{=}\mathsf{get@}n_1\ \mathsf{in}\ t\rangle \rightsquigarrow n_1\langle v\rangle \| n_2\langle \mathsf{let}\ x{:}T{=}v\ \mathsf{in}\ t\rangle$ GET$_i$

$n\langle \mathsf{suspend}(o);t\rangle \rightsquigarrow n\langle \mathsf{release}(o);\mathsf{grab}(o);t\rangle$ SUSPEND

$o[c,F,\bot] \| n\langle \mathsf{grab}(o);t\rangle \xrightarrow{\tau} o[c,F,\top] \| n\langle t\rangle$ GRAB

$o[c,F,\top] \| n\langle \mathsf{release}(o);t\rangle \xrightarrow{\tau} o[c,F,\bot] \| n\langle t\rangle$ RELEASE

Figure 4.7: Internal steps

$$\mathbf{0} \parallel C \equiv C \qquad C_1 \parallel C_2 \equiv C_2 \parallel C_1 \qquad (C_1 \parallel C_2) \parallel C_3 \equiv C_1 \parallel (C_2 \parallel C_3)$$
$$C_1 \parallel \nu(n{:}T).C_2 \equiv \nu(n{:}T).(C_1 \parallel C_2) \qquad \nu(n_1{:}T_1).\nu(n_2{:}T_2).C \equiv \nu(n_2{:}T_2).\nu(n_1{:}T_1).C$$

Figure 4.8: Structural congruence

needed to do the last reduction step. I.e., the reduction sequence below runs in parallel with $c[\![F', M]\!] \parallel o[c, F, L]$, where in particular the method suite M, stored in the class c of the object o, contains the definition of the method body. That definition is needed for binding operation in the last reduction step. In the corresponding rule BIND$_i$, this is written as $M.l(o)(\vec{v})$. In the final configuration, t' contains the result of looking up the method body and is of the form $\mathsf{grab}(o); M.l(o)(\vec{v})$. The overall behavior of the fulfilled promise n_2, i.e., after the binding step, is: first acquire the lock of the object, afterwards execute the method body with the formal parameters including the self-parameter appropriately substituted. With the return value computed and remembered in z, the lock is released and the result is made available under the future reference n_2:

$$\begin{array}{lr}
n_1\langle \mathsf{let}\, x{:}[T]^{+-} = \mathsf{promise}\, T\, \mathsf{in}\ (\mathsf{let}\, y : T_2 = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow x\, \mathsf{in}\, t)\rangle & \rightsquigarrow \\
\nu(n_2{:}[T]^{+-}).(n_1\langle \mathsf{let}\, x{:}[T]^{+-} = n_2\, \mathsf{in}\ (\mathsf{let}\, y{:}[T]^{+} = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow x\, \mathsf{in}\, t)\rangle \parallel n_2\langle\bullet\rangle) & \rightsquigarrow \\
\nu(n_2{:}[T]^{+-}).(n_1\langle \mathsf{let}\, y{:}[T]^{+} = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n_2\, \mathsf{in}\, t[n_2/x]\rangle \parallel n_2\langle\bullet\rangle) & \xrightarrow{\tau} \\
\nu(n_2{:}[T]^{+-}).(n_1\langle \mathsf{let}\, y{:}[T]^{+} = n_2\, \mathsf{in}\, t[n_2/x]\rangle \parallel n_2\langle \mathsf{let}\, z{:}T = t'\, \mathsf{in}\, \mathsf{release}(o); z\rangle) &
\end{array}$$

Note that the overall behavior of first creating a promise and, in a next step, binding a method body to it, corresponds exactly to the behavior of an asynchronous method call. Asynchronous method calls can therefore be seen as syntactic sugar. The introduction of promises as a separate datatype and

$$\frac{C \equiv \rightsquigarrow \equiv C'}{C \rightsquigarrow C'} \qquad \frac{C \rightsquigarrow C'}{C \parallel C'' \rightsquigarrow C' \parallel C''} \qquad \frac{C \rightsquigarrow C'}{\nu(n{:}T).C \rightsquigarrow \nu(n{:}T).C'}$$
$$\frac{C \equiv \xrightarrow{\tau} \equiv C'}{C \xrightarrow{\tau} C'} \qquad \frac{C \xrightarrow{\tau} C'}{C \parallel C'' \xrightarrow{\tau} C' \parallel C''} \qquad \frac{C \xrightarrow{\tau} C'}{\nu(n{:}T).C \xrightarrow{\tau} \nu(n{:}T).C'}$$

Figure 4.9: Reduction modulo congruence

binding as corresponding, separate operation on promises therefore generalizes the setting with futures and asynchronous method calls, only.

In the following, we show that the type system indeed assures what it is supposed to, most importantly that a promise is indeed fulfilled only once. An important part of it is a standard property, namely preservation of well-typedness under internal reduction (subject reduction). First we characterize as erroneous situations where a promise is about to be written a second time: A configuration C contains a *write-error* if it is of the form $C \equiv \nu(\Theta').(C' \parallel n'\langle \mathsf{let}\, x : T = \mathsf{bind}\, t_1 : T_1 \hookrightarrow n\, \mathsf{in}\, t_2\rangle \parallel n\langle t\rangle)$. Configurations without such write-errors are called *write-error free*, denoted $\vdash C : ok$. In [82], an analogous condition is called *handle-error*.

The ancillary lemmas proceed in general by induction on the typing derivations for judgments of the form $\Delta \vdash C : \Theta$. From a proof-theoretical (and algorithmic) point of view, the type system as formalized in Figures 4.4, 4.5, and 4.6 has an unwelcome property: it is too "non-deterministic" in that it allows the non-structural subsumption rules T-SUB on the level of threads t and on the level of components C at any point in the derivation. This liberality is unwelcome for proofs by induction on the typing derivation as one loses knowledge about the structure of the premises of an applied rule in the derivation. We write $\Delta \vdash_m C : \Theta$ for derivations where subsumption at the level of components (by rule T-SUB from Figure 4.4) is *not* used, and subsumption from Figure 4.6 is only used "when needed", i.e., for adaptation. Taking for instance T-BIND and concentrating on the premises relevant for the illustration: Given as the interface type of the class $\Gamma; \Delta, n{:}[T]^+ \vdash_m c : [\![(\ldots, l{:}\vec{T} \to T, \ldots)]\!]$ and furthermore $\Gamma; \Delta, n{:}[T]^+ \vdash_m \vec{v} : \vec{S}$, the minimal types $\vec{S}$ of the $\vec{v}$ may not directly match the expected argument type $\vec{T}$ of the method labeled l (as is required in the premise of the rule T-BIND). Restricting now the use of subsumption to "*adapt*" the $\vec{S}$ to $\vec{T}$ gives the type system for minimal types (denoted by using $\vdash_m$ instead of $\vdash$). This could be explicitly done by removing the freely applicable T-SUB and distributing its effect into the premises of structural rules, where such adaptation is needed. In the discussed rule T-BIND, by stipulating

$$\frac{\ldots \qquad \Gamma; \Delta, n{:}[T]^+ \vdash_m c : [\![(\ldots, l{:}\vec{T} \to T, \ldots)]\!] \quad \Gamma; \Delta, n{:}[T]^+ \vdash_m \vec{v} : \vec{S} \quad \vec{S} \leq \vec{T}}{\Gamma; \Delta, n : [T]^{+-} \vdash_m \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n : [T]^+ :: \acute{\Gamma}; \acute{\Delta}, n{:}[T]^+} \text{ T-BIND}_m$$

where $\vec{S} \leq \vec{T}$ is interpreted pointwise $S_i \leq T_i$, for all i. As the formulation of that type system is rather standard and straightforward, we omit its definition.

Lemma 4.2.3 (Minimal typing) *1. If $\Delta \vdash_m C : \Theta$ and $\Delta' \vdash C : \Theta'$, then $\Delta \leq \Delta'$ and $\Theta' \leq \Theta$.*

2. If $\Delta \vdash_m C : \Theta$ then $\Delta \vdash C : \Theta$.

3. If $\Delta' \vdash C : \Theta'$, then $\Delta \vdash_m C : \Theta$ with $\Delta \leq \Delta'$ and $\Theta' \leq \Theta$, for some Δ and Θ.

Proof: Straightforward. □

First we show that a well-typed component does not contain a manifest write-error.

Lemma 4.2.4 *If $\Delta \vdash_m C : \Theta$, then $\vdash C : ok$.*

Proof: By induction on the typing derivations for judgments on the level of components, i.e., for judgments of the form $\Delta \vdash C : \Theta$; the subordinate typing rules from Figures 4.5 and 4.6 on the level of threads and expressions do not play a role for the proof. The empty component in rule T-Empty, one of two base cases, is clearly write-error free. So is the unfulfilled promise of rule T-NThread, the other base case. The cases for the rule T-Nu is proved by straightforward induction. The cases for rules T-NClass, T-NObj, and T-NThread are trivially satisfied, as they mention a single, basic component, only.
Case: Rule T-Par
We are given $\Delta_1, \Theta_2 \vdash C_1 : \Theta_1$ and $\Delta_2, \Theta_1 \vdash C_2 : \Theta_2$ with $\Delta = \Delta_1 \oplus \Delta_2$. By induction, both C_1 and C_2 are write-error free. The non-trivial case (which we lead to a contradiction) is when one of the components attempts to write to a promise and the partner already has fulfilled it. So, without loss of generality assume that $C_1 = \nu(\Theta'_1).(C'_1 \parallel n_1\langle \mathsf{let}\, x : T = \mathsf{bind}\, x : T \hookrightarrow n_2 \,\mathsf{in}\, t''\rangle$ and $C_2 = \nu(\Theta'_2).(C'_2 \parallel n_2\langle t_2\rangle)$. Assume that n_2 occurs in neither Θ'_1 nor Θ'_2, otherwise no write-error is present (since in that case, the name n_2 mentioned on both sides of the parallel refers to different entities). For C_1 to be well-typed, we have $\Delta_1, \Theta_2 \vdash n_2 : [T_2]^{+-}$ for some type T_2. For C_2 to be well-typed, we have $\Theta_2 \vdash n : [T_2]^{+}$ for some type T_2. Thus, $\Delta \vdash C_1 \parallel C_2 : \Theta_1, \Theta_2$ cannot be derived, which contradicts the assumption. □

Lemma 4.2.5 (Subject reduction: $\equiv$) *If $\Delta \vdash_m C_1 : \Theta$ and $C_1 \equiv C_2$, then $\Delta \vdash_m C_2 : \Theta$.*

Proof: We show preservation of typing by the axioms of Figure 4.8. Proceed by induction on the derivation of $\Delta \vdash_m C_1 : \Theta$.

Case: $C \parallel \mathbf{0} \equiv C$ (idempotence)
We are given $\Delta \vdash C \parallel \mathbf{0} : \Theta$. Inverting rule T-Par and by rule T-Empty we get as sub-goals $\Delta, \Theta \vdash_m \mathbf{0} : ()$ and $\Delta \vdash_m C : \Theta$, which concludes the case.

Case: $C \equiv C \parallel \mathbf{0}$ (idempotence)
Immediate using rules T-Par and T-Empty.

Case: $C_1 \parallel C_2 \equiv C_2 \parallel C_1$ (commutativity)
Immediate.

Case: $C_1 \parallel (C_2 \parallel C_3) \equiv (C_1 \parallel C_2) \parallel C_2$ and vice versa (associativity)
By straightforward induction.

Case: $C_1 \parallel \nu(n{:}T).C_2 \equiv \nu(n{:}T).(C_1 \parallel C_2)$
where n does not occur free in C_1. We are given $\Delta \vdash C_1 \parallel \nu(n{:}T).C_2 : \Theta_1, \Theta_2$, where n occurs in neither Θ_1 nor Θ_2. Inverting rules T-Par and T-Nu, we obtain as two subgoals $\Delta, \Theta_2 \vdash C_1 : \Theta_1$ and $\Delta, \Theta_1 \vdash C_2 : \Theta_1, \Theta_2, n{:}T$, and the result follows by rules T-Par and T-Nu.

Case: $\nu(n_1{:}T_1).\nu(n_2{:}T_2).C \equiv \nu(n_2{:}T_2).\nu(n_1{:}T_1).C$
Analogously. □

The next lemma is another step towards subject reduction. Note that minimal types are *not* preserved by reduction. Especially executing a bind-operation with rule Bind$_i$ changes the type of the corresponding name from $[T]^{+-}$ to $[T]^{+}$.

Lemma 4.2.6 (Subject reduction: $\xrightarrow{\tau}$ and $\rightsquigarrow$) *Assume* $\Delta \vdash_m C : \Theta$.

1. *If* $C \xrightarrow{\tau} \acute{C}$*, then* $\Delta \vdash \acute{C} : \Theta$.

2. *If* $C \rightsquigarrow \acute{C}$*, then* $\Delta \vdash \acute{C} : \Theta$.

Proof: The reduction rules of Figure 4.7 are all of the form $C_1 \parallel n\langle t_1\rangle \xrightarrow{\tau} C_2 \parallel n\langle t_2\rangle$, where often $C_1 = C_2$ or C_1 and C_2 missing. In the latter case, it suffices to show that $;\Delta, n{:}[T]^{+} \vdash_m t_1 : T$ implies $;\Delta, n{:}[T]^{+} \vdash t_2 : T$.

Case: Rule Red: $n\langle \mathsf{let}\, x : T = v \,\mathsf{in}\, t\rangle \rightsquigarrow n\langle t[v/x]\rangle$
By preservation of typing under substitution.

The five rules for let and for conditionals are straightforward. The case for stop follows from the fact that stop has every type (cf. rule T-Stop).

Case: Rule Prom: $n\langle \mathsf{let}\, x{:}T' = \mathsf{promise}\, T\, \mathsf{in}\, t\rangle \rightsquigarrow \nu(n'{:}T').(n\langle \mathsf{let}\, x : T' = n'\, \mathsf{in}\, t\rangle \parallel n'\langle\bullet\rangle)$
The type system (for minimal types) assures that $T' = [T]^{+-}$, i.e., for the left-hand side of the reduction step, we obtain as one subgoal (inverting rules T-NThread$'$, T-Let, and T-Prom) $x{:}[T]^{+-}; \Delta, n{:}[S]^{+} \vdash t : S$. The result follows from rules T-Nu, T-Par, T-Let, and T-NThread$'$ (and weakening):

$$\dfrac{\dfrac{\dfrac{\dfrac{\ldots \qquad x{:}[T]^{+-}; \Delta, n'{:}[T]^{+-}, n{:}[S]^{+} \vdash t : S}{; \Delta, n'{:}[T]^{+-}, n{:}[S]^{+} \vdash \mathsf{let}\, x : [T]^{+-} = n'\, \mathsf{in}\, t : S}}{\Delta, n'{:}[T]^{+-} \vdash n\langle \mathsf{let}\, x : [T]^{+-} = n'\, \mathsf{in}\, t\rangle : n{:}[S]^{+}} \quad \dfrac{}{\Delta, n{:}[S]^{+}, n'{:}[T]^{+} \vdash n'\langle\bullet\rangle : n'{:}[T]^{+-}}}{\Delta \vdash n\langle \mathsf{let}\, x : [T]^{+-} = n'\, \mathsf{in}\, t\rangle \parallel n'\langle\bullet\rangle : (n{:}[S]^{+}, n'{:}[T]^{+-})}\,\text{T-Par}}{\Delta \vdash \nu(n'{:}[T]^{+-}).(n\langle \mathsf{let}\, x : T' = n'\, \mathsf{in}\, t\rangle \parallel n'\langle\bullet\rangle) : (n{:}[S]^{+})}\,\text{T-Nu}$$

Case: Rule Bind$_i$ $n_1\langle t\rangle \parallel n_2\langle\bullet\rangle = n_1\langle \mathsf{let}\, x{:}T = \mathsf{bind}\, o.l(\vec{v}) : T_2 \hookrightarrow n_2\, \mathsf{in}\, t_1\rangle \parallel n_2\langle\bullet\rangle \xrightarrow{\tau} n_1\langle \mathsf{let}\, x{:}T = n_2\, \mathsf{in}\, t_1\rangle \parallel n_2\langle \mathsf{let}\, x{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); x\rangle$
The type system assures (cf. rule T-Bind) that $T = [T_2]^{+}$. By assumption, we are given $\Delta \vdash n_1\langle t\rangle : \Theta$, which implies $\Theta = n_1{:}[T_1]^{+}$ for some type T_1. Inverting rules T-Par, T-NThread, T-Let, and T-Bind gives for the named thread n_1:

$$\dfrac{\dfrac{\dfrac{; \Delta_1, n_2{:}[T_2]^{+}, n_1{:}[T_1]^{+} \vdash \vec{v} : \vec{T} \quad \Delta'' = \Delta' \setminus (\vec{v}{:}\vec{T}) \quad \ldots}{; \Delta_1, n_2{:}[T_2]^{+-} \vdash \mathsf{bind}\, o.l(\vec{v}) : T_2 \hookrightarrow n_2 : T :: \; ; \Delta_1'', n_2{:}[T_2]^{+}}\,\text{T-Bind} \qquad x{:}[T_2]^{+}; \Delta_1'', n_2{:}[T_2]^{+} \vdash t_1 : T_1 :: x{:}[T_2]^{+}; \acute{\Delta}_2, n_2{:}[T_2]^{+}}{; \Delta_1, n_2{:}[T_2]^{+-}, n_1{:}[T_1]^{+} \vdash \mathsf{let}\, x{:}[T_2]^{+} = \mathsf{bind}\, o.l(\vec{v}) : T_2 \hookrightarrow n_2\, \mathsf{in}\, t_1 : T_1 :: \; ; \acute{\Delta}_1, n_2{:}[T_2]^{+}, n_1{:}[T_1]^{+}}}{\Delta_1, n_2{:}[T_2]^{+-} \vdash n_1\langle \mathsf{let}\, x{:}[T_2]^{+} = \mathsf{bind}\, o.l(\vec{v}) : T_2 \hookrightarrow n_2\, \mathsf{in}\, t_1\rangle : n_1{:}[T_1]^{+}}$$

Rule T-Bind (and T-NThread) implies that the assumption context Δ contains especially the binding $n_2{:}[T]^{+-}$, i.e., the assumption Δ in the last conclusion is of the form $\Delta', n_2{:}[T_2]^{+-}$.

Now to the post-configuration after the $\xrightarrow{\tau}$-step. With rule T-Par we obtain the following two sub-goals:

$$\dfrac{\Delta, n_2{:}[T_2]^{+-} \vdash n_1\langle \mathsf{let}\, x{:}T = n_2\, \mathsf{in}\, t_1\rangle : n_1{:}[T_1]^{+} \Delta, n_1{:}[T_1]^{+} \vdash n_2\langle \mathsf{let}\, x{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); x\rangle : n_2{:}[T_2]^{+-}}{\Delta \vdash n_1\langle \mathsf{let}\, x{:}T = n_2\, \mathsf{in}\, t_1\rangle \parallel n_2\langle \mathsf{let}\, x{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); x\rangle : n_1{:}[T_1]^{+}, n_2{:}[T_2]^{+-}}$$

The left one is derived using rules T-NThread, T-Let, and T-Name, where there second premise of rule T-Let is discharged by the corresponding assumption from above and weakening.

$$\dfrac{\dfrac{\dfrac{}{\Delta, n_2{:}[T_2]^+, n_1{:}[T_2]^+ \vdash n_2 : [T_2]^+}\ \textsc{T-Name} \qquad x{:}[T_2]^+, \Delta; n_2{:}[T_2]^+, n_1{:}[T_2]^+ \vdash t_1 : T_1}{\Delta, n_2{:}[T_2]^+, n_1{:}[T_2]^+ \vdash \mathsf{let}\, x{:}T = n_2\, \mathsf{in}\, t_1 : T_1}\ \textsc{T-Let}}{\Delta, n_2{:}[T_2]^+ \vdash n_1 \langle \mathsf{let}\, x{:}[T_2]^+ = n_2\, \mathsf{in}\, t_1 \rangle : n_1{:}[T_1]^+}$$

The second premise can be derived as follows:

$$\dfrac{\dfrac{\dfrac{\dots \qquad \dfrac{; \Delta, n_1{:}[T_1]^+, n_2{:}[T_2]^+ \vdash M.l(o)(\vec{v}) : T_2 \qquad \dfrac{\dfrac{}{y{:}T_2; \Delta, n_1{:}[T_1]^+, n_2{:}[T_2]^+ \vdash y : T_2}\ \textsc{T-Var}}{y{:}T_2; \Delta, n_1{:}[T_1]^+, n_2{:}[T_2]^+ \vdash \mathsf{release}(o); y : T_2}}{; \Delta, n_1{:}[T_1]^+, n_2{:}[T_2]^+ \vdash \mathsf{let}\, y{:}T_2 = M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); y : T_2}\ \textsc{T-Let}}{; \Delta, n_1{:}[T_1]^+, n_2{:}[T_2]^+ \vdash \mathsf{let}\, y{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); y : T_2}}{\Delta, n_1{:}[T_1]^+ \vdash n_2 \langle \mathsf{let}\, y{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); y \rangle : n_2{:}[T_2]^+}\ \textsc{T-NThread}}{\Delta, n_1{:}[T_1]^+ \vdash n_2 \langle \mathsf{let}\, y{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); y \rangle : n_2{:}[T_2]^{+-}}\ \textsc{T-Sub}$$

The premise $; \Delta, n_1{:}[T_1]^+, n_2{:}[T_2]^+ \vdash M.l(o)(\vec{v}) : T_2$ follows by preservation of typing by substitution. Note the use of subsumption in the last step. □

Lemma 4.2.7 (Subject reduction: $\equiv$) *If $\Delta \vdash C_1 : \Theta$ and $C_1 \equiv C_2$, then $\Delta \vdash C_2 : \Theta$.*

Proof: Assume $\Delta \vdash C_1 : \Theta$ and $C_1 \equiv C_2$. By Lemma 4.2.3(3), $\Delta' \vdash_m C_1 : \Theta'$ s.t. $\Delta \leq \Delta'$ and $\Theta' \leq \Theta$. By Lemma 4.2.5, $\Delta' \vdash_m C_2 : \Theta'$, and hence by Lemma 4.2.3(2), also $\Delta' \vdash C_2 : \Theta'$, and the result follows by subsumption (rule T-Sub). □

Lemma 4.2.8 (Subject reduction: $\xrightarrow{\tau}$ and $\rightsquigarrow$) *Assume $\Delta \vdash C : \Theta$.*

1. *If $C \xrightarrow{\tau} \acute{C}$, then $\Delta \vdash \acute{C} : \Theta$.*
2. *If $C \rightsquigarrow \acute{C}$, then $\Delta \vdash \acute{C} : \Theta$.*

Proof: As consequence of the corresponding property for minimal typing from Lemma 4.2.6, Lemma 4.2.3, and subsumption. □

Lemma 4.2.9 (Subject reduction) *If $\Delta \vdash C : \Theta$ and $C \Longrightarrow \acute{C}$, then $\Delta \vdash \acute{C} : \Theta$.*

Proof: A consequence of Lemma 4.2.7 and 4.2.8. □

A direct consequence is that all reachable configurations are write-error free:

Corollary 4.2.10 *If $\Delta \vdash C : \Theta$ and $C \Longrightarrow \acute{C}$, then $\vdash \acute{C} : ok$.*

Proof: A consequence of Lemma 4.2.4 and subject reduction from Lemma 4.2.9. □

External semantics

In this section we introduce the external semantics that defines the interaction between component and environment. We start by formalizing typing judgments and transitions between typing judgments, being the basic form of the external steps. We continue with static typing assumptions for well-formed and well-typed labels. Context updates, given next, express the dynamic change of typing judgments for incoming and outgoing communications. Making use of the above formalisms, we give the steps of the external semantics.

The external semantics formalizes the interaction of an open component with its environment. The semantics is given as labeled transitions between typing judgments on the level of components (cf. Figure 4.4), i.e., judgments of the form

$$\Delta \vdash C : \Theta, \tag{4.3}$$

where, as before, Δ represents the assumptions about the environment of the component C and Θ the commitments. The assumptions require the existence of named entities in the environment (plus giving static typing information), and dually, the commitment promises the existence of such entities in C. It is an invariant of the semantics, that the assumption and commitment contexts are disjoint concerning their name bindings. In addition, the interface keeps information about whether the value of a future n is already known at the interface (this information is not needed in the static type system of Figure 4.4). If it is, we write $n{:}T = v$ as binding of the context. We write furthermore $\Delta \vdash n = v$, if Δ contains the corresponding value information (and if not interested in the type) and write $\Delta \vdash n = \bot$, if that is not the case. This extension makes the value of a future (once successfully claimed) available at the interface. With these judgments, the external transitions are of the form:

$$\Delta \vdash C : \Theta \quad \xrightarrow{a} \quad \acute{\Delta} \vdash \acute{C} : \acute{\Theta}\ . \tag{4.4}$$

$$\begin{array}{lcll}
\gamma & ::= & n\langle call\ o.l(\vec{v})\rangle \mid n\langle get(v)\rangle \mid \nu(n{:}T).\gamma & \text{basic labels} \\
a & ::= & \gamma? \mid \gamma! & \text{receive and send labels}
\end{array}$$

Figure 4.10: Labels

Notation 4.2.11 *We abbreviate the tuple of name contexts Δ, Θ as Ξ. Furthermore we understand $\acute{\Delta}, \acute{\Theta}$ as $\acute{\Xi}$, etc.*

The labels of the external transitions represent single steps of the interface interactions (cf. Figure 4.10). A component exchanges information with the environment via *call*- and *get*-labels (by convention, referred to as γ_c and γ_g, for short). Interaction is either incoming or outgoing, indicated by ?, resp., !. In the labels, n is the identifier of the thread (i.e., also future/promise) carrying out the call resp. of being queried via claim or get. Besides that, object and future names (but no class names) may appear as arguments in the communication. Scope extrusion of names across the interface is indicated by the ν-binder. Given a basic label $\gamma = \nu(\Xi).\gamma'$ where Ξ is a name context such that $\nu(\Xi)$ abbreviates a sequence of single $n{:}T$ bindings (whose names are assumed all disjoint, as usual) and where γ' does not contain any binders, we call γ' the *core* of the label and refer to it by $\lfloor\gamma\rfloor$. We define the core analogously for receive and send labels. The free names $fn(a)$ and the bound names $bn(a)$ of a label a are defined as usual, whereas $names(a)$ refer to all names of a. In addition, we distinguish between names occurring as arguments of a label, in *passive* position, and the name occurring as carrier of the activity, in *active* position. Name n, for illustration, occurs actively and free in $n\langle call\ o.l.(\vec{v})\rangle$ and in $n\langle get(v)\rangle$. We write $fn_a(a)$ for the free names occurring in active position, $fn_p(a)$ for the free names in passive position, etc. All notations are used analogously for basic labels γ. Note that for incoming labels, Ξ contains only bindings to environment objects (besides future names), as the environment cannot create component objects; dually for outgoing communication.

The steps of the operational semantics for open systems check the *static* assumptions, i.e., whether at most the names actually occurring in the core of the label are mentioned in the ν-binders of the label, and whether the transmitted values are of the correct types. This is covered in the following definition.

Definition 4.2.12 (Well-formedness and well-typedness) *A label $a = \nu(\Xi).\lfloor a\rfloor$ is* well-formed, *written $\vdash a$, if $dom(\Xi) \subseteq names(\lfloor a\rfloor)$ and if Ξ is* *a well-formed name-context for object and future names, i.e., no name bound in* Ξ *occurs twice. The assertion*

$$\acute{\Xi} \vdash o.l? : \vec{T} \to T \tag{4.5}$$

("an incoming call of the method labeled l *in object* o *expects arguments of type* $\vec{T}$ *and results in a value of type* T*") is given by the following rule, i.e., implication:*

$$\frac{;\acute{\Theta} \vdash o : c \qquad ;\acute{\Xi} \vdash c : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!]}{\acute{\Xi} \vdash o.l? : \vec{T} \to T} \tag{4.6}$$

For outgoing calls, $\acute{\Xi} \vdash o.l! : \vec{T} \to T$ *is defined dually. In particular, in the first premise,* $\acute{\Theta}$ *is replaced by* $\acute{\Delta}$. Well-typedness *of an incoming core label* a *with expected type* $\vec{T}$*, resp.,* T*, and relative to the name context* $\acute{\Xi}$ *is asserted by*

$$\acute{\Xi} \vdash a : \vec{T} \to \_ \quad \textit{resp.,} \quad \acute{\Xi} \vdash a : \_ \to T\ , \tag{4.7}$$

as given by Figure 4.11. Finally, let $\acute{\Xi}_0$ *abbreviate* $;\acute{\Xi}$*. Then* $;\acute{\Xi} \vdash \vec{v} : \vec{T}$ *means:* $\acute{\Xi}_i \vdash v_i : T_i$ *and* $\acute{\Xi}_{i+1} = \acute{\Xi}_i \setminus T_i$*, for all* $0 \leq i \leq n-1$.

Note that the receiver o of the call is checked using only the commitment context $\acute{\Theta}$, to assure that o is a component object. Note further that to check the interface type of the class c, the full $\acute{\Xi}$ is consulted, since the argument types $\vec{T}$ or the result type T may refer to both component and environment classes.

The premise $;\acute{\Xi} \vdash \vec{v} : \vec{T}$ in rule LT-CallI is interpreted in such a way that checking for write-permission *consumes* that permission (analogous to the corresponding premise of rule T-Bind in Figure 4.6). This is formalized in the definition of $;\Xi \vdash \vec{v} : \vec{T}$ for well-typedness of a sequence of values, given at the end of Definition 4.2.12, which iterates through the sequence, potentially removing write-permission for a v_i s.t. the permission is no longer available for type cheking the rest of the sequence.

In a similar spirit: requiring that $\acute{\Xi}$ is of the form $\acute{\Xi}_1, n{:}[T]^+, \acute{\Xi}_2$ assures that it is not possible to transmit n with write-permissions if n is the active thread of the label.

Besides *checking* whether the assumptions are met before a transition, the contexts are *updated* by a transition step, especially extended by the new names, whose scope extrudes. For the binding part Ξ' of a label $\nu(\Xi').\gamma$, the scope of the references to existing objects and thread names Δ' extrudes across

$$\frac{\acute{\Xi} = \acute{\Xi}_1, n{:}[T]^+, \acute{\Xi}_2 \qquad ; \acute{\Xi} \vdash \vec{v} : \vec{T} \qquad a = n\langle call\ o_r.l(\vec{v})\rangle?}{\acute{\Xi} \vdash a :\ \vec{T} \to \_}\ \text{LT-CALLI}$$

$$\frac{; \acute{\Xi} \vdash v : T \qquad a = n\langle get(v)\rangle?}{\acute{\Xi} \vdash a : \_ \to T}\ \text{LT-GETI}$$

Figure 4.11: Typechecking labels

the border. In the step, Δ' extends the assumption context Δ and Θ' the commitment context Θ. Besides information about new names, the context information is potentially updated wrt. the availability of a future *value.* This is done when a *get*-label is exchanged at the interface for the first time, i.e., when a future value is claimed successfully for the first time. For outgoing communication, the situation is dual.

Before we come to the corresponding Definition 4.2.13 below, we make clear (again) the interpretation of judgments $\Delta \vdash C : \Theta$. Interesting is in particular the information $n{:}[T]^{+-}$, stipulating that name n is available with write-permission (and result type T). In case of $\Delta \vdash n : [T]^{+-}$, the name n is assumed to be available in the environment as writable, and conversely $\Theta \vdash n : [T]^{+-}$ asserts write-permission for the component. Since read-permissions, captured by types $[T]^+$, are not treated linearly —one is allowed to read from a future reference as many times as wished— the treatment of bindings $n{:}[T]^+$ is simpler. Hence, we concentrate here on $n{:}[T]^{+-}$ and the write-permissions.

As the domains of Δ and Θ are disjoint, bindings $n{:}T'$ cannot be available in the assumption context Δ and the commitments Θ at the same time. The information $T' = [T]^{+-}$ indicates which side, component or environment, has the write-permission. If, e.g., $\Delta \vdash n : [T]^{+-}$, then the component is not allowed to execute a bind on reference n. In the mentioned situation, the component can execute a claim-operation on n. The same applies if $\Delta \vdash n : [T]^+$. In other words, a name n can be accessed by reading by both the environment and the component once known at the interface, independent from whether it is part of Δ or of Θ. A difference between bindings of the form $n{:}[T]^{+-}$ and $n{:}[T]^+$ (and likewise $n{:}[T]^+ = v$) is, that communication can *change* $\Delta \vdash n : [T]^{+-}$ to $\Theta \vdash n : [T]^{+-}$ and vice versa. For names n of type $[T]^+$, this change of side is impossible. The latter kind of information, for instance $\Theta \vdash n : [T]^+$, implies that the code has been bound to n and it is placed in the component. Once fixed there, the reference to n may, of course, be passed around, but the thread named n itself cannot change to the environment since the language does not support *mobile* code.

Now, how do interface interactions update the contexts? We distinguish two ways, the name n can be transmitted in a label: *passively,* when transported as the argument of a call or a *get*-interaction, and *actively,* when mentioned as the carrier of the activity, as the n in $n\langle \mathit{call}\ o.l(\vec{v})\rangle$ and $n\langle \mathit{get}(v)\rangle$. As usual, such references (actively or passively) can be transmitted as fresh names, i.e., under a ν-binder, or alternatively as an already known name. When transmitted *passively* and typed with $[T]^{+-}$ for some type T, the write-permission to n is handed over to the receiving side and at the same time, that permission is removed from the sender side. So if, e.g., the environment is assumed to possess the write-permission for reference n, witnessed by $\Delta \vdash n : [T]^{+-}$, then sending n as argument in a communication to the component removes the binding from the environment and adds the permission to the component side, yielding $\Theta \vdash n : [T]^{+-}$.

Now, what about transmitting n *actively?* An incoming call $n\langle \mathit{call}\ o.l(\vec{v})\rangle?$, e.g., reveals at the interface that the promise indeed has been fulfilled. As, in that situation of an incoming call, the thread, executing the call, is located at the component, the commitment context is updated to satisfy $\Theta \vdash n : [T]^{+} = \bot$ (for an appropriate type T) after the communication. Indeed, before the step it is checked, that the environment actually has write-permission for n, i.e., that $\Delta \vdash n : [T]^{+-}$, or that the name n is new. See the incoming call in Figure 4.12(a), where the n is fresh, resp. in Figure 4.12(c), where the n has been transmitted passively and with write-permissions to the environment before the call (in the dotted arrow).

Whereas call-labels make public, at which side the thread in question resides, *get*-labels, on the other hand, reveal that the computation has terminated and fix the result value (if the information about the result value had not been public interface information before). There are two situations, where a, say, outgoing *get*-communication is possible. In both cases, the named thread, representing the future, resides in the component and after the *get*-communication, the value is determined, i.e., $\Theta \vdash n : [T]^{+} = v$ (if not already before the step). One scenario is that $\Delta \vdash n : [T]^{+} = \bot$ before the step still. If, in that situation, the get is executed by the environment, it is required that the component must have had write-permission before, i.e., $\Theta \vdash n : [T]^{+-}$ (cf. Figure 4.12(b)). The only way, the value for n is available for the environment now is that the promise had been fulfilled and the corresponding thread already has terminated, and this could have been done by the component, only. In that situation, the contexts are updated from $\Theta \vdash n : [T]^{+-}$ to $\Theta \vdash n : [T]^{+} = v$

by the *get*-interaction. Alternatively, the thread may be known to be part of the component with the promise already fulfilled ($\Theta \vdash n : [T]^+ = \bot$, as shown in Figures 4.12(a) and 4.12(c)). Finally, the value for n might already been known at the interface, i.e., already before the step, $\Theta \vdash n : [T]^+ = v$ holds. In that situation, v has been added as interface information previously by a prior *get*-interaction, and the situation corresponds to the very last get in Figures 4.12(b) and 4.12(c).

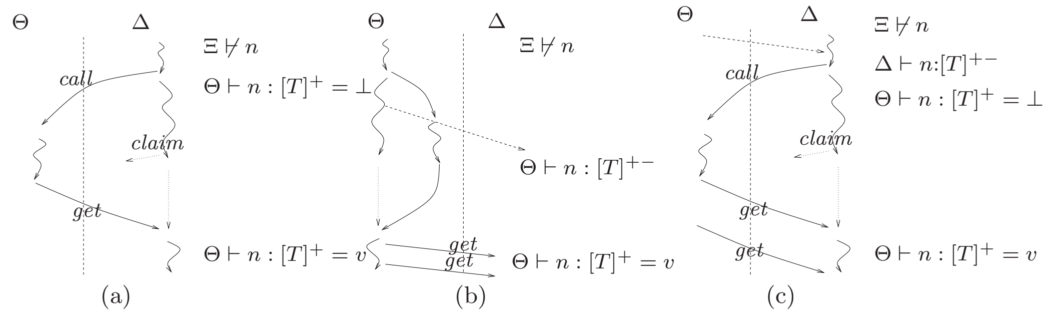
Θ
Δ
$\Xi \not\vdash n$
call
$\Theta \vdash n : [T]^+ = \bot$
claim
get
$\Theta \vdash n : [T]^+ = v$
(a)
Θ
Δ
$\Xi \not\vdash n$
$\Theta \vdash n : [T]^{+-}$
get
get
$\Theta \vdash n : [T]^+ = v$
(b)
Θ
Δ
$\Xi \not\vdash n$
$\Delta \vdash n{:}[T]^{+-}$
$\Theta \vdash n : [T]^+ = \bot$
call
claim
get
get
$\Theta \vdash n : [T]^+ = v$
(c)

Figure 4.12: Scenarios

Definition 4.2.13 (Context update) *Let Ξ be a name context and $a = \nu(\Xi').\lfloor a \rfloor$ an* incoming *label. Let $\acute{\Xi} = \Xi + a$ be defined as follows.*

We define the (intermediate) contexts $\Theta'' = \Theta$ and $\Delta'' = \Delta, \Xi'$. Let furthermore Σ'' be the set of bindings defined as follows. In case of a call-label, i.e., $\lfloor a \rfloor = n\langle call\ o.l(\vec{v})\rangle?$, let the vector of types $\vec{T}$ be defined by $\Xi \vdash o.l? : \vec{T} \to T$ according to Equation (4.5) of Definition 4.2.12. Then Σ'' consists of bindings of the form $v_i{:}[T_i']^{+-}$ for values v_i from $\vec{v}$ such that $T_i = [T_i']^{+-}$. In case of a get*-label, i.e., $\lfloor a \rfloor = n\langle get(v)\rangle?$, the context Σ'' is $v{:}[T]^{+-}$ if $\Delta'' \vdash n : [[T]^{+-}]^+$, and empty otherwise.*

With Σ'' given this way, the definitions of the post-contexts $\acute{\Delta}$ and $\acute{\Theta}$ distinguish between calls and get*-interaction: If a is a call-label and $n \in names_a(a)$, we define*

$$\acute{\Delta} = (\Delta'' \setminus \Sigma'') \setminus n{:}[T]^{+-} \quad \textit{and} \quad \acute{\Theta} = \Theta'', \Sigma'', n{:}[T]^+ \ . \tag{4.8}$$

If a is a get*-label $a = \nu(\Xi').n\langle get(v)\rangle?$ and $n \in names_a(a)$, $\acute{\Delta}$ and $\acute{\Theta}$ are given by:*

$$\acute{\Delta} = (\Delta'' \setminus \Sigma''), n{:}[T]^+ = v \quad \textit{and} \quad \acute{\Theta} = \Theta'', \Sigma'' \ . \tag{4.9}$$

For outgoing *communication, the definition is applied dually.*

The definition proceeds in two stages. In a first step, the assumption context Δ is extended with the bindings Ξ' carried with the incoming label a. Note that the bindings $\Xi' \vdash n : [T]^{+-}$ or $\Xi' \vdash n : [T]^{+}$ for promise/future references, kept in Σ', are added to the assumption context Δ but not the commitment context (in the considered case of incoming communication). The second step deals with the write-permissions, i.e., it transfers the write-permission transmitted from the sender to the receiver. The binding context Σ'' deals with the permissions carried by thread names transmitted passively, i.e., as arguments of the communication. It remains to take care also of the information carried by the active thread. There, we distinguish calls and *get*-labels. An incoming call (cf. Equation (4.8)) with n as active thread is the sign that the thread is now located at the component side and that the write-permission has been consumed by the environment. Hence, in Equation (4.8), the environment loses the write-permission and the component is extended by the binding $n{:}[T]^{+}$. In case of an incoming get, the transmitted value v is remembered as part of Δ (cf. Equation (4.8)).

The previous definition deals with the change of context information by communication. Apart from that, unfulfilled promises of the form $n\langle\bullet\rangle$ also change side, if their name is exchanged together with write-permission. As notation, we use $C(\Xi)$ to denote the component $n_1\langle\bullet\rangle \parallel \ldots \parallel n_k\langle\bullet\rangle$, where the names n_i correspond to all names of the context Ξ mentioned as $n_i : [T_i]^{+-}$ for some type T_i.

Now to the interface behavior, given by the external steps of Figure 4.13. Corresponding to the labels from Figure 4.10, there are a number of rules for external communication: either incoming or outgoing calls, resp., exchange of *get*-labels. Most rules have some premises in common. In all cases of a labeled transition, the context Ξ is updated to $\acute{\Xi} = \Xi + a$ using Definition 4.2.13. The rules for incoming communication differ from the corresponding ones for outgoing communication in that well-typedness and well-formedness of the label is checked by the premises $\acute{\Xi} \vdash \lfloor a \rfloor : \vec{T} \to \_$, resp. $\acute{\Xi} \vdash \lfloor a \rfloor : \_ \to \vec{T}$ (for calls) resp., $\acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T$ (for *get*-labels), using Definition 4.2.12. For outgoing communication, the check is unnecessary as starting with a well-typed component, there is no need in re-checking now, as the operational steps preserve well-typedness (subject reduction).

When the component claims the value of a future, we distinguish two situations: the future value is accessed for the first time across the interface or not. In the first case (rules ClaimI$_1$ and ClaimI$_2$), the interface does not contain the value of the future yet, stipulated by the premise $\Delta \vdash n' = \bot$. Remember that $\Delta \vdash n'$ requires that n' is part of the environment. In that situation it is unclear from the perspective of the component, whether or not the value has already been computed. Hence, it is possible that executing claim is immediately successful (cf. rule CLAIM_1) or that the thread n trying to obtain the value has to suspend itself and try later (cf. rul CLAIM_2). Rule CLAIM_2 works exactly like the corresponding internal rule CLAIM_i^2 from Figure 4.7, except that here it is required that the queried future n' is part of the environment. The behavior of a thread wrt. claiming a future value has been illustrated in Figure 4.3 earlier. If the future value is already known at the interface (cf. rule CLAIM_3 and especially premise $\Delta \vdash n' = v$), executing claim is always successful and the value v is (re-)transmitted. get works analogously to claim, except that get insists on obtaining the value, i.e., the alternative of relinquishing the lock and trying again as in rule CLAIM_2, is not available for get. The last two rules deal with the situation that the environment fetches the value.

Finally, we characterize the *initial* configuration. Initially, the component contains at most one initial activity and no objects. More precisely, given that $\Xi_0 \vdash C_0$ is the initial judgment, then C_0 contains no objects. Concerning the threads: initially exactly one thread is executing, either at the component side or at the environment side. The distinction is made at the interface that initially either $\Theta_0 \vdash n$ or $\Delta_0 \vdash n$, where n is the only thread name in the system.

Remark 4.2.14 (Comparison with Java-like multithreading) *The formalization for the multithreaded case, for instance in [5], is quite similar. One complication encountered there is that one has to take* reentrance *into account. The rule for an incoming call* CALLI *in Figure 4.13 deals with a* non-reentrance *situation, which is the only situation relevant in the setting here. In addition to the rule* CALLI*, Java-like multithreading requires further* CALLI*-rules to cover the situations, when the call is reentrant.* □

4.3 Interface behavior

Next we characterize the possible ("legal") *interface behavior* as interaction traces between component and environment. Half of the work has been done already in the definition of the external steps in Figure 4.13: For incoming communication, for which the environment is responsible, the assumption contexts are consulted to check whether the communication originates from a realizable environment. Concerning the reaction of the component, no such checks are necessary. To characterize when a given trace is *legal,* the behavior of the

$$\frac{a = \nu(\Xi').\ n\langle call\ o.l(\vec{v})\rangle? \quad \acute{\Xi} = \Xi + a \quad (\Xi' \vdash n \vee \Delta \vdash n : [\_]^{+-}) \quad \acute{\Xi} \vdash o.l? : \vec{T} \to T \quad \acute{\Xi} \vdash \lfloor a \rfloor : \vec{T} \to \_}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash C \parallel C(\Xi') \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{grab}(o); M.l(o)(\vec{v})\ \mathsf{in}\ \mathsf{release}(o); x\rangle} \ \textsc{CallI}$$

$$\frac{a = \nu(\Xi').\ n\langle call\ o.l(\vec{v})\rangle! \quad \Xi' = fn(\lfloor a \rfloor) \cap \Xi_1 \quad \acute{\Xi}_1 = \Xi_1 \setminus \Xi' \quad \Delta \vdash o \quad \acute{\Xi} = \Xi + a}{\begin{array}{l}\Xi \vdash \nu(\Xi_1).(C \parallel C(\Xi') \parallel n\langle \bullet \rangle \parallel n'\langle \mathsf{let}\, x{:}T = \mathsf{bind}\ o.l(\vec{v}) : T \hookrightarrow n \ \ \mathsf{in}\, t\rangle) \xrightarrow{a} \\ \qquad \acute{\Xi} \vdash \nu(\acute{\Xi}_1).(C \parallel n'\langle \mathsf{let}\, x : T = n \ \ \mathsf{in}\, t\rangle)\end{array}} \ \textsc{CallO}$$

$$\frac{a = \nu(\Xi').\ n'\langle get(v)\rangle? \qquad \acute{\Xi} = \Xi + a \qquad \Delta \vdash n' = \bot \qquad \acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{claim@}(n', \_)\ \mathsf{in}\, t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\Xi_1).(C \parallel C(\Xi') \parallel n\langle \mathsf{let}\, x{:}T = v\ \mathsf{in}\, t\rangle)} \ \textsc{ClaimI}_1$$

$$\frac{\Delta \vdash n' = \bot}{\begin{array}{l}\Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{claim@}(n', o)\ \mathsf{in}\, t\rangle) \rightsquigarrow \\ \qquad \Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\, x : T = \mathsf{release}(o); \mathsf{get@}n'\ \mathsf{in}\ \mathsf{grab}(o); t\rangle)\end{array}} \ \textsc{ClaimI}_2$$

$$\frac{a = n'\langle get(v)\rangle? \qquad \Delta \vdash n' = v \qquad \Xi \vdash \lfloor a \rfloor : \_ \to T}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{claim@}(n', \_)\ \mathsf{in}\, t\rangle) \xrightarrow{a} \Xi \vdash \nu(\Xi_1).(C \parallel C(\Xi') \parallel n\langle \mathsf{let}\, x{:}T = v\ \mathsf{in}\, t\rangle)} \ \textsc{ClaimI}_3$$

$$\frac{a = \nu(\Xi').\ n'\langle get(v)\rangle? \qquad \acute{\Xi} = \Xi + a \qquad \Delta \vdash n' = \bot \qquad \acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{get@}n'\ \mathsf{in}\, t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\Xi_1).(C \parallel C(\Xi') \parallel n\langle \mathsf{let}\, x{:}T = v\ \mathsf{in}\, t\rangle)} \ \textsc{GetI}_1$$

$$\frac{a = n'\langle get(v)\rangle? \qquad \Delta \vdash n' = v \qquad \Xi \vdash \lfloor a \rfloor : \_ \to T}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{get@}n'\ \mathsf{in}\, t\rangle) \xrightarrow{a} \Xi \vdash \nu(\Xi_1).(C \parallel C(\Xi') \parallel n\langle \mathsf{let}\, x{:}T = v\ \mathsf{in}\, t\rangle)} \ \textsc{GetI}_2$$

$$\frac{a = \nu(\Xi').n\langle get(v)\rangle! \quad \Xi' = fn(\lfloor a \rfloor) \cap \Xi_1 \quad \acute{\Xi}_1 = \Xi_1 \setminus \Xi' \quad \acute{\Xi} = \Xi + a}{\Xi \vdash \nu(\Xi_1).(C \parallel C(\Xi') \parallel n\langle v\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Xi}_1).(C \parallel n\langle v\rangle)} \ \textsc{GetO}_1$$

$$\frac{a = n\langle get(v)\rangle! \quad \Theta \vdash n = v}{\Xi \vdash C \ \xrightarrow{a} \ \Xi \vdash C} \ \textsc{GetO}_2$$

Figure 4.13: External steps

component side, i.e., the outgoing communication, must adhere to the dual discipline we imposed on the environment for the open semantics. This means, we analogously abstract away from the program code, rendering the situation symmetric.

4.3.1 Legal traces system

The rules of Figure 4.14 specify legality of traces. We use the same conventions and notations as for the operational semantics (cf. Notation 4.2.11). The judgments in the derivation system are of the form

$$\Xi \vdash s : trace \ . \tag{4.10}$$

$$\Xi \vdash \epsilon : \mathit{trace} \qquad \text{L-EMPTY}$$

$$\frac{\begin{array}{c} a = \nu(\Xi').\ n\langle \mathit{call}\ o.l(\vec{v})\rangle? \quad \acute{\Xi} = \Xi + a \quad (\Xi' \vdash n \vee \Delta \vdash n : []^{+-}) \\ \acute{\Xi} \vdash o.l? : \vec{T} \to \_ \quad \acute{\Xi} \vdash \lfloor a \rfloor : \vec{T} \to \_ \quad \acute{\Xi} \vdash s : \mathit{trace} \end{array}}{\Xi \vdash a\ s : \mathit{trace}} \ \text{L-CALLI}$$

$$\frac{a = \nu(\Xi').n\langle \mathit{get}(v)\rangle? \quad \acute{\Xi} = \Xi + a \quad \Delta \vdash n = \bot \quad \acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T \quad \acute{\Xi} \vdash s : \mathit{trace}}{\Xi \vdash a\ s : \mathit{trace}} \ \text{L-GETI}_1$$

$$\frac{a = n\langle \mathit{get}(v)\rangle? \quad \Delta \vdash n = v \quad \Xi \vdash s : \mathit{trace}}{\Xi \vdash a\ s : \mathit{trace}} \ \text{L-GETI}_2$$

Figure 4.14: Legal traces (dual rules omitted)

We write $\Xi \vdash t : \mathit{trace}$, if there exists a derivation according to the rules of Figure 4.14 with an instance of L-EMPTY as axiom. The empty trace is always legal (cf. rule L-EMPTY), and distinguishing according to the first action a of the trace, the rules check whether a is possible. Furthermore, the contexts are updated appropriately, and the rules recur checking the tail of the trace. The rules are symmetric wrt. incoming and outgoing communication (the dual rules are omitted). Rule L-CALLI for incoming calls works completely analogously to the rule CALLI in the semantics: the second premise updates the context Ξ appropriately with the information contained in a, premise $\Xi' \vdash n$ of L-CALLI assures that the identity n of the future, carrying out the call, is fresh and the two premises $\acute{\Xi} \vdash o.l? : \vec{T} \to \_$ and $\acute{\Xi} \vdash \lfloor a \rfloor : \vec{T} \to \_$ together assure that the transmitted values are well-typed (cf. Definition 4.2.12); the latter two checks correspond to the analogous premises for the external semantics in rule CALLI, except that the return type of the method does not play a role here. The L-GETI-rules for claiming a value work similarly. In particular the type checking of the transmitted value is done by the combination of the premises $\Delta \vdash n : [T]$ and $\acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T$. As in the external semantics, we distinguish two cases, namely whether the value of the future has been incorporated in the interface already or not (rules L-GETI$_2$ and L-GETI$_1$). In both cases, the thread must be executing on the side of the environment for an incoming get. This is checked by the premise $\Delta \vdash n = \bot$ resp. by $\Delta \vdash n = v$. In case of

L-GetI$_2$, where the value of the future has been incorporated as v into the interface information, the actual parameter of the *get*-label must, of course, be v. If not (for L-GetI$_1$), the transmitted argument value is arbitrary, apart from the fact that it must be consistent with the static typing requirements.

It remains to show that the behavioral description, as given by Figure 4.14, actually does what it claims to do, to characterize the possible interface behavior of well-typed components. We show the soundness of this abstraction plus the necessary ancillary lemmas such as subject reduction. Subject reduction means, preservation of well-typedness under reduction. In the formulation of subject reduction, we make sure that the write-permissions of the environment are not available for type-checking the component. We use $\lfloor\Delta\rfloor$ instead of Δ as assumption, were $\lfloor\_\rfloor$ replaces each binding $n{:}[T]^{+-}$ in Δ by $n{:}[T]^{+}$.

Lemma 4.3.1 (Subject reduction) *If* $\lfloor\Delta\rfloor \vdash C : \Theta$ *and* $\Delta \vdash C : \Theta \stackrel{s}{\Longrightarrow} \acute{\Delta} \vdash \acute{C} : \acute{\Theta}$*, then* $\lfloor\acute{\Delta}\rfloor \vdash \acute{C} : \acute{\Theta}$*.*

Proof: By induction on the number of reduction steps. That internal steps preserve well-typedness, i.e., $\lfloor\Delta\rfloor \vdash C : \Theta \Longrightarrow \lfloor\acute{\Delta}\rfloor \vdash C : \Theta$, follows from the corresponding Lemma 4.2.9 for internal steps. That leaves the external reduction steps of Figure 4.13.

Case: CallI

We are given $\lfloor\Delta\rfloor \vdash C : \Theta$. The disjunctive premise of the rule distinguishes two sub-cases: 1) $\Xi' \vdash n$ (where Ξ' are the bindings carried along with the call-label, i.e., the thread name is transmitted freshly) or 2) $\Delta \vdash n : [\_]^{+-}$ (the thread is not transmitted freshly and the environment has write-permission before the step). Both are treated uniformly in the following argument. For the right-hand side of the transition, we need to show $\lfloor\acute{\Delta}\rfloor \vdash C' \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); x\rangle : \acute{\Theta}$, where C' corresponds to C extended by new $n'\langle\bullet\rangle$-promises. According to the definition of context update (Definition 4.2.13), $\acute{\Xi} = \acute{\Delta}, \acute{\Theta}$, where $\acute{\Theta} = \Theta, \Sigma'', n{:}[T]^{+}$ and where Σ'' contains bindings $n'{:}[T']^{+-}$ for those references transmitted with read-/write-permission as argument of the call (see the right-hand of Equation (4.8)). The assumption context $\acute{\Delta}$ for $\acute{C}$ after the step (by the left-hand of the same equation) is of the form $(\Delta, \Xi') \setminus \Sigma'' \setminus n{:}[T]^{+-}$. So for the new thread n at component side, we need to show that

$$\lfloor(\Delta, \Xi') \setminus \Sigma'' \setminus n{:}[T]^{+-}\rfloor \vdash C \parallel C(\Sigma'') \parallel n\langle t'\rangle : \Theta, \Sigma'', n{:}[T]^{+} \qquad (4.11)$$

with t' given as $\mathsf{let}\, x{:}T = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); x$. To derive Equation (4.11), using a number of instances of rule T-Par in the last derivation steps, gives

$$\dfrac{\dfrac{\tilde{\Delta}, \lfloor \Sigma'' \rfloor, n{:}[T]^+ \vdash C : \Theta \qquad \tilde{\Delta}, n{:}[T]^+, \lfloor \Theta \rfloor \vdash C(\Sigma'') : \Sigma'' \qquad \dfrac{\tilde{\Delta}, \lfloor \Theta \rfloor, \Sigma'', n{:}[T]^+ \vdash t' : T}{\tilde{\Delta}, \lfloor \Theta \rfloor, \Sigma'' \vdash n\langle t' \rangle : n{:}[T]^+}}{\vdots}}{\lfloor (\Delta, \Xi') \setminus \Sigma'' \setminus n{:}[T]^{+-} \rfloor \vdash C \parallel C(\Sigma'') \parallel n\langle t' \rangle : \Theta, \Sigma'', n{:}[T]^+} \tag{4.12}$$

where $\tilde{\Delta}$ abbreviates the assumption context $\lfloor (\Delta, \Xi') \setminus \Sigma'' \setminus n{:}[T]^{+-} \rfloor$ from Equation (4.11). Note, how the write-permissions from Σ'' in the commitment of the conclusion at the bottom are split among the three subgoals. All write-permissions are given to the assumptions of $n\langle t' \rangle$, whereas C can assume only read-access (cf. rule T-Par and the definition of $\oplus$ from Definition 4.2.1). The context Θ is split similarly. The left open goal can be rephrased as $\lfloor \Delta \rfloor, \lfloor \Xi' \rfloor, n{:}[T]^+ \vdash C : \Theta$ and can be discharged using the given $\lfloor \Delta \rfloor \vdash C : \Theta$ and weakening. The open goal in the middle follows directly from an appropriate number of instances of rule T-NThread$'$.

It remains to show the right-upper subgoal (with $\tilde{\Delta}$ expanded):

$$\lfloor (\Delta, \Xi') \setminus \Sigma'' \setminus n{:}[T]^{+-} \rfloor, \lfloor \Theta \rfloor, \Sigma'', n{:}[T]^+ \vdash \mathsf{let}\, x{:}T = t' : T \tag{4.13}$$

Note that, apart from the write-permissions, the complicated type context corresponds to: Δ, Ξ', Θ, or more formally:

$$\lfloor (\lfloor (\Delta, \Xi') \setminus \Sigma'' \setminus n{:}[T]^{+-} \rfloor, \lfloor \Theta \rfloor, \Sigma'', n{:}[T]^+) \rfloor = \lfloor \Delta, \Xi', \Theta \rfloor$$

Intuitively, it means, t' must be checked with all name bindings available from Δ and Θ plus the ones, which scope is exchanged in Ξ' as part of the label. No *write*-permissions, however, can be used to type-check t' *except* those being transmitted by the argument of the call and which are kept in Σ'' (the context Σ'' is the only part of t' typing context *not* being stripped off the write-permissions by $\lfloor \_ \rfloor$).

Note that the meta-mathematical notation $M.l(o)(\vec{v})$ in t' stands for the substitution $t_{body}[o/s][\vec{v}/\vec{x}]$, i.e., the method body with the self-parameter s substituted by the callee's identity and with the formal parameter replaced by the actual ones. Now, the well-typedness of the pre-configuration $\lfloor \Delta \rfloor \vdash C : \Theta$ together with the premise $\acute{\Xi} \vdash o.l : ?\vec{T} \to T$ of rule CallI (cf. Definition 4.2.12) implies that C is of the form $C' \parallel c[\![\ldots, l = \varsigma(s{:}c).\lambda(\vec{x} : \vec{T}).t_{body}, \ldots]\!]$, and further that $\lfloor \Delta \rfloor \vdash C : \Theta$ has $\vec{x}{:}\vec{T}; \lfloor \Delta \rfloor, \Theta \vdash t_{body} : T$ as subgoal. The remaining subgoal Equation (4.13) of the derivation Equation (4.12) follows by rules T-Let, T-Grab, preservation of typing under substitution, rule T-Release, and the axiom T-Var.

Case: CallO
We are given $\Delta \vdash \nu(\Xi_1).(C \parallel n\langle\bullet\rangle \parallel n'\langle \mathsf{let}\, x{:}T = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n \ \ \mathsf{in}\, t\rangle) : \Theta$ before the step and $\acute{\Delta} \vdash \nu(\acute{\Xi}_1).(C \parallel n'\langle \mathsf{let}\, x : T = n \ \ \mathsf{in}\, t\rangle) : \acute{\Theta}$ afterwards, with $C = C' \parallel n_1\langle\bullet\rangle \parallel \ldots \parallel n_k\langle\bullet\rangle$. By one of the premises of rule CallO we know $\Delta \vdash o$, i.e., object o is an environment object.[8] That the name o refers to an object is assured by the type system and the assumption that the pre-configuration is well-typed. By inverting the rules T-Nu, T-Par, T-NThread, T-Let, and T-Bind, we get:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\ldots \quad \Xi' = \lfloor\Delta\rfloor, \Xi_1, \Theta \quad \acute{\Xi}' = \Xi' \setminus (\vec{v}{:}\vec{T}, n{:}[T]^{+-})}{;\lfloor\Delta\rfloor, \Xi_1, \Theta \vdash \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n : T :: \acute{\Delta}'}\,\text{T-Bind} \qquad x{:}T; \acute{\Xi}' \vdash t : T' :: \ldots}{;\lfloor\Delta\rfloor, \Xi_1, \tilde{\Theta}, n'{:}[T']^{+} \vdash \mathsf{let}\, x{:}T = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n\, \mathsf{in}\, t : T'}}{\lfloor\Delta\rfloor, \Xi_1, \tilde{\Theta} \vdash n'\langle \mathsf{let}\, x{:}T = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n\, \mathsf{in}\, t\rangle : n'{:}[T']^{+}}}{\vdots}\,\text{T-Par, T-Nu}\ldots}{\lfloor\Delta\rfloor \vdash \nu(\Xi_1).(C \parallel n'\langle \mathsf{let}\, x : T = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n \ \ \mathsf{in}\, t\rangle \parallel n\langle\bullet\rangle) : \Theta}$$

Note that t in the left-upper leaf is type-checked in the context Ξ', which corresponds to $\Xi' = \lfloor\Delta\rfloor, \Xi_1, \tilde{\Theta}, n'{:}[T']^{+}$ with those write-permissions *removed* that are transmitted via the arguments of the method l (cf. rule T-Bind).

To derive well-typedness of the post-configuration, we distinguish two subcases, namely whether 1.) the promise n is known at the interface before the step or 2.) it is still hidden. In the first subcase, we have $\Theta \vdash n : T'$ with $T' = [T]^{+-}$ (as a consequence of the fact that the configuration is well-typed), or more precisely, $\Theta = \tilde{\Theta}', n{:}[T]^{+-}, n'{:}[T']^{+}$. The derivation of well-typedness of the post-configuration $\lfloor\acute{\Delta}\rfloor \vdash \nu(\acute{\Xi}_1).(C' \parallel n'\langle \mathsf{let}\, x : T = n \ \ \mathsf{in}\, t\rangle) : \acute{\Theta}$ works as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{;\lfloor\acute{\Delta}\rfloor, \acute{\Xi}_1, \acute{\Theta} \vdash n : T \qquad x{:}T; \lfloor\acute{\Delta}\rfloor, \acute{\Xi}_1, \acute{\Theta} \vdash t : T'}{;\lfloor\acute{\Delta}\rfloor, \acute{\Xi}_1, \acute{\Theta} \vdash \mathsf{let}\, x{:}T = n\, \mathsf{in}\, t : T'}\,\text{T-Let}}{\lfloor\acute{\Delta}\rfloor, \acute{\Xi}_1, \tilde{\acute{\Theta}} \vdash n'\langle \mathsf{let}\, x{:}T = n\, \mathsf{in}\, t\rangle : n'{:}[T']^{+}}\,\text{T-NThread}}{\vdots}\,\text{T-Par, T-Nu}\ldots}{\lfloor\acute{\Delta}\rfloor \vdash \nu(\acute{\Xi}_1).(C' \parallel n'\langle \mathsf{let}\, x : T = n \ \ \mathsf{in}\, t\rangle) : \acute{\Theta}}$$

[8]We do not allow cross-border instantiation in this paper, i.e., the component is not allowed to instantiate environment objects and vice versa.

where $\acute{\Theta} = \tilde{\acute{\Theta}}, n':[T']^+$. Using the premises of the reduction rule CallO, that relates the different binding contexts mentioned in the step (i.e., $\acute{\Xi}_1 = \Xi_1 \setminus \Xi'$, where Ξ' are the bindings mentioned in the call labels), $\Delta' = \Delta, \Xi', n:[T]^+$ (as stipulated by the premise $\acute{\Xi} = \Xi + a$ of rule CallO and given by Definition 4.2.13, especially Equation (4.8)). Note that Equation (4.8) is formulated for incoming communication, i.e., used dually here, and that in the considered subcase, we assume that n is known at the interface before the step, i.e., $\Theta \vdash n:[T]^{+-}$, as agreed upon earlier. It is straightforward to see that the combined context Δ, Ξ_1, Θ equals $\acute{\Theta}, \acute{\Xi}_1, \acute{\Delta}$, with the exception, that the former contains $n:[T]^{+-}$ (as part of Θ) and the latter only $n:[T]^+$ (as part of $\acute{\Delta}$. Cf. especially by Equation (4.8)). Furthermore, considering $\lfloor\Delta\rfloor$ and $\lfloor\acute{\Delta}\rfloor$ instead of Δ and $\acute{\Delta}$:

$$\lfloor\acute{\Delta}\rfloor, \acute{\Xi}_1, \acute{\Theta} = (\lfloor\Delta\rfloor, \Xi_1, \Theta) \setminus (n:[T]^{+-}, \vec{v}:\vec{T}) \tag{4.14}$$

where $\vec{v}:\vec{T}$ is given as mentioned in the left-upper leaf of the first derivation tree and as defined by the premise of rule T-Bind (these bindings correspond to the Σ'' used in Equation (4.8) and represent the write-permissions transmitted by the call-label from the component to the environment). This discharges the top-left subgoal of the derivation. The second subcase with $\Xi_1 \vdash n:[T]^{+-}$ works analogously.

Case: Claim$_1$

The core of the type preservation here is to assure that the claim-statement in the pre-configuration and the transmitted value v in the post-configuration are of the same appropriate type T. Well-typedness of the pre-configuration implies with claim@$(n', o,)$ of type T, that the reference n' is of type $[T]^+$. The third premise of rule ClaimI$_1$ states $\acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T$, which implies with Definition 4.2.12, especially rule LT-GetI of Figure 4.11, that also v is of type T, as required.

Case: Claim$_2$

By inverting the type rules T-Nu, T-Par, T-Let and T-Claim for the pre-configuration of the step, and by using the same typing rules (except T-Claim) plus T-Get, T-Release, and T-Grab.

The remaining rules work similarly. □

Lemma 4.3.2 (Soundness of abstractions) *If $\Xi_0 \vdash C$ and $\Xi_0 \vdash C \stackrel{t}{\Longrightarrow}$, then $\Xi_0 \vdash t : trace$.*

Proof: By induction on the number of steps in $\stackrel{t}{\Longrightarrow}$. The base case of zero steps (which implies $t = \epsilon$) is immediate, using rule L-Empty. The induction for internal steps of the form $\Xi \vdash C \Longrightarrow \Xi \vdash \acute{C}$ follows by subject reduction for internal steps from Lemma 4.2.9; in particular, internal steps do not change the context Ξ. The external steps of Figure 4.13 remain. First note the contexts Ξ are *updated* by each external step to $\acute{\Xi}$ the same way as the contexts are updated in the legal trace system.

The cases for incoming communication are checked straightforwardly, as the operational rules check incoming communication for legality, already, i.e., the premises of the operational rules have their counterparts in the rules for legal traces.

Case: CallI
Immediate, as the premises of rule L-CallI coincide with the ones of rule CallI.

Case: Claim$_1$ and Get$_1$
The two cases are covered by rule L-Get$_1$, which has the same premises as the operational rules.

Case: Claim$_2$
Trivial, as the step is an internal one.

Case: Claim$_3$ and Get$_2$
The two cases are covered by rule L-Get$_2$.

The cases for outgoing communication are slightly more complex, as the label in the operational rule is not type-checked or checked for well-formedness as for incoming communication and as is done in the rules for legality.

Case: CallO
We need to check whether the premises of rule L-CallO, the dual to rule L-CallI of Figure 4.14, are satisfied. By assumption, the pre-configuration

$$\Xi \vdash \nu(\Xi_1).(C \parallel n'\langle \mathsf{let}\, x{:}T = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n \ \ \mathsf{in}\, t\rangle) \qquad (4.15)$$

is well-typed. For thread name n this implies, it is bound either in Ξ or in Ξ_1, more precisely, either $\Theta \vdash n : [T]^{+-}$ (it is public interface information that the component has write-permission for n) or $\Xi_1 \vdash n : [T]^{+-}$ (the name n is not yet known in the environment before the communication). In the latter situation we obtain $\Xi' \vdash n : [\_]^{+-}$ by the premise $\Xi' = fn(\lfloor a \rfloor) \cap \Xi_1$ of rule CallO. Thus, the third premise $\Xi' \vdash n \vee \Theta \vdash n : [\_]^{+-}$ of rule L-CallO is satisfied. We furthermore need to check whether the label is type-correct (checked by premises nr. 4 and 5 of rule L-CallO). Its easy to check that the label is well-formed (cf. the first part of Definition 4.2.12). The first premise of the check of Equation (4.6), that the receiving object o is an environment object, is directly given by the premise $\Delta \vdash o$ of rule CALLO. That the object o supports a method labeled l (of type $\vec{T} \to T$) follows from the fact that the pre-configuration of the call-step is well-typed. This gives the premise $\acute{\Xi} \vdash o.l! : \vec{T} \to T$ of rule L-CALLO. The type check $\acute{\Xi} \vdash \lfloor a \rfloor : \vec{T} \to \_$ (checking that the transmitted values $\vec{v}$ are of the expected type $\vec{T}$) remains, which again follows from well-typedness of Equation (4.15) (especially inverting rule T-BIND).

The remaining cases work similarly. □

Remark 4.3.3 (Comparison with reentrant threading) *In a multithreaded setting with synchronous method calls (see for instance [5] [95]), the definition of legal traces is more complicated. Especially, to judge whether a trace s is possible requires referring to the past. I.e., instead of judgments of the form of Equation (4.10), the check for legality with synchronous calls uses judgments of the form:*

$$\Xi \vdash r \rhd s : trace \ ,$$

reading "after history *r (and in the context Ξ), the trace s is possible". This difference has once more to do with reentrance, resp. with the absence of this phenomenon here. In the threaded case, where, e.g., an outgoing call can be followed by a subsequent incoming call as a "call-back". To check therefore, whether a call or a return is possible as a next step involves checking the proper nesting of the call- and return-labels. This nesting requirement (also called the balance condition) degenerates here in the absence of call-backs to the given requirement that each call uses a fresh (future) identity and that each get-label (taking the role of the return label in the multithreaded setting) is preceded by exactly one matching preceding call. This can be judged by $\Delta \vdash n : [\_]$ or $\Theta \vdash n : [\_]$ (depending on whether we are dealing with incoming or outgoing $getfuts$-labels) and especially, no reference to the history of interface interactions is needed.* □

Remark 4.3.4 (Monitors) *The objects of the calculus act as monitors as they allow only one activity at a time inside the object. For the operational semantics of Section 4.2.3, the lock-taking is part of the* internal *steps. In other words, the handing-over of the call at the interface and the actual entry into the synchronized method body is* non-atomic*, and at the interface, objects are* input-enabled.

This formalization therefore resembles the one used for the interface description of Java-like reentrant monitors in [5]. To treat the interface interaction and actual lock-grabbing as non-atomic leads to a clean separation of *concerns of the component and of the environment. In [5], this non-atomicity, however, gives rise to quite complex conditions characterizing the legal interface behavior. In short, in the setting of [5], it is non-trivial to characterize exactly those situations, when the lock of the object is* necessarily *taken by one thread which makes certain interactions of other threads impossible. This characterization is non-trivial especially as the interface interaction is non-atomic.*

Note, however, that these complications are not present *in the current setting with active objects, even if the objects act as monitors like in [5]. The reason is simple: there is no* need *to capture situations when the lock is taken. In* Java, *the synchronization behavior of a method is part of the* interface *information. Concretely, the* synchronized-*modifier of* Java, *specifies that the body of a method is executed atomically in that object without interference of other*[9] *threads, assuming that all other methods of the callee are synchronized, as well. Here, in contrast, there is no interface information that guarantees that a method body is executed atomically. In particular, the method body can give up the lock temporarily via the* `suspend`-*statement, but this fact is not reflected in the interface information here. This absence of knowledge simplifies the interface description considerably.* □

4.4 Conclusion

We presented an open semantics describing the interface behavior of components in a concurrent object-oriented language with futures and promises. The calculus corresponds to the core of the *Creol* language, including classes, asynchronous method calls, the synchronization mechanism, and futures, and extended by promises. Concentrating on the black-box interface behavior, however, the interface semantics is, to a certain extent, independent of the concrete language and is characteristic for the mentioned features; for instance, extending *Java* with futures (see also the citations below) would lead to a quite similar formalization (of course, low level details may be different). Concentrating on the concurrency model, certain aspects of *Creol* have been omitted here, most notably inheritance and safe asynchronous class upgrades.

[9] Note that a thread can "interfere" in that setting with itself due to recursion and reentrance.

Future work

An obvious way to proceed is to consider more features of the *Creol*-language, in particular inheritance and subtyping. Incorporating inheritance is challenging, as it renders the system open wrt. a new form of interaction, namely the environment inheriting behavior from a set of component classes or vice versa. Also *Creol*'s mechanisms for dynamic class upgrades should be considered from a behavioral point of view (that we expect to be quite more challenging than dealing with inheritance). An observational, black-box description of the system behavior is necessary for the compositional account of the system behavior. Indeed, the legal interface description is only a first, but necessary, step in the direction of a compositional and ultimately fully-abstract semantics, for instance along the lines of [95]. Based on the interaction trace, it will be useful to develop a logic better suited for specifying the desired interface behavior of a component than enumerating allowed traces. Another direction is to use the results in the design of a black-box testing framework, as we started for *Java* in [41]. We expect that, with the theory at hand, it is straightforward to adapt the implementation to other frameworks featuring futures, for instance, to the future libraries of *Java* 5.

Chapter 5

Termination Detection for Active Objects[1]

In this chapter we investigate deadlock detection for a modeling language based on active objects. To detect deadlock in an Actor-like subset of Creol we focus on the communication between the active objects. For the analysis of the model we translate a Creol configuration to a process algebra featuring the Linda coordination primitives. The translation preserves the deadlock behaviour of the model and allows us to apply a formalism introduced by Busi et al. [27] to detect global deadlocks in the process algebra.

5.1 Introduction

Active objects form a well established model for distributed systems. We present a static technique for the detection of global deadlock in concurrent systems of active objects. Our technique is based on a translation into a process algebra which features the coordination primitives of the Linda language and the representation of the process algebra as a P/T net following the formalism of Busi et al. [27].

We apply this technique to an Actor-like subset of the Creol modeling language. Creol [65, 66] is a modeling language for distributed concurrent systems based on asynchronous communication. In Creol a system consists of active objects communicating via asynchronous method calls, futures, and promises [4, 42]. Creol objects encapsulate their data and can only be accessed by their interfaces. In contrast to the synchronous setting where control, i.e.

[1]The work presented in this chapter was published as [44].

threads, passes object boundaries, each method call spawns a new process within the called object. However, only one process is active within an object. Further, instead of returning the result of a method invocation to the caller the result is stored in a future. Active processes can release control by means of, for example, waiting on a future for the result of a method call. This gives rise to a so-called discipline of cooperative multi-tasking within an object.

Creol programs in general give rise to complex *deadlock* situations because of the synchronization between processes involving requests for the result of an invocation represented by a future. The main result of this work is decidability of detection of global deadlocks for an Actor-like subset [7, 94] of Creol which restricts the fine-grained synchronization between the processes within an object to the coarse grained run-to-completion pattern of Actor-like languages. In other words, the execution of a method cannot be preempted in this subset. A request for the result of a computation by an active process of an object therefore blocks the object itself.

Abstracting further from data we show in this work that the coordination language Linda can be used as a natural model to describe the network communications of our Actor-like language by externalizing the set of pending calls of the objects into the tuple space. Busi et al. also investigate the consequences of two different semantics for message generation. In their work the distinction between ordered and unordered semantics is crucial. In the ordered semantics a message is generated immediately. Due to this choice messages appear in the order in which they were sent in the tuple space. In the unordered semantics a send-box for the message is added to the tuple space which has to be turned into the message itself in an internal step later. The ordered semantics is more expressive than the unordered one. In fact the ordered semantics is Turing-powerful. Of particular interest is that this distinction does not apply to the semantics of Creol programs because Creol programs do not allow for testing for a message or conditional branching on the presence/absence of messages. One of the main challenges to automated termination/global deadlock detection of Creol programs is the unbounded generation of fresh names representing futures. The main result of this chapter is that restricting to the run-to-completion model of execution, i.e., disallowing preemption of active processes, allows for a finite representation of futures and an application of the techniques described in Busi et. al. [27].

Outline This chapter is organized as follows. We start with the introduction of an Actor-like subset of Creol in Section 5.2 followed by a presentation of the used Linda dialect in Section 5.3. In Section 5.4 we introduce the translation to Linda. We investigate the properties of our translation in Section 5.5. We briefly discuss in Section 5.6 the semantic consequences of extending our Actor-like language to more refined communication primitives and more fine-grained synchronization patterns.

5.2 Active Objects

A Creol model describes a system of active objects. The active objects communicate via asynchronous method calls. Each active object is a monitor and allows at most one *active* process to execute within the object. Scheduling among the processes of an object is cooperative, i.e. a process has to release the monitor lock explicitly – except for termination. Each object has an unbounded set of pending processes. In case the lock of an active object is free any process in the set of pending processes can grab the lock and start to execute. The initial "active" behavior of an object is given in terms of a designated run-method which is the active process after object creation.

The explanation above characterizes the general behaviour of Creol objects. We ristrict ourselves to an Actor-like subset C_A of Creol which allows a translation to the coordination language Linda. In C_A we forbid so called processor release points, i.e. in Creol a process can suspend execution by releasing the object lock and continue execution later after grabbing the lock again. In the Actor setting methods run to completion, i.e. suspension is not available. We give syntax and operational rules for our C_A which resemble the rules for standard Creol without the corresponding primitives for suspension and processor release.

We focus on the analysis of the communication structure of the model to detect deadlocks caused by the communication pattern used by the model. Consequently we abstract from data except for object identities and futures. Due to the abstraction from data, conditional branching becomes non-deterministic choice $(e_1 + e_2)$. Abstracting from data we also abstract from object creation and assume all objects to be given in advance.

To translate C_A to Linda we introduce as an intermediate step an abstract semantics for C_A. In the concrete semantics runtime labels for futures are unique. The translation of C_A to Linda would involve a translation of these runtime labels into messages in Linda, consequently keeping these runtime labels would result in an unbounded message alphabet. To obtain a finite alphabet of messages we replace the unique runtime labels by communication labels which are only unique with respect to their syntactic occurence in the method definitions, i.e. consecutive invocations of the same method may use the same labels for the communication. This change is reflected by the abstract semantics. Furthermore we change the semantics of the request of the result of a call. In Creol reading a future is a non-destructive operation, i.e. a future can be read an arbitrary number of times. With respect to termination or deadlock detection consecutive reading of a future does not provide new information. If the first read operation on a future is successful all consecutive read operations on that future are successful. This property of the read operation allows us to omit all consecutive read operations on a future and to replace the non-destructive read operation of the concrete semantics by a destructive read operation in the abstract semantics.

5.2.1 Syntax

Given a set of method names M with typical element m, the definition of an object consists of a labeled set of method definitions of the following form

$$\textbf{Object}\ o\{run = e; ret, m_1 = e_1; ret, \ldots, m_n = e_n; ret\}.$$

The designated *run*-method $run \in M$ defines the initial activity of the object which is executed after object creation. The expressions for the method bodies are given by the following grammar:

$$e \quad ::= \quad \tau \mid o.m \mid \rhd f = o.m \mid \rhd f? \mid e; e \mid e + e \qquad \text{expression}$$

Here, τ denotes an internal step. Both $o.m$ and $\rhd f = o.m$ denote an asynchronous method call to method m of object o. We distinguish anonymous calls $o.m$ and named calls $\rhd f = o.m$. Anonymous calls and named calls differ in the way they treat the result of the call. For anonymous calls no reference to the result is saved, i.e. the result cannot be requested by the caller. For named calls a reference to the future is stored in the local state of the object and the caller can request the result later by a corresponding get-expression, e.g. $\rhd f?$. We call the future f the name of the method call. We give a notion of balanced method definitions with respect to the named calls in a method body, i.e. in each branch of a choice the same set of futures has to be requested. The notion of balanced methods facilitates the definition of the finite representation of communication labels. This is restriction for technical convenience only. It is possible to deal with unbalanced method definitions. Applying static analysis and/or program transformations any method definition can be transformed into a balanced one.

The expression $\rhd f?$ is a blocking request of the result of the call with name $\rhd f$. The result is consumed upon request, which entails that $\rhd f?; \rhd f?$ leads to a deadlock. The return symbol *ret* represents the writing of the result and the termination of the method. The grammar guarantees that any method definition ends with a *ret*–statement and that there is only the *ret*–statement in the final position and no intermediate ones. Both ending each method with an explicit *ret*–statement and restricting the definition to exactly one *ret*–statement is done for technical convenience only. In abuse of notation we use e as a shorthand for $e'; ret$ throughout the remainder of this paper.

The language has no explicit construct for iteration but recursion is supported via anonymous method calls. For technical convenience we assume without loss of generality that the identifiers of all futures in the program are statically unique and hence identify the occurence of the corresponding call, i.e. we can use the futures as communication labels.

A Creol model is given in terms of the set of the objects O defining the model. By D_o we refer to the set of method definitions $m = e; ret$ given in o. For technical convenience we assume the names of the methods to be unique among all objects. By $D_o(m)$ we denote the definition given for method m in o, i.e. $e; ret$ for $m = e; ret$.

Example 5.2.1 (Running Example) *We give a running example in terms of a Creol model inspired by Dijsktra's classical deadly embrace example. The system consists of two objects o_1 and o_2 with similar behaviour. First each object calls a method on itself. Then the invocation calls a method on the other object and waits for the result. Depending on the scheduling of the methods the model can run into a deadlock.*

Object $o_1\{run = o_1.m_1; ret, m_1 = \rhd f = o_2.m_4; \rhd f?; ret, m_2 = ret\}$.
Object $o_2\{run = o_2.m_3; ret, m_3 = \rhd g = o_1.m_2; \rhd g?; ret, m_4 = ret\}$.

Both a non-deadlocking and a deadlocking run of the model are presented in section 5.2.2. The deadlock related to this example is an instance of the circular waiting problem.

Definition 5.2.2 (Well-formedness) *A method is* well-formed *if each request of a future, e.g. $\rhd f?$, is in the scope of a corresponding declaration, e.g. $\rhd f = o.m$, and if each future is only declared once. Well-typed Creol programs satisfy this requirement.*

As futures are local to method bodies and cannot be passed around, the request for a future that has not been declared before always leads to a deadlock. We only consider well-formed programs.

Example 5.2.3 (Running Example: Well-formedness) *It is easy to see that our running example is a well-formed model, i.e. each future that is requested was declared before.*

Example 5.2.4 (Well-formedness) *We give a slight variation of the definition of o_1 to illustrate the importance of the notion of well-formedness. The following program is not well-formed:*

> **Object** $o_1\{run = o_1.m_1; ret, m_1 = \rhd f?; ret, m_2 = ret\}$
> **Object** $o_2\{run = o_2.m_3; ret, m_3 = \rhd g = o_1.m_2; \rhd g?; ret, m_4 = ret\}$

If we request the future f without declaring it first, i.e. without doing the actual call. o_1 will wait forever, thus o_1 is deadlocked forever. In this case the whole system will deadlock. Either the calls started by the initial activity of o_2, i.e. m_3 and m_2, are scheduled before m_1 is scheduled and the activity initiated by o_2 terminates which leaves m_1 as the only and deadlocked process or m_1 blocks object o_1 before m_3 or m_2 are scheduled. In this case the call m_2 is deadlocked by m_1 blocking o_1. Here we get a circle of processes waiting: m_3 is waiting for the result of m_2 which is waiting for m_1 to free o_1 which is waiting for the undeclared future f, i.e. for a result that can never be calculated.

5.2.2 Operational Semantics

Once scheduled a process does not release control until termination, i.e. the method "runs-to-completion". The operational semantics of a corresponding system of active objects is described by a labeled transition relation between *configurations*. Given a set of object definitions, a *global configuration* Θ contains a set of *object configurations* and a set of futures. We assume an infinite set of run-time labels κ for the identification of named calls and their corresponding futures. For technical convenience, the distinguished label $\perp$ will be used to identify processes generated by anonymous calls. A configuration of an object named o is given by a tuple $(o, (\sigma, a), \Gamma)$. We assume object names to be unique in valid configurations Θ. The status of the currently active process is given by σ and a, where a is the expression representing the active process and σ is the process's local state assigning run-time labels to names of futures. Finally, Γ is the set of pending calls resp. the set of pending processes. An active process is a pair of a run-time label and an expression to be executed. A pending process is a pair of a run-time label and a method name. For a process generated by a named call its unique runtime label κ also identifies the return value. By ϵ we denote the absence of an active process, i.e. an empty local state and an idle object.

Note that in the run-to-completion semantics there are no partially evaluated processes in the set of pending processes but only "fresh" method invocations. The active process represents the method currently executed by the object and it has exclusive access to the object.

Initial configuration The initial configuration θ_o of an object o is given by the object itself containing the definitions of the methods, the active process, and an empty set of pending processes:

$$\theta_o = \{(o, (\sigma_\perp, \perp : D_o(run)), \emptyset)\} \ .$$

Here, $D_o(run)$ denotes the body of the *run*-method given in o. The active process is labelled with the runtime label $\perp$ as an anonymous invocation, i.e. "no" future will be produced. The initial local state $\sigma_\perp$ does not contain any assignments for the futures.

The initial configuration Θ_I of the model is the set of the initial configuration of the objects:

$$\Theta_I = \bigcup_{o \in O} \theta_o \ .$$

Method scheduling An idle object can non-deterministically schedule any pending process:

$$\Theta \cup \{(o, \epsilon, \Gamma \cup \{\kappa : m\})\} \to \Theta \cup \{(o, (\sigma_\perp, \kappa : D_o(m)), \Gamma)\}$$

Here, ϵ denotes the idle object, i.e. no active process in execution and no local state. Upon scheduling the method body $D_o(m)$ of method m is inlined and the local state $\sigma_\perp$ is initiated, i.e. an empty assignment is provided.

Method termination Upon method termination the object is set to idle and a future representing the result of the corresponding call is created:

$$\Theta \cup \{(o, (\sigma, \kappa : ret), \Gamma)\} \to \Theta \cup \{(o, \epsilon, \Gamma)\} \cup \{\kappa\}$$

Upon method termination the local state is discarded. The absence of a calculated result value justifies the representation of the result of a method call by transforming the runtime label κ of the invocation into a future denoting the termination of the corresponding invocation.

Choice The rules for the non-deterministic choice in C_A are as expected:

$$\Theta \cup \{(o, (\sigma, \kappa : e_1 + e_2; e), \Gamma)\} \to \Theta \cup \{(o, (\sigma, \kappa : e_1; e), \Gamma)\}$$

$$\Theta \cup \{(o, (\sigma, \kappa : e_1 + e_2; e), \Gamma)\} \to \Theta \cup \{(o, (\sigma, \kappa : e_2; e), \Gamma)\}$$

Internal Step Internal steps have no side effect on the configuration.

$$\Theta \cup \{(o, (\sigma, \kappa : \tau; e), \Gamma)\} \to \Theta \cup \{(o, (\sigma, \kappa : e), \Gamma)\}$$

Method call An anonymous method call adds a corresponding invocation to the set of pending processes of the callee and allows the caller to continue execution.

$$\begin{aligned} & \Theta \cup \{(o, (\sigma, \kappa : o'.m'; e), \Gamma)\} \cup \{(o', a', \Gamma')\} \\ \to \quad & \Theta \cup \{(o, (\sigma, \kappa : e), \Gamma)\} \cup \{(o', a', \Gamma' \cup \{\perp : m'\})\} \end{aligned}$$

Here $\perp$ suffices as a label for the process caused by an anonymous call to method m' of object o' by object o since no return value is requested.

Future A method call adds the call to the set of pending processes of the callee and allows the caller to continue execution.

$$\begin{aligned} & \Theta \cup \{(o, (\sigma, \kappa : f = o'.m'; e), \Gamma)\} \cup \{(o', a', \Gamma')\} \\ \to \quad & \Theta \cup \{(o, (\sigma[f := \kappa'], \kappa : e), \Gamma)\} \cup \{(o', a', \Gamma' \cup \{\kappa' : m'\})\} \end{aligned}$$

Here $\sigma[f := \kappa']$ denotes the result of assigning the label κ' to the future name $\rhd f$ in σ.

Requesting result A result to a method call is consumed upon request.

$$\Theta \cup \{(o, (\sigma[f := \kappa'], \kappa : f?; e), \Gamma)\} \cup \{\kappa'\} \to \Theta \cup \{(o, (\sigma, \kappa : e), \Gamma)\}$$

Consumption of the result is modeled by removing the future κ' from the configuration. Please note that requesting a result is blocking. In case the result is not available the process and thereby the object containing the process are stuck.

Example 5.2.5 (Running Example: Concrete Semantics) *We revisit our running example of a Creol program to illustrate the semantics of our Creol modelling language. The initial configuration consists of the two objects* o_1 *and*

o_2. For each object the run method is the active process. First both run methods are executed which leads to a pending call of m_1 at o_1 and a pending call of m_3 at o_2. The pending call of m_1 at o_1 is scheduled which leads to a named call f of m_4 at o_2 with runtime label κ. The call of method m_4 at o_2 is scheduled which produces the result κ. The result is consumed by the request of process m_1 at o_1. Now the pending call of m_3 at o_2 is scheduled. The execution of the method leads to an invocation of m_2 at o_1. The invocation of m_2 at o_1 is scheduled and produces a result κ'. The result is consumed by the invocation of method m_3 at o_2 and the execution terminates. During the execution a couple of anonymous results $\perp$ are produced. By definition these can not be claimed by any process.

$$\begin{array}{rl}
\Theta_I & = \{(o_1, (\sigma_\perp, \perp : o_1.m_1; \mathit{ret}), \emptyset), (o_2, (\sigma_\perp, \perp : o_2.m_3; \mathit{ret}), \emptyset)\} \\
& \to \{(o_1, (\sigma_\perp, \perp : \mathit{ret}), \{\perp : m_1\}), (o_2, (\sigma_\perp, \perp : o_2.m_3; \mathit{ret}), \emptyset)\} \\
& \to \{(o_1, (\sigma_\perp, \perp : \mathit{ret}), \{\perp : m_1\}), (o_2, (\sigma_\perp, \perp : \mathit{ret}), \{\perp : m_3\})\} \\
& \to \{(o_1, \epsilon, \{\perp : m_1\}), (o_2, (\sigma_\perp, \perp : \mathit{ret}), \{\perp : m_3\}), \perp\} \\
& \to \{(o_1, \epsilon, \{\perp : m_1\}), (o_2, \epsilon, \{\perp : m_3\}), \perp, \perp\} \\
& \to \{(o_1, (\sigma_\perp, \perp : \rhd f = o_2.m_4; \rhd f?; \mathit{ret}), \emptyset), (o_2, \epsilon, \{\perp : m_3\}), \perp, \perp\} \\
& \to \{(o_1, (\sigma[f := \kappa], \perp : \rhd f?; \mathit{ret}), \emptyset), (o_2, \epsilon, \{\perp : m_3, \kappa : m_4\}), \perp, \perp\} \\
& \to \{(o_1, (\sigma[f := \kappa], \perp : \rhd f?; \mathit{ret}), \emptyset), (o_2, (\sigma_\perp, \kappa : \mathit{ret}), \{\perp : m_3\}), \perp, \perp\} \\
& \to \{(o_1, (\sigma[f := \kappa], \perp : \rhd f?; \mathit{ret}), \emptyset), (o_2, \epsilon, \{\perp : m_3\}), \kappa, \perp, \perp\} \\
& \to \{(o_1, (\sigma, \perp : \mathit{ret}), \emptyset), (o_2, \epsilon, \{\perp : m_3\}), \perp, \perp, \perp\} \\
& \to \{(o_1, \epsilon, \emptyset), (o_2, \epsilon, \{\perp : m_3\}), \perp, \perp, \perp\} \\
& \to \{(o_1, \epsilon, \emptyset), (o_2, (\sigma_\perp, \perp : \rhd g = o_1.m_2; \rhd g?; \mathit{ret}), \emptyset), \perp, \perp, \perp\} \\
& \to \{(o_1, \epsilon, \{\kappa' : m_2\}), (o_2, (\sigma'[g := \kappa'], \perp : \rhd g?; \mathit{ret}), \emptyset), \perp, \perp, \perp\} \\
& \to \{(o_1, (\sigma_\perp, \kappa' : \mathit{ret}), \emptyset), (o_2, (\sigma'[g := \kappa'], \perp : \rhd g?; \mathit{ret}), \emptyset), \perp, \perp, \perp\} \\
& \to \{(o_1, \epsilon, \emptyset), (o_2, (\sigma'[g := \kappa'], \perp : \rhd g?; \mathit{ret}), \emptyset), \kappa', \perp, \perp, \perp\} \\
& \to \{(o_1, \epsilon, \emptyset), (o_2, (\sigma', \perp : \mathit{ret}), \emptyset), \perp, \perp, \perp\} \\
& \to \{(o_1, \epsilon, \emptyset), (o_2, \epsilon, \emptyset), \perp, \perp, \perp, \perp\}
\end{array}$$

In the remainder of this chapter we ignore, i.e. omit, anonymous results $\perp$ whenever appropriate.

Enabledness Under the assumption that all objects which are referenced, i.e. called, are included in the configuration any step except from requesting a result is always enabled. Requesting a result is only enabled if the return value of the call is available in the configuration.

Terminal Configuration We call a configuration terminal if there are no enabled steps for the configuration, i.e. there are no enabled steps for any object in the configuration.

Example 5.2.6 (Running Example: Terminal Configuration) *The execution of our running example, we presented above, ends in a terminal configuration.*

Definition 5.2.7 (Deadlock) *A terminal configuration is called a deadlock iff it contains an object that is not idle, i.e. which contains a (blocked) active process.*

The notion of deadlock is global in that it requires the whole configuration to be stuck. Local deadlocks, e.g. circular waiting among only a subset of the objects, is not covered by the definition. The detection of local deadlocks is not in the scope of this chapter.

Example 5.2.8 (Running Example: Deadlock) *We vary the execution of our running example of a Creol program. Instead of scheduling the call of m_4 at o_2 we schedule the call of m_3 at o_2 first. This leads to an invocation of m_2 at o_1. Now the active processes of both objects wait for the result of a call to the other object which is blocked by the other active process, i.e. no further progress can be made.*

$$
\begin{aligned}
\theta_o &= \{(o_1, (\sigma_\perp, \perp : o_1.m_1; \mathit{ret}), \emptyset), (o_2, (\sigma_\perp, \perp : o_2.m_3; \mathit{ret}), \emptyset)\} \\
&\rightarrow \{(o_1, (\sigma_\perp, \perp : \mathit{ret}), \{\perp : m_1\}), (o_2, (\sigma_\perp, \perp : o_2.m_3; \mathit{ret}), \emptyset)\} \\
&\rightarrow \{(o_1, (\sigma_\perp, \perp : \mathit{ret}), \{\perp : m_1\}), (o_2, (\sigma_\perp, \perp : \mathit{ret}), \{\perp : m_3\})\} \\
&\rightarrow \{(o_1, \epsilon, \{\perp : m_1\}), (o_2, (\sigma_\perp, \perp : \mathit{ret}), \{\perp : m_3\})\} \\
&\rightarrow \{(o_1, \epsilon, \{\perp : m_1\}), (o_2, \epsilon, \{\perp : m_3\})\} \\
&\rightarrow \{(o_1, (\sigma_\perp, \perp : \rhd f = o_2.m_4; \rhd f?; \mathit{ret}), \emptyset), (o_2, \epsilon, \{\perp : m_3\})\} \\
&\rightarrow \{(o_1, (\sigma[f := \kappa], \perp : \rhd f?; \mathit{ret}), \emptyset), (o_2, \epsilon, \{\perp : m_3, \kappa : m_4\})\} \\
&\rightarrow \{(o_1, (\sigma[f := \kappa], \perp : \rhd f?; \mathit{ret}), \emptyset), \\
&\qquad (o_2, (\sigma_\perp, \perp : \rhd g = o_1.m_2; \rhd g?; \mathit{ret}), \{\kappa : m_4\})\} \\
&\rightarrow \{(o_1, (\sigma[f := \kappa], \perp : \rhd f?; \mathit{ret}), \{\kappa' : m_2\}), \\
&\qquad (o_2, (\sigma'[g := \kappa'], \perp : \rhd g?; \mathit{ret}), \{\kappa : m_4\})\}
\end{aligned}
$$

The execution leads to a terminal configuration which contains blocked active processes thus it leads to deadlock.

We are going to detect deadlock via detection of termination. To distinguish normal termination and deadlock we introduce divergence by means of a decoration of the objects with a diverging method. We change the definition of an object to introduce divergence.

Object $o\{run = e; o.div_o; ret, m_1 = e_1; ret, \ldots, m_n = e_n; ret, div_o = o.div_o; ret\}$

It is easy to see that in case of the first execution in Example 5.2.5 the objects can start to execute the diverging methods, i.e. the last configuration is no longer terminal and no terminal configuration is reachable from the last configuration. In case of the second execution in Example 5.2.8 this is not the case. The diverging methods are only additional pending calls in the set of pending calls. The last configuration is still terminal and a deadlock. Using only standard primitives, i.e. method definition and anonymous call, we hide the diverging methods and simply assume it to be present instead of treating it explicitly. We only make use of the diverging methods in case of normal termination of a model.

In order to prove that our modelling language coincides with our Linda representation with respect to termination we give as an intermediate step an abstract semantics which introduces a finite representation for the potentially unbounded number of run-time labels generated by the concrete semantics. We show that the "concrete" and the "abstract" semantics coincide in case of well–formed, balanced programs. We show how to translate a Creol model in the abstract setting into a Linda configuration. Finally we give a bisimulation between our Creol model in the abstract setting and its counterpart in Linda.

5.2.3 Finite Representation of Creol

Next we give a finite representation of the unbounded generation of run-time labels. To obtain such a finite representation we restrict ourselves to balanced programs. Note, however, that this will *not* render the state-space finite as the set of messages is unbounded.

Definition 5.2.9 (Balanced) *A program is balanced if all objects are balanced. An object is balanced if all its methods are balanced. A method is* balanced *if for each choice in all branches of the choice the same multiset of futures is requested with respect to the set of futures that were defined before the choice.*

The unbounded production of run-time labels for futures is a major problem in the automated analysis of Creol program. To address this problem the combination of statically unique labels, balanced programs, and run–to–completion semantics is crucial. The statically unique labels allow to distinguish the individual calls by their futures. Due to the balancing a future is either consumed in any terminating execution of a method or by none. The run–to–completion semantics prevents two invocations of the same method to be interleaved. Together statically unique labels, balancing and run–to–completion semantics ensure that the results of two invocations are not confused. This allows for a precise analysis with respect to termination without unique runtime labels. For technical convenience we extend the notion of balancing. In addition to the branches behaving similarly we require each future to be consumed. This way we get an exact matching between named calls in the concrete semantics and named calls in the abstract semantics.

Example 5.2.10 (Balancing) *We give method definitions to illustrate the notion on balancing.*

$$m_1 = \rhd f = o.m'; (\rhd f?) + (\rhd g = o'.m''; \rhd f?); ret \tag{5.1}$$

$$m_2 = \rhd f = o.m'; (\rhd g = o'.m'') + (\rhd h = o''.m'''; \rhd h?); \rhd f?; ret \tag{5.2}$$

$$m_3 = \rhd f = o.m'; (\rhd f?) + (\rhd g = o'.m''; \rhd g?); ret \tag{5.3}$$

All method definitions are well-formed. Method definition m_1 is balanced. The future f is the only future that is defined before the choice and it is consumed in both branches of the choice. Method definition m_2 is balanced. The future f is the only future that is defined before the choice and it is not consumed in any of the branches of the choice.

Method definition m_3 is unbalanced. The future f is the only future that is defined before the choice and it is consumed in the left branch of the choice but not in the right branch. In case of an execution of m_3 a future f can be produced which is not consumed, i.e. if the right branch is chosen. This future remains in the configuration and in case of a later invocation of m_3 the left branch can be chosen and the request for the result of f can be served by the result of the first invocation. To exclude this kind of mismatch we restrict ourselves to balanced programs.

Intuitively, being balanced implies that there are no dangling run-time labels, i.e., labels not referenced by any future.

Lemma 5.2.11 (No dangling run-time labels) *Assume a balanced program. Then for every configuration reachable from the initial one we have* *the following mapping: For every run-time label κ not equal to $\perp$ appearing in a configuration Θ there exist a unique local state σ in Θ and a unique future name $\rhd f$ such that $\sigma(f) = \kappa$.*

Proof: The proof proceeds by induction on the length of the computation.

Base case: The base case is trivial because the initial configuration does not contain run-time labels different from $\perp$.

Induction step: We treat the following main cases of method call, scheduling, and return:

For a method call a fresh run-time label is created which is assigned only once to the corresponding future name. The well-formedness of the program definition guarantees that the future name has not been instantiated in the local state of the active process. The uniqueness of the future names guarantees that the future name has not been instantiated in the local state of a different object.

For method scheduling the local state $\sigma_\perp$ is assigned to the active process. For a freshly scheduled process there are no pending invocations or futures.

For method return due to balancing there are neither pending invocation nor pending futures labeled with a run-time label $\kappa \in \sigma$ with σ being the local state of the method returning. □

Next we observe that in absence of rescheduling only the active process has a local state and hence, future names are unambiguous. Together with the above lemma this allows us to replace the run-time labels of the concrete semantics by their corresponding future names.

The observation above allows us to abstract from the unique runtime labels and to give an abstract semantics for Creol. In abuse of notation we use ϵ to denote the absence of an active process in the abstract semantics.

Method scheduling Any pending process can be scheduled if the object is idle.

$$\Theta \cup \{(o, \epsilon, \Gamma \cup \{\rhd f : e\})\} \to \Theta \cup \{(o, \rhd f : e, \Gamma)\}$$

Method termination Upon method termination an abstract call label l is added to the configuration. For a named call the future f is added to the configuration for an anonymous call the designated label $\perp$ is added to the configuration which can not be consumed.

$$\Theta \cup \{(o, l : \mathit{ret}, \Gamma)\} \to \Theta \cup \{(o, \epsilon, \Gamma)\} \cup \{l\}$$

Method call For an anonymous call only the abstract label is added to the set of pending processes.

$$\Theta\cup\{(o,\rhd f{:}o'.m';e,\Gamma)\}\cup\{(o',a',\Gamma')\}\to\Theta\cup\{(o,\rhd f{:}e,\Gamma)\}\cup\{(o',a',\Gamma'\cup\{\bot{:}D_{o'}(m')\})\}$$

Future For a future the abstract label is added to the set of pending processes.

$$\begin{array}{rl} & \Theta \cup \{(o, \rhd f' : f = o'.m'; e, \Gamma)\} \cup \{(o', a', \Gamma')\} \\ \to & \Theta \cup \{(o, \rhd f' : e, \Gamma)\} \cup \{(o', a', \Gamma' \cup \{\rhd f : D_{o'}(m')\})\} \end{array}$$

Requesting result Requesting a result to a method call consumes an abstract call label.

$$\begin{array}{rc} & \Theta \cup \{(o, \rhd f' : f?; e, \Gamma)\} \cup \{\rhd f\} \\ \to & \Theta \cup \{(o, \rhd f' : e, \Gamma)\} \end{array}$$

Please note that by convention $\rhd f$ is unique for the static structure of the model.

Example 5.2.12 (Abstract Runtime Labels) *We give an example to illustrate the abstract semantics of Creol. Furthermore the example provides some insight into the notion of balancing.*

Object $o_1\{run = o_1.m_1; ret, m_1 = \rhd f = o_2.m_2; \rhd g = o_2.m_2; \rhd g?; o_1.m_1; ret\}$.
Object $o_2\{run = ret, m_2 = ret\}$.

The execution is a recursive execution of method m_1 which does two method calls to m_2 at o_2. The result of the call labeled with g is consumed. The result *of the call labeled with f is not consumed.*

$$\begin{array}{rl}
\theta_o = & \{(o_1, \bot : o_1.m_1; ret, \emptyset), (o_2, \bot : ret, \emptyset)\} \\
\rightarrow & \{(o_1, \bot : o_1.m_1; ret, \emptyset), (o_2, \epsilon, \emptyset)\} \\
\rightarrow & \{(o_1, \bot : ret, \{\bot : m_1\}), (o_2, \epsilon, \emptyset)\} \\
\rightarrow & \{(o_1, \epsilon, \{\bot : m_1\}), (o_2, \epsilon, \emptyset)\} \\
\rightarrow & \{(o_1, \bot : \rhd f = o_2.m_2; \rhd g = o_2.m_2; \rhd g?; o_1.m_1; ret, \emptyset), (o_2, \epsilon, \emptyset)\} \\
\rightarrow & \{(o_1, \bot : \rhd g = o_2.m_2; \rhd g?; o_1.m_1; ret, \emptyset), (o_2, \epsilon, \{f : m_2\})\} \\
\rightarrow & \{(o_1, \bot : \rhd g?; o_1.m_1; ret, \emptyset), (o_2, \epsilon, \{f : m_2, g : m_2\})\} \\
\rightarrow & \{(o_1, \bot : \rhd g?; o_1.m_1; ret, \emptyset), (o_2, f : ret, \{g : m_2\})\} \\
\rightarrow & \{(o_1, \bot : \rhd g?; o_1.m_1; ret, \emptyset), (o_2, \epsilon, \{g : m_2\}), f\} \\
\rightarrow & \{(o_1, \bot : \rhd g?; o_1.m_1; ret, \emptyset), (o_2, g : ret, \emptyset), f\} \\
\rightarrow & \{(o_1, \bot : \rhd g?; o_1.m_1; ret, \emptyset), (o_2, \epsilon, \emptyset), f, g\} \\
\rightarrow & \{(o_1, \bot : o_1.m_1; ret, \emptyset), (o_2, \epsilon, \emptyset), f\} \\
\rightarrow & \{(o_1, \bot : ret, \{\bot : m_1\}), (o_2, \epsilon, \emptyset), f\} \\
\rightarrow & \{(o_1, \epsilon, \{\bot : m_1\}), (o_2, \epsilon, \emptyset), f\} \\
\rightarrow^* & \ldots \\
\rightarrow & \{(o_1, \epsilon, \{\bot : m_1\}), (o_2, \epsilon, \emptyset), f, f\}
\end{array}$$

According to the core definition of balanced models a result is either consumed always, e.g. g, or never, e.g. f. This guarantees that results can never be mistaken but due to the constant generation of new futures f the mapping of concrete configurations to abstract ones gets complicated. This is why we enforce balanced models to consume the futures they declare for technical convenience.

Example 5.2.13 (Running Example: Abstract Semantics) *We revisit the deadlocking execution of our running example from Example 5.2.8 to present the deadlock in the abstract semantics.*

$$\begin{array}{rl}
\theta_o = & \{(o_1, \bot : o_1.m_1; ret, \emptyset), (o_2, \bot : o_2.m_3; ret, \emptyset)\} \\
\rightarrow & \{(o_1, \bot : ret, \{\bot : m_1\}), (o_2, \bot : o_2.m_3; ret, \emptyset)\} \\
\rightarrow & \{(o_1, \bot : ret, \{\bot : m_1\}), (o_2, \bot : ret, \{\bot : m_3\})\} \\
\rightarrow & \{(o_1, \epsilon, \{\bot : m_1\}), (o_2, \sigma_\bot, \bot : ret)\{\bot : m_3\}\} \\
\rightarrow & \{(o_1, \epsilon, \{\bot : m_1\}), (o_2, \epsilon, \{\bot : m_3\})\} \\
\rightarrow & \{(o_1, \bot : \rhd f = o_2.m_4; \rhd f?; ret, \emptyset), (o_2, \epsilon, \{\bot : m_3\})\} \\
\rightarrow & \{(o_1, \bot : \rhd f?; ret, \emptyset), (o_2, \epsilon, \{\bot : m_3, f : m_4\})\} \\
\rightarrow & \{(o_1, \bot : \rhd f?; ret, \emptyset), (o_2, \bot : \rhd g = o_1.m_2; \rhd g?; ret, \{f : m_4\})\} \\
\rightarrow & \{(o_1, \bot : \rhd f?; ret, \{g : m_2\}), (o_2, \bot : \rhd g?; ret, \{f : m_4\})\}
\end{array}$$

The execution leads to a terminal configuration in the abstract semantics which contains blocked active processes thus it leads to deadlock.

For the remainder of this chapter we assume programs to be balanced and well-formed.

Theorem 5.2.14 (Termination equivalence (Creol)) *An execution of a balanced and well–formed Creol model terminates in the concrete semantics iff an execution of the model in the abstract semantics terminates.*

Proof: We prove the relation by a bisimulation. The local variable assignment defines the mapping between a concrete configuration Θ and the corresponding abstract configuration $\alpha(\Theta)$. We define the global mapping σ to be the disjoint union of the local mappings σ_o: $\sigma = \bigcup_{o \in O} \sigma_o$

It is sufficient to show $\Theta \rightarrow \Theta'$ iff $\alpha(\Theta) \rightarrow \alpha(\Theta')$. □

5.3 Linda

Linda is a coordination language. It provides a tuple space of messages and primitives to add messages to the tuple space, remove messages from the tuple space and test the tuple space for the existence of messages. Processes exchange messages via the tuple space and only via the tuple space.

Busi et al. [27] introduce a process algebra containing the coordination primitives of Linda. This process algebra is interpreted in two different semantics treating the creation of messages differently. The so-called ordered semantics features immediate message creation, i.e. upon execution of the creation primitive the message is available in the tuple space. This is not the case for the so-called unordered semantics. In the unordered semantics a sendbox is added to the tuple space which is responsible for creating the actual message at a later point in time. The ordered semantics is Turing powerful whereas the unordered one is not.

The differences in the expressiveness of the two semantics originate from a coupling between the message provider and the message consumer that is established via the conditional input primitives - in the ordered semantics one process can derive knowledge about the state of another process by the existence or absence of messages. In this chapter we do not use these primitives to translate our Creol model to Linda. For our subset of the language the ordered and unordered semantics coincide. The semantics of the subset we use is not Turing powerful. We do not give a detailed proof of this property as it would require to repeat significant parts of Busi et al. [27]. Furthermore it is a straightforward variation of the proof of Busi et al. [27] that the unordered semantics is not Turing powerful. We give intuition on how to vary the proof.

Busi et al. [27] give a representation of the process algebra in terms of a restricted form of contextual P/T nets which can be simulated by finite P/T nets. For finite P/T nets termination is decidable. To detect deadlock Busi et al. model deadlock as termination and check the P/T net representation for termination. We follow their approach to detect deadlock in Creol models. We translate our Creol model to Linda and employ the formalism of Busi et al. [27] to detect deadlock.

5.3.1 Syntax

Let the messages be a denumerable set of message names, ranged over $a, b, \ldots$. The syntax of the language $\mathbf{L}_A$ is defined by the following grammar:

$$\begin{array}{rcl} P & ::= & \langle a \rangle \mid C \mid P|P \\ C & ::= & 0 \mid \eta.C \mid C|C \ . \end{array}$$

where:

$$\eta \ ::= \ \mathrm{in}(a) \mid \mathrm{out}(a) \mid !\mathrm{in}(a)$$

Compared to $\mathbf{L}$ from Busi et al. [27], we omit conditional choices $\mathrm{inp}(a)?C\_C$ and $\mathrm{rdp}(a)?C\_C$, and the test for presence of messages $\mathrm{rd}(a)$. The difference between the ordered and unordered semantics results from the conditional choices in combination with the semantics of the output action. Without conditional choice the difference between instantaneous and buffered output is no longer observable.

5.3.2 Semantics

We give semantics for our process algebra following Busi et al. [27]. Though we omitted the primitives for testing for messages and conditional branching we do present these rules. In Section 5.6 we discuss the semantic consequences of adding conditional branching and conditional scheduling to our subset of Creol for this discussion it is helpful to have the rules for the corresponding Linda primitives at hand. Figure 5.1 shows the reduction rules for Linda. Rule (1) describes the input of a message from the point of view of the message. Rule (2) describes the input of a message from the point of view of the receiver. Rules (1), (2) and (11) describe the input of a message. Rule (3) describes the testing for a message from the point of view of the tester. Rules (1), (3) and (12) describe the testing for a message.

Rule (4) describes the replication operation. The trigger message for the replication is consumed in the replication step. Rules (5) and (7) describe conditional branching. The guard is a message. The guard message is consumed in case of its existence. Rules (6) and (8) describe conditional branching, too. In this case the condition is a test for existence of a message. The guard message is not consumed in this case.

Rules (9) and (10) describe the parallel execution. To have a sound treatment of conditional branching we have to ensure that we only decide on non–existence of a message ($\neg a$) if the message does not exist in any of the parallel processes.

We present the full set of rules as presented in [27]. Though we use only rules (3) and (5)–(8). We do not need testing or conditional branching to translate Creol models to Linda.

$$
\begin{array}{clcl}
(1) & \langle a \rangle \xrightarrow{\overline{a}} 0 & (2) & \mathrm{in}(a).P \xrightarrow{a} P \\
(3) & \mathrm{rd}(a).P \xrightarrow{a} P & (4) & !\mathrm{in}(a).P \xrightarrow{a} P \mid !\mathrm{in}(a).P \\
(5) & \mathrm{inp}(a)?.P\_Q \xrightarrow{a} P & (6) & \mathrm{rdp}(a)?.P\_Q \xrightarrow{a} P \\
(7) & \mathrm{inp}(a)?.P\_Q \xrightarrow{\neg a} Q & (8) & \mathrm{rdp}(a)?.P\_Q \xrightarrow{\neg a} Q \\
(9) & \dfrac{P \xrightarrow{\alpha} P' \quad \alpha \neq \neg a}{P \mid Q \xrightarrow{\alpha} P' \mid Q} & (10) & \dfrac{P \xrightarrow{\neg a} P' \quad Q \not\xrightarrow{\overline{a}}}{P \mid Q \xrightarrow{\neg a} P' \mid Q} \\
(11) & \dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} & (12) & \dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q}
\end{array}
$$

Figure 5.1: Linda operational semantics (symmetric rules omitted)

Ordered Message Output In Figure 5.2 we present the output–rule for the ordered semantics. In case of the ordered semantics the message is immediatly visible in the tuple space. The semantics is called ordered because output messages occure in the tuple space in the order in which they were issued.

$$(13) \quad \mathrm{out}(a).P \quad \xrightarrow{\tau} \quad \langle a \rangle \mid P$$

Figure 5.2: Message sending – ordered semantics

Unordered Message Output In Figure 5.3 we present the output-rules for the unordered semantics. In case of the unordered semantics a sendbox for the message is added to the tuple space (see Rule (14)) and the message is not yet visible in the tuple space. Only after another internal step the sendbox delivers the message to the tuple space (see Rule (15)). The semantics is called unordered because output messages occur in the tuple space in an arbitrary order.

$$\begin{array}{lrcl} (14) & \mathrm{out}(a).P & \xrightarrow{\tau} & \langle\langle a\rangle\rangle \mid P \\ (15) & \langle\langle a\rangle\rangle & \xrightarrow{\tau} & \langle a\rangle \end{array}$$

Figure 5.3: Message sending – unordered semantics

Example 5.3.1 (Linda) *Consider the following program* $P = out(a).inp(a)?\langle b\rangle.0\_\langle c\rangle.0$. *In case of the instantaneous output only the first branch* $\langle b\rangle.0$ *is reachable. In case of the buffered output both branches are reachable. The immediate visiblity of the output is crucial for the construction of the Random Access Machine in the proof of* $\mathbf{L}_o$ *being Turing-powerful.*

5.3.3 Expressivness

For a Linda dialect without testing (rd(a)) and without conditional branching ($s?P\_Q$) the difference between the ordered and the unordered semantics is no longer observable.

Lemma 5.3.2 *For a Linda dialect without testing and without conditional branching the ordered and the unordered semantics are both not Turing powerful.*

The proof for the ordered semantics being Turing powerful in [27] depends on conditional branching. So our first observation is that this proof is no longer valid if we remove testing and conditional branching. For the proof of the unordered semantics a P/T net is constructed which coincides with the Linda program with respect to termination. Then termination for the P/T net is shown to be decideable. The construction of a P/T net without testing and conditional branching is straightforward. Testing and conditional branching introduce an observable difference between initial message (which are always there) and "normal" messages which are created at an arbitrary point after the output operation (in the unordered semantics). Furthermore due to the test for zero (conditional branching) we need to count the number of messages (at least "zero" and "more than zero"). Following the proofs of [27] both semantics can be shown to be not Turing powerful.

5.4 Translation of Abstract Creol Configurations to Linda Configurations

Instead of translating the Creol model to a Linda program we translate Creol configurations with the Creol model being the initial configuration to Linda configurations. Translating the configurations facilitates the proof of the simulation relation between the Creol model and the Linda program. In addition to the translation of the model code we have to provide a translation for the objects, the processes in execution, and the futures. We translate objects as an object lock and a collection of processes sharing this lock. We translate processes in execution by translating the code to be executed. We translate futures to messages.

The crucial step in the modeling of the communication is the abstraction from the runtime labels. To use the results of Busi et al. [27] and to get decidability of termination we need to represent the Linda model by a finite P/T net ,i.e. we are restricted to a finite message alphabet. We give an abstraction that fulfills this requirement. Instead of creating a unique runtime label for each method invocation we identify the method invocation by statically unique labels, i.e. labels that are unique on the level of the model code. For balanced, well-formed programs this identification is sufficient. Due to the balancing the lifespan of a future is restricted to one method invocation only which allows to preserve decidability of termination. For details we refer to Section 5.5.

To translate an abstract Creol configuration to a Linda configuration we have to translate object definitions, objects, and futures into Linda processes and messages:

Translation of artifacts	
Abstract Creol	Linda
object definitions	replication
object states	processes and messages
futures	messages

We present a translation for each of these artifacts. The translation of the abstract Creol configuration to Linda is given by the translation of the individual elements of the Creol configuration.

We develop the translation of the method definitions by starting with the translation for a method body which covers the communication steps and the modeling of non–deterministic choice by parallelism. At this point we do not cover the production of return values or scheduling. The translation of method bodies is integrated into the translation of the method definitions of a single object by adding the production of return values and the scheduling of methods. The translation of the method definitions for the Creol model is given by the parallel composition of the translations of the method definitions of the individual objects.

Active processes are translated using the translation function for method bodies. Pending processes are translated to messages in Linda. Futures are translated to messages. The translation of an abstract Creol configuration is given by the parallel composition of the translation of the individual elements of the configuration.

Pruning Linda does not provide a primitive for internal steps. To facilitate the translation from Creol to Linda we "prune" the Creol configuration from internal steps as far as possible. Pruning of the internal steps is also in line with our intention to model the network communication only. The pruning function $\Downarrow$ takes a Creol configuration and removes as many internal steps as possible. In the end internal steps can only occur in a choice and even there at most in one branch.

$$\begin{array}{rcll}
\Downarrow(ret) & ::= & ret & \\
\Downarrow(\tau) & ::= & \tau & \\
\Downarrow(o.m) & ::= & o.m & \\
\Downarrow(f = o.m) & ::= & f = o.m & \\
\Downarrow(f?) & ::= & f? & \\
\Downarrow(e_1; e_2) & ::= & \downarrow(\Downarrow(e_1); \Downarrow(e_2)) & \\
\Downarrow(e_1 + e_2) & ::= & \downarrow(\Downarrow(e_1) + \Downarrow(e_2)) & \\
\downarrow(e_1; e_2) & ::= & e_2 & \text{if} \quad e_1 = \tau \\
 & ::= & e_1; e_2 & \text{otherwise} \\
\downarrow(e_1 + e_2) & ::= & \tau & \text{if} \quad e_1 = e_2 = \tau \\
 & ::= & e_1 + e_2 & \text{otherwise}
\end{array}$$

Since the choice operator is non–deterministic, the pruning of the Creol configuration does not change the behavior of the program with respect to the network communication. This follows directly from the definition of the pruning. From now on we assume the Creol configurations to be pruned.

5.4.1 Translation of a Single Method

First we give a translation for a method in isolation not taking scheduling or the production of return values into account. These are modeled in the following Section 5.4.2. We introduce messages to deal with anonymous, asynchronous calls $(\perp : o.m)$ denoting the callee o and the method name m. To deal with futures we introduce two messages $(f : o.m)$ and (f). The message $(f : o'.m)$ denotes a call to method m of object o assigned to the future name f. The message (f) denotes the result of a call assigned to the future name f.

Though future names are statically unique modeling method calls and returns this way is ambiguous. Method calls and returns issued by different method invocations might be mixed up. This problem is avoided by restricting to balanced programs only.

Non–deterministic choices are modeled by two processes competing for a designated choice message $(o, +)$ which is statically unique. The processes model the different branches of the choice. At termination the processes issue a termination message $(\overline{o, +})$ allowing the main process to continue. At this point of time we do not care about the production of return values or the scheduling of method invocations.

We take the following two properties into account. Due to the definition of the syntax each method definition ends with a return statement. Due to the pruning (silent) internal steps can only occur in (at most one branch of) a choice $e_1 + e_2$. We assume that the method we are modeling is a method of object o.

Internal Steps Even after pruning, in case of a choice one of the branches can consist of an internal step only. This internal step is translated into the empty process 0.

$$\mathcal{L}(\tau) ::= 0$$

Method termination The end of the method is denoted by the return statement and is (for the time being) translated into the empty process 0.

$$\mathcal{L}(ret) ::= 0$$

Method call An anonymous method call to method m in object o is translated to the generation of a message $(\perp : o, m)$, where o is the callee and m the method name.

$$\mathcal{L}(o.m) ::= \text{out}((\perp : o.m))$$

Future A method call to method m in object o with label f is translated to the generation of a message $(f : o.m)$. Here we abstract from the actual future.

$$\mathcal{L}(f = o.m) ::= \text{out}((f : o.m))$$

Requesting result The (blocking) request of an result to a method call with label f is translated to the consumption of a message (f).

$$\mathcal{L}(f?) ::= \text{in}((f))$$

Sequential composition A sequence of Creol statements is translated into a sequence of Linda statements. Please note that this can only occur as an intermediate step (since each method definition is of the form $e; ret$) and leads to a sequence of communication and choice steps. We lift the definition of the prefix operator "." in a straight-forward manner from single statements to sequences of statements.

$$\mathcal{L}(e_1; e_2) ::= \mathcal{L}(e_1).\mathcal{L}(e_2)$$

Choice We model internal (non–deterministic) choice in Creol by adding (generators for) processes for each branch of the choice in parallel to the method body. Upon arrival at the choice a trigger message for the choice is generated. Both branches compete for this trigger message – modeling the choice.

$$\mathcal{L}(e_1 + e_2) ::= \text{out}((o,+)).\text{in}((\overline{o,+}))|E_1|E_2$$

where $E_x ::= !\text{in}((o,+)).\mathcal{L}(e_x).\text{out}((\overline{o,+})).0$.

Here $(o,+)$ denotes a statically unique label for the choice $+$ in object o denoting the arrival at the choice. The label $(\overline{o,+})$ denotes the completion of the chosen branch.

5.4.2 Translation of a Single Object

The call with label f of a method m on object o denoted by a message $(f : o, m)$ is supposed to produce a future (f). To model the relation between a caller and a future we model each named invocation. The future to be produced is decided at the time of method reception. For each label we add a generator process that creates an instance of a process to execute an invocation of the method. Furthermore we create a generator for processes to deal with anonymous calls.

In a Creol object at most one active process is allowed this is modeled by an object token implemented as a message (o). Only the process that holds the token (modeled by removing the object token from the tuple space) is allowed to execute. At termination the process frees the token again (modeled by adding the object token to the tuple space). The system contains either exactly one token or active process per object.

Initially the object contains the process for the *run*–method. The initial activity holds the object token. This prevents other methods from being scheduled before the initial activity has terminated. Upon termination the *run*–method creates the object token for the first time and adds it to the tuple space.

Named invocation We explicitly model the communications for each named invocation f to assign the proper return value.

$$\mathcal{L}(m) ::= \Pi_f\ !\mathrm{in}((f : o.m)).\mathcal{L}(f, e)|!\mathrm{in}((\perp : o.m)).\mathcal{L}(\perp, e)$$

where $m = e$ is the method definition in o. Here $\Pi_{p \in P}\ p$ denotes the parallel composition of the processes in P.

We extend the definition of $\mathcal{L}$ to reflect the two modes (named and anonymous) of asynchronous calls in our translation function.

$$\begin{array}{rcl}
\mathcal{L}(\perp, \mathit{ret}) & ::= & 0 \\
\mathcal{L}(f, \mathit{ret}) & ::= & \mathrm{out}((f)).0 \\
\mathcal{L}(\gamma, e_1 + e_2) & ::= & \mathrm{out}((o, +)).\mathrm{in}((\overline{o, +}))|E_1|E_2 \\
 & & \text{where} \quad E_x ::= \mathrm{in}((o, +)).\mathcal{L}(\gamma, e_x).\mathrm{out}((\overline{o, +})).0 \\
\mathcal{L}(\gamma, e_1; e_2) & ::= & \mathcal{L}(\gamma, e_1).\mathcal{L}(\gamma, e_2) \\
\mathcal{L}(\gamma, o'.m) & ::= & \mathcal{L}(o'.m) \\
\mathcal{L}(\gamma, f = o'.m) & ::= & \mathcal{L}(f = o'.m) \\
\mathcal{L}(\gamma, f?) & ::= & \mathcal{L}(f?)
\end{array}$$

We only produce a result in case of a named call.

Scheduling At each point in time at most one process can be active in each object. We model this by an access token o for object o. Upon reception of a call to m a new process is spawn to execute the call. The new process first waits for the object token. Reception of the token models scheduling of the method. At the end of its execution the process frees the token.

$$\mathcal{L}(o) ::= \Pi_{m \in o}\ \mathcal{L}(m)$$

where $\mathcal{L}(m)$ is an extension of the above mentioned translation rule $\mathcal{L}(m)$ taking the object lock into account:

$$\mathcal{L}(m) ::= \Pi_f\ !\mathrm{in}((f : o.m)).\mathrm{in}(o).\mathcal{L}(f, e) \mid !\mathrm{in}((\perp : o.m)).\mathrm{in}(o).\mathcal{L}(\perp, e)$$

Furthermore the rules for method termination are extended to free the lock upon method termination.

$$\begin{array}{rcl}\mathcal{L}(\bot, ret) & ::= & \mathrm{out}((o)).0 \\ \mathcal{L}(f, ret) & ::= & \mathrm{out}((f)).\mathrm{out}((o)).0\end{array}$$

Active Behavior To model the active behavior we model the *run*–method as a process. Being the initial activity the *run*–method is modeled as an anonymous call.

$$\mathcal{L}_I(o) ::= \mathcal{L}(\bot, D_o(\mathrm{run}))$$

Please note that the initial activity starts directly with the execution and does not have to grab the object token. In fact the object token is introduced by the *run*-method upon termination.

5.4.3 Translation of a Creol model

A Creol model is the parallel composition of the individual objects and their initial activities.

$$\mathcal{L}(\Theta_I) ::= \Pi_{o\in O}\ (\mathcal{L}(o) \mid \mathcal{L}_I(o))$$

5.4.4 Translation of a Creol configuration

A Creol configuration contains the model definitions, the object locks, the active processes, pending calls and futures. In case an object does not contain an active process the lock message is added to the tuple space. For each pending call a corresponding call message is added to the tuple space. The remaining program code of the active process is translated. Futures are translated to corresponding messages. We focus on the individual parts of the translation of an object.

Pending calls For each pending call to the object the corresponding call message is added to the tuple space.

$$\mathcal{L}((o, \_, \Gamma \cup \{f : m\})) \quad ::= \quad \mathcal{L}((o, \_, \Gamma)) \mid (f : o.m)$$

Object lock and active process In case the object does not contain an active process the object lock message is added to the tuple space. Otherwise the active process is translated by translation of the remaining code to be executed taking the corresponding result to be produced into account.

$$\begin{array}{rcl} \mathcal{L}((o,\epsilon,\emptyset)) & ::= & (o) \\ \mathcal{L}((o,f:e,\emptyset)) & ::= & \mathcal{L}(f,e) \end{array}$$

Future A future is translated to the corresponding future message.

$$\mathcal{L}(f) ::= (f)$$

A Creol configuration is the parallel composition of the object definitions, the objects, the processes and the futures.

$$\mathcal{L}(\Theta) ::= \Pi_{o\in O}\ \mathcal{L}(o) \mid \Pi_{o\in O}\ \mathcal{L}((o,\_,\_)) \mid \Pi_{f\in F}\mathcal{L}(f)$$

Here F denotes the multi-set of all futures.

Example 5.4.1 (Running Example: Translation to Linda) *We revisit our running example to illustrate the translation of Creol models to Linda. We translate the model by translation of the objects and the initial methods.*

$$\begin{array}{rcl} \mathcal{L}(\theta_o) & = & \mathcal{L}(o_1) \mid \mathcal{L}((o_1,\bot:o_1.m_1;ret,\emptyset)) \mid \mathcal{L}(o_2) \mid \mathcal{L}((o_2,\bot:o_2.m_3;ret,\emptyset)) \\ & = & \mathcal{L}(o_1) \mid \mathcal{L}(o_2) \mid \mathcal{L}(\bot:o_1.m_1;ret) \mid \mathcal{L}(\bot,o_2.m_3;ret) \\ & = & \mathcal{L}(o_1) \mid \mathcal{L}(o_2) \mid \mathcal{L}(\bot:o_1.m_1).\mathcal{L}(\bot:ret) \mid \mathcal{L}(\bot,o_2.m_3).\mathcal{L}(\bot,ret) \\ & = & \mathcal{L}(o_1) \mid \mathcal{L}(o_2) \mid \mathcal{L}(o_1.m_1).0 \mid \mathcal{L}(o_2.m_3).0 \\ & = & \mathcal{L}(o_1) \mid \mathcal{L}(o_2) \mid out((\bot:o_1.m_1)).out((o_1)).0 \\ & \mid & out((\bot:o_2.m_3)).out((o_2)).0 \end{array}$$

The translation of an object is the translation of the method definitions. The translation is triggered by the futures of the calls to the method. Please note that due to the unique names of the futures there is a distinct relation between the futures and the method definitions.

$$\begin{array}{rcl} \mathcal{L}(o_1) & = & \mathcal{L}(m_1) \mid \mathcal{L}(m_2) \\ & = & !in((\bot:o_1.m_1)).in((o_1)).\mathcal{L}(\bot,D_{o_1}(m_1)) \\ & \mid & !in((g:o_1.m_2)).in((o_1)).\mathcal{L}(g,D_{o_1}(m_2)) \\ & = & !in((\bot:o_1.m_1)).in((o_1)).\mathcal{L}(\bot,\rhd f=o_2.m_4;\rhd f?;ret) \\ & \mid & !in((g:o_1.m_2)).in((o_1)).\mathcal{L}(g,ret) \\ & = & !in((\bot:o_1.m_1)).in((o_1)).\mathcal{L}(\rhd f=o_2.m_4).\mathcal{L}(\rhd f?).\mathcal{L}(\bot,ret) \\ & \mid & !in((g:o_1.m_2)).in((o_1)).out((g)).out((o_1)).0 \\ & = & !in((\bot:o_1.m_1)).in((o_1)).out((f:o_2.m_4)).in((f)).out((o_1)).0 \\ & \mid & !in((g:o_1.m_2)).in((o_1)).out((g)).out((o_1)).0 \end{array}$$

The translation of object o_2 is similar.

$$\begin{array}{rcl}\mathcal{L}(o_2) & = & !in((\bot : o_2.m_3)).in((o_2)).out((g : o_1.m_2)).in((g)).out((o_2)).0 \\ & | & !in((f : o_2.m_4)).in((o_2)).out((f)).out((o_2)).0\end{array}$$

Technical note: In general the process $!in((\bot : o_1.m_2)).in((o_1)).\mathcal{L}(\bot, e)$ resp. $!in((\bot : o_2.m_4)).in((o_2)).\mathcal{L}(\bot, e)$ would be part of the translation, too. Since there are no anonymous calls to m_2 at o_1 resp. m_4 at o_2 we omit the translation of the processes for brevity.

Now we present an execution of the model in Linda. We omit terminated Linda processes, i.e. processes 0.

$$\begin{array}{rl}
 & \mathcal{L}(o_1) \mid out((\bot : o_1.m_1)).out((o_1)).0 \\
 | & \mathcal{L}(o_2) \mid out((\bot : o_2.m_3)).out((o_2)).out((o_2)).0 \\
 = & \mathcal{L}(o_1) \mid (\bot : o_1.m_1) \mid out((o_1)).0 \\
 | & \mathcal{L}(o_2) \mid out((\bot : o_2.m_3)).out((o_2)).out((o_2)).0 \\
 = & \mathcal{L}(o_1) \mid out((o_1)).0 \mid in((o_1)).out((f : o_2.m_4)).in((f)).out((o_1)).0 \\
 | & \mathcal{L}(o_2) \mid out((\bot : o_2.m_3)).out((o_2)).0 \\
 = & \mathcal{L}(o_1) \mid (o_1) \mid in((o_1)).out((f : o_2.m_4)).in((f)).out((o_1)).0 \\
 | & \mathcal{L}(o_2) \mid out((\bot : o_2.m_3)).out((o_2)).0 \\
 = & \mathcal{L}(o_1) \mid out((f : o_2.m_4)).in((f)).out((o_1)).0 \\
 | & \mathcal{L}(o_2) \mid out((\bot : o_2.m_3)).out((o_2)).0 \\
 = & \mathcal{L}(o_1) \mid in((f)).out((o_1)).0 \\
 | & \mathcal{L}(o_2) \mid (f : o_2.m_4) \mid out((\bot : o_2.m_3)).out((o_2)).0 \\
 = & \mathcal{L}(o_1) \mid in((f)).out((o_1)).0 \\
 | & \mathcal{L}(o_2) \mid (f : o_2.m_4) \mid (\bot : o_2.m_3) \mid out((o_2)).0 \\
 = & \mathcal{L}(o_1) \mid in((f)).out((o_1)).0 \\
 | & \mathcal{L}(o_2) \mid (f : o_2.m_4) \mid (\bot : o_2.m_3) \mid (o_2) \\
 = & \mathcal{L}(o_1) \mid in((f)).out((o_1)).0 \\
 | & \mathcal{L}(o_2) \mid (f : o_2.m_4) \mid (o_2) \mid in((o_2)).out((g : o_1.m_2)).in((g)).out((o_2)).0 \\
 = & \mathcal{L}(o_1) \mid in((f)).out((o_1)).0 \\
 | & \mathcal{L}(o_2) \mid (f : o_2.m_4) \mid out((g : o_1.m_2)).in((g)).out((o_2)).0 \\
 = & \mathcal{L}(o_1) \mid (g : o_1.m_2) \mid in((f)).out((o_1)).0 \\
 | & \mathcal{L}(o_2) \mid (f : o_2.m_4) \mid in((g)).out((o_2)).0 \\
 = & \mathcal{L}(o_1) \mid in((o_1)).out((g)).out((o_1)).0 \mid in((f)).out((o_1)).0 \\
 | & \mathcal{L}(o_2) \mid in((o_2)).out((f)).out((o_2)).0 \mid in((g)).out((o_2)).0
\end{array}$$

Here the execution is stuck. Both $\mathcal{L}(o_1)$ and $\mathcal{L}(o_2)$ only consume messages. All remaining processes wait for input without any messages present in the configuration. Please note that the terminal abstract Creol configuration of Ex-

ample 5.2.13 translates to the second to last Linda configuration. We elaborate on this relation in the next section.

5.5 Termination

We prove a correspondence between the abstract semantics of pre-processed Creol models and their Linda counterparts. In order to formalise the correspondence between the transitions of Creol models and their Linda counterparts we introduce a reduction semantics for Linda translations of Creol models. The reduction semantics is defined with respect to congruence rules which capture implementation details of the translation of Creol models to Linda, e.g. lock message for scheduling and control messages to resolve choice.

As explained in Section 5.4 we abstract from the unique runtime labels of Creol in Linda. With the intermediate semantics we move this abstraction to the Creol level making the abstraction more comprehensible to the reader. Instead of storing unique runtime labels we use statically unique call labels. In case of a request of a future f the request can only be met if a call for f is pending and a future token is available in the configuration. In case the request can be met both the future and the syntactic label are removed. Due to the run–to–completion semantics and the balancing of the programs the lifespan of a future is restricted to one method invocation. This makes explicit runtime labels superfluous.

We introduced auxiliary constructs, e.g. code replication, object lock messages and choice messages, to translate certain aspects of the Creol execution like method scheduling, choice and method in-lining. Due to these auxiliary constructs the execution of a Linda program can contain intermediate steps and states that can not be directly mapped to the Creol configuration to address these differences we define congruence rules on Linda configurations. We present these rules in Figure 5.4.

To simulate the execution of a Creol configuration in Linda, we need to give for each step in Creol a corresponding step in Linda. Some of the steps, e.g. method scheduling or choice, need preparation steps first which are provided by the congruence rules. Figure 5.5 summarizes the simulation steps by providing for each Creol step the corresponding Linda counterpart.

The reduction semantics for translated Creol configurations is given by rules of the form:

$$\frac{P_1 \equiv P_1' \quad P_2 \equiv P_2' \quad P_1' \longrightarrow P_2'}{P_1 \longrightarrow P_2}$$

where the transitions are restricted to the simulation steps of Figure 5.5.

Congruence Rules			
$!\mathrm{in}((f:o.m)).P \mid (f:o.m)$	$\equiv$	$!\mathrm{in}((f:o.m)).P \mid P$	method inlining
$\mathrm{out}((o,+)).P$	$\equiv$	$P \mid (o,+)$	trigger choice
$\mathrm{out}((\overline{o,+})).P$	$\equiv$	$P \mid (\overline{o,+})$	signal end of choice
$\mathrm{in}((\overline{o,+})).P \mid (\overline{o,+})$	$\equiv$	P	continue after choice
$\mathrm{out}((o)).P$	$\equiv$	$P \mid (o)$	release object lock

Figure 5.4: Congruence rules

Simulation Steps		
Method scheduling	Input lock message	$\mathrm{in}(o).P \mid (o) \longrightarrow P$
Method termination	Output future message	$\mathrm{out}((f)).P \longrightarrow P \mid (f)$
Method call	Output call message	$\mathrm{out}((\bot{:}o.m)).P \longrightarrow P \mid (\bot{:}o.m)$
Future	Output call message	$\mathrm{out}((f{:}o.m)).P \longrightarrow P \mid (f{:}o.m)$
Requesting result	Input future message	$\mathrm{in}((f)).P \mid (f) \longrightarrow P$
Choice	Choice	$!\mathrm{in}((o,+)).P_1 \mid !\mathrm{in}((o,+)).P_2 \mid (o,+)$
		$\longrightarrow !\mathrm{in}((o,+)).P_1 \mid !\mathrm{in}((o,+)).P_2 \mid P_x$

Figure 5.5: Simulation steps

Given the above reduction semantics for translated Creol configurations we have the following correspondence between Creol configurations and their Linda counterparts:

Theorem 5.5.1 (Bisimulation (Creol vs. Linda)) *For any two valid abstract Creol configurations Θ and Θ', $\Theta \longrightarrow \Theta'$ iff $\mathcal{L}(\Theta) \longrightarrow \mathcal{L}(\Theta')$.*

Proof: The proof follows by a straightforward case distinction on the Creol steps and the corresponding reduction steps in Linda depicted in the tables above. □

Corollary 5.5.2 (Termination equivalence (Creol vs. Linda)) *A Creol program terminates iff the corresponding Linda model terminates.*

Proof: It suffices to observe that termination of a translated Creol configuration is preserved under the reduction steps. □

Deadlock Detection To summarize the process of deadlock detection for Creol models, we have the following steps:

1. As explained in Section 5.2 we add divergence to distinguish normal termination and global deadlock.
2. We translate the model to a Linda program as explained in Section 5.4.
3. We apply the technique of Busi et al. [27] to check the Linda program for global deadlocks.

In case the Linda program is free of deadlocks our Creol model is free of deadlocks.

5.6 Conclusion

We have presented an Actor-like subset of Creol and we have given two semantics for this subset: a concrete semantics which resembles the semantics of full Creol including the generation of unique run-time labels and a semantics which abstracts from these labels and yields a finite representation of Creol configurations in P/T nets. We have proven that executions in the two semantics coincide with respect to termination.

To obtain a representation of a Creol model in a P/T net we presented a translation of Creol configurations to a process algebra based on the coordination language Linda. For this process algebra there are two different semantics which differ in the expressiveness of the language. We argued that for the subset of the process algebra we are using for our translation these semantics coincide with respect to expressiveness and that both are not Turing powerful. We have proven that executions of the translation of the Creol model in the process algebra and executions of the model itself in the abstract semantics coincide with respect to termination.

The preservation of the termination property allows us to apply the formalism of Busi et al. [27] to reduce deadlock detection to termination detection for finite P/T nets.

Compared to our previous work on termination detection for concurrent objects communicating synchronously [43] the asynchronous setting based on futures requires a different approach. In [47] various decidability results are introduced for different classes of infinite state systems communicating via FIFO queues. The specific distinguishing feature of the infinite state systems considered in this paper concerns the integration of asynchronous communication in Actor-like languages with futures which gives rise to the unbounded generation of fresh names. In [54] a deadlock analysis is presented of of a calculus with object groups based on abstract descriptions of methods behaviours. It however does not provide a full treatment of futures.

Future work

The main challenge for future work is the investigation into the decidability of asynchronous communication based on futures in the context of cooperative multi-tasking, e.g., the processor release statements of the full Creol language. We give some directions how to extend our subset of Creol and the semantic consequences of these extensions.

Conditional Branching Besides the blocking request for a result of a call Creol features primitives to poll a future, i.e. not requesting the result itself but just the information whether or not a result has already been calculated. To model such a command we can use the rdp(a)?.P_Q primitive of Linda. In this case the difference between the ordered and the unordered semantics becomes visible again and we have to opt for the unordered semantics to keep termination decidable.

Conditional Scheduling A condition on the existence of futures can be used to trigger a processor release and rescheduling. The await–statement denotes such a conditional scheduling point. In case all futures, given in a guard expression, are available the process continues execution otherwise the process suspends itself to wait for the missing futures to be calculated. In the meantime another process is scheduled and executed. We can model the conditional scheduling on one future by the rdp(a)?.P_Q primitive of Linda. Conditional scheduling on a number of futures can be modeled by a sequence of the conditional branching primitives in Linda. Due to the abstraction of the run-time labels we need to check for the existence of all requested futures every time we check.

In this case we lose precision with respect to the abstraction. Now futures of different method invocations can be mixed up. In this case the problem is inherent to the conditional scheduling and cannot be avoided.

Technical Improvement The size of the resulting P/T net can be reduced if Creol concepts like co-interfaces are taken into account for the translation.

Creol objects are typed by interfaces. The co-interfaces restrict the set of possible callers of a method by requiring callers to implement the co-interfaces of the method. Switching to interfaces and co-interfaces it suffices to model caller–method–pairs for valid combinations (with respect to the co-interfaces).

Object Creation We can model finite object creation by object activation, i.e. an objects exists in the initial configuration but is deactivate until an activation message is received. Object activation can be realized by activation messages similar to the scheduling tokens. In this case the activation token blocks the process for the initial *run*-method until the object creator has send the activation token.

Direct Translation to P/T net Using only parts of the process algebra Busi et al. [27] introduced the direct translation of abstract Creol to P/T nets might yield a simpler translation than using an intermediate translation to Linda.

Appendix A

Proofs

A.1 Wellformedness of generated sequences

In this section we present an extensive proof of Theorem 3.4.2.

Proof: We prove this by induction on the length of the derivation. We treat the following cases:

Let $I_c = \emptyset$ and $t_c \in L$ in the following derivation

$$(I, L) : G \cup \{S^t\} \Rightarrow t_r^m\ (I \cup \{t_c\}, L) : G \cup \{S^t\} \Rightarrow^* t_r^m W$$

(where m is a synchronised method of c). By the induction hypothesis we have that

- W is synchronised and
- $I \cup \{t_c\} \cup \mathrm{Lock}(W) = L$.

It follows that $t_r^m W$ is synchronised and that $I \cup \mathrm{Lock}(t_r^m W) = I \cup \{t_c\} \cup \mathrm{Lock}(W) = L$.

Let $I_c = L_c = \emptyset$ in the following derivation

$$(I, L) : G \cup \{B^t\} \Rightarrow t_r^m\ (I \cup \{t_c\}, L \cup \{t_c\}) : G \cup \{r^t\} \Rightarrow^* t_r^m W$$

(where m is a synchronised method of c). By the induction hypothesis we have that

- W is synchronised and
- $I \cup \{t_c\} \cup \mathrm{Lock}(W) = L \cup \{t_c\}$.

It follows that $t^m_r W$ is synchronised and that $I \cup \mathrm{Lock}(t^m_r W) = I \cup \mathrm{Lock}(W) = L$ (note that $I_c = L_c = \emptyset$ and in $t^m_r W$ all calls by t have a matching return, so $\mathrm{Lock}(t^m_r W) = \mathrm{Lock}(W)$).

Next we treat the case

$$(I, L) : G \cup \{r^t\} \Rightarrow (I, L) : G \cup \{B^t\}\ t_r \Rightarrow^* W t_r$$

By the induction hypothesis we have that

- W is synchronised and
- $I \cup \mathrm{Lock}(W) = L$.

So it suffices to observe that, since there exist no pending calls of t in W, we have $\mathrm{Lock}(W t_r) = \mathrm{Lock}(W)$.

As a final case we treat the derivation:

$$(I, L) : G_1 \circ G_2 \Rightarrow (I, L') : G_1\ (L', L) : G_2 \Rightarrow^* W$$

This derivation is decomposed to $(I, L') : G_1 \Rightarrow^* W_1$ and $(L', L) : G_2 \Rightarrow^* W_2$, where $W = W_1 W_2$. By the induction hypothesis we have that

- W_1 and W_2 are synchronised and
- $I \cup \mathrm{Lock}(W_1) = L'$ and $L' \cup \mathrm{Lock}(W_2) = L$.

We first argue that $W = W_1 W_2$ is synchronised. Let t^m_r be a synchronised call in W, with m a synchronised method of c. If t^m_r appears in W_1 then there exists no preceding pending call to a synchronised method of c by another thread because W_1 is synchronised. On the other hand, if t^m_r appears in W_2 then there exists no preceding pending call (to a synchronised method of c) by another thread *in* W_2 because W_2 is synchronised. There also does not exist such a call by a thread t' different from t in W_1 because $I \cup \mathrm{Lock}(W_1) = L'$ implies $t'_c \in L'$, which in turn rules out the call t^m_r because W_2 is synchronised.

Furthermore, if $t_c \in I$ then there exists no call in W to a synchronised method of c by another thread because both W_1 and W_2 are synchronised.

Finally, $I \cup \mathrm{Lock}(W) = I \cup \mathrm{Lock}(W_1) \cup \mathrm{Lock}(W_2) = L' \cup \mathrm{Lock}(W_2) = L$. Note that indeed $\mathrm{Lock}(W) = \mathrm{Lock}(W_1) \cup \mathrm{Lock}(W_2)$ because if W_2 contains a matching return t_r for a pending call t^m_r in W_1 then $r^t \in G_2$. But this is ruled out by $(I, L) : G_1 \circ G_2 \Rightarrow (I, L') : G_1\ (L', L) : G_2$ because r^t cannot be generated by a split. $\square$

A.2 Existence of derivation

In this section we present an extensive proof of Lemma 3.5.1.

Proof: We prove the lemma by induction on the length of the word W.

Base Case $W = \epsilon$ is straightforward by application of rule $(I, I) : G ::= \epsilon$

Induction Step Let $W = w_1, \ldots, w_n, w_{n+1}$ be the well-formed synchronised sequence. Since w_{n+1} can be either a call or a return there are two cases to deal with. According to the induction hypothesis there exists a well-formed sequence $w'_1, \ldots, w'_n$ such that $G_0 \Rightarrow^* w'_1, \ldots, w'_n$, and $w'_1, \ldots, w'_n \approx w_1, \ldots, w_n$.

First we consider the case that w_{n+1} is a method call t^m_r. We prefix the derivation $G_0 \Rightarrow^* w'_1, \ldots, w'_n$ by a composition step: $G_0 \Rightarrow G_0\ G_0 \Rightarrow^* w'_1, \ldots, w'_n G_0$, and apply the rules $G_0 ::= t^m_r G_0$ and $G_0 ::= \epsilon$ to obtain $G_0 \Rightarrow^* w'_1, \ldots, w'_n, w_{n+1}$.

The proof of the equivalence $w'_1, \ldots, w'_n, w_{n+1} \approx w_1, \ldots, w_n, w_{n+1}$ is straightforward. Appending the same call to both sequences $w'_1, \ldots, w'_n$ and $w_1, \ldots, w_n$ preserves the equality of projection and the equality of the lock sets.

Next we consider the case that w_{n+1} is a return t_r. Let $w'_i = t^m_r$ be the matching call in $w'_1, \ldots, w'_n$. For this call we can decompose the derivation into $G_0 \Rightarrow^* \ldots G \cup \{S^t\} \ldots \Rightarrow \ldots t^m_r\ G \cup \{S^t\} \ldots \Rightarrow^* w'_1, \ldots, w'_n$. In this derivation we replace the step $\ldots G \cup \{S^t\} \ldots \Rightarrow \ldots t^m_r G \cup \{S^t\} \ldots$ by the following sequence of steps

$$\ldots G \cup \{S^t\} \ldots \Rightarrow \ldots G \cup \{B^t\} \ldots \Rightarrow \ldots t^m_r\ G \cup \{r^t\} \ldots \Rightarrow \ldots t^m_r\ G \cup \{B^t\}\ t_r \ldots$$

Note that in the derivation $\ldots t^m_r\ G \cup \{S^t\} \ldots \Rightarrow^* w'_1, \ldots, w'_n$ the non-terminal S^t can be replaced by B^t since after the call t^m_r the thread t only generates a balanced sequence of calls and returns. Therefore we obtain a derivation

$$G_0 \Rightarrow^* w'_1, \ldots, w'_i, w'_{i+1}, \ldots, w'_k, t_r, w'_{k+1}, \ldots, w'_n$$

where $G \cup \{B^t\} \Rightarrow^* w'_{i+1} \ldots w'_k$. Due to the nested nature of the method calls (and the grammar rules) t_r is added to $w'_1, \ldots, w'_n$ in such a way that it appears at the end of the projection of $w'_1, \ldots, w'_i, w'_{i+1}, \ldots, w'_k, t_r, w'_{k+1}, \ldots, w'_n$ to t like it does for the projection of $w'_1, \ldots, w'_n, t_r$ to t preserving the equality of the projections. Since the same return is added to both sequences the equality of the lock sets is preserved. This establishes $w'_1, \ldots, w'_i, w'_{i+1}, \ldots, w'_k, t_r, w'_{k+1}, \ldots, w'_n \approx w'_1, \ldots, w'_n, t_r$. □

A.3 Soundness of lock handling

In this section we present an extensive proof of Lemma Lemma 3.5.2.

Proof: Instead of proving the lemma directly we prove a more general statement: If $G_0 \Rightarrow^* W$ with W synchronised with respect to I, then $(I, I \cup \mathrm{Lock}(W)) : G_0 \Rightarrow^* W$.

We prove the lemma by induction on the length of the derivation $G_0 \Rightarrow^* W$.

Base Case $G ::= \epsilon$ is straightforward by application of rule $(I, I) : G ::= \epsilon$.

Induction Step We first treat the following cases of synchronised method calls and returns:

Pending Synchronised Call Let m be a synchronised method in class c and $G \cup \{S^t\} \Rightarrow t_r^m\ G \cup \{S^t\} \Rightarrow^* t_r^m W$ with $t_r^m W$ is synchronised with respect to I. Due to t_r^m being a pending call we derive that W is synchronised with respect to $I \cup \{t_c\}$. By the induction hypothesis we get $(I \cup \{t_c\}, I \cup \{t_c\} \cup \mathrm{Lock}(W)) : G \cup \{S^t\} \Rightarrow^* W$. By application of rule $(I, I \cup \mathrm{Lock}(W)) : G \cup \{S^t\} ::= (I, I \cup \mathrm{Lock}(W)) : G \cup \{S^t\}$ in case $t_c \in I_c$, resp. application of rule $(I, I \cup \mathrm{Lock}(W)) : G \cup \{S^t\} ::= (I \cup \{t_c\}, I \cup \mathrm{Lock}(W)) : G \cup \{S^t\}$ with $t_c \in \mathrm{Lock}(W)$ otherwise, we get a derivation $(I, I \cup \mathrm{Lock}(W)) : G \cup \{S^t\} \Rightarrow^* t_r^m W$.

Matching Synchronised Call For the next case let m be a synchronised method in class c and $G \cup \{B^t\} \Rightarrow t_r^m\ G \cup \{r^t\} \Rightarrow^* t_r^m W$ with $t_r^m W$ is synchronised with respect to I. Due to $t_r^m W$ being synchronised with respect to I we conclude W is synchronised with respect to $I \cup \{t_c\}$. By the induction hypothesis we get $(I \cup \{t_c\}, I \cup \{t_c\} \cup \mathrm{Lock}(W)) : G \cup \{r^t\} \Rightarrow^* W$. By application of rule $(I, I \cup \mathrm{Lock}(W)) : G \cup \{B^t\} ::= (I, I \cup \mathrm{Lock}(W)) : G \cup \{r^t\}$ in case $t_c \in I_c$, resp. application of rule $(I, I \cup \mathrm{Lock}(W)) : G \cup \{B^t\} ::= (I \cup \{t_c\}, I \cup \{t_c\} \cup \mathrm{Lock}(W)) : G \cup \{r^t\}$ otherwise, we get a derivation $(I, I \cup \mathrm{Lock}(W)) : G \cup \{B^t\} \Rightarrow^* t_r^m W$.

Return of a Synchronised Method For the next case let t_r be a return to a call of a synchronised method in class c and $G \cup \{r^t\} \Rightarrow G \cup \{B^t\}\ t_r \Rightarrow^* W t_r$ with $W t_r$ is synchronised with respect to I. Due to $t_r^m W$ being synchronised with respect to I we conclude W is synchronised with respect to I. By the induction hypothesis we get $(I, I \cup \mathrm{Lock}(W)) : G \cup \{B^t\} \Rightarrow^* W$. Since W is derived from $G \cup \{B^t\}$ it does not contain a pending call of t. It follows that

$\text{Lock}(Wt_r) = \text{Lock}(W)$. We conclude that $(I, I \cup \text{Lock}(Wt_r)) : G \cup \{r^t\} \Rightarrow (I, I \cup \text{Lock}(Wt_r)) : G \cup \{B^t\}\ t_r \Rightarrow^* Wt_r$

We treat composition as the final case

Composition $G_1 \circ G_2 \Rightarrow G_1 G_2 \Rightarrow^* W$ with W is synchronised with respect to I. It follows that $G_i \Rightarrow^* W_i$ $(i = 1, 2)$, with $W = W_1 W_2$, W_1 is synchronised with respect to I and W_2 is synchronised with respect to $I \cup \text{Lock}(W_1)$. By the induction hypothesis we get derivations $(I, I \cup \text{Lock}(W_1)) : G_1 \Rightarrow^* W_1$ and $(I \cup \text{Lock}(W_1), I \cup \text{Lock}(W_1) \cup \text{Lock}(W_2)) : G_2 \Rightarrow^* W_2$ We have that $I \cup \text{Lock}(W_1) \cup \text{Lock}(W_2) = I \cup \text{Lock}(W)$ as argued in the proof of theorem 1. So we conclude that $(I, I \cup \text{Lock}(W)) : G_1 \circ G_2 \Rightarrow^* W$ $\square$

Abstract

Formal methods provide the foundation to reason about systems and their characteristics, e.g. safety or security properties. To be able to reason about systems and to provide valid statements about the system the formal methods must provide means to address key features of the language, e.g. concurrency or object creation. These features introduce complexity to systems and the formal methods to reason about these systems.

Both the development of hardware, e.g. multicore processors or the increase of available memory, and the development of software, e.g. networked and integrated systems and the introduction of high-level programming languages like *Java* and C$^\sharp$, lead to an increased usage of the aforementioned features. New concepts to address problems like concurrency and delegation, e.g. futures and promises, are developed and promoted. Formal methods need to include these features to be capable of reasoning about state of the art software systems.

In this thesis, we show how to address the complexity introduced by modern software systems, e.g. structural complexity introduced by object creation and behavioural complexity introduced by multi-threading. We present formalisms to check interesting properties, e.g. deadlock freedom or termination, of such systems. Furthermore we show how to extend a calculus to reason about systems based on active objects with futures and promises.

In the first part of the thesis we address complexity introduced by object creation. Object creation is an abstraction from the underlying representation of objects used for example in object-oriented programming languages like *Java* or C$^\sharp$. For practical purposes it is important to be able to specify and verify properties of objects at the abstraction level of the programming language. We give a representation of a weakest precondition calculus for abstract object creation in dynamic logic which allows to both specify and verify properties of objects at the abstraction level of the programming language. We generalize this approach to allow for symbolic execution to integrate our approach to the setting of the KeY theorem prover.

In the second part of this thesis we address complexity introduced by multi-threading. Multi-threaded programs show complex behaviour due to the interleaving of activities of the individual processes and the sharing of state among these processes. One example of such complex behaviour is the so-called deadlock. Deadlock describes a situation in which concurrent processes share resources. Though shared among the processes a single access to a resource is exclusive. Depending on the order the processes are interleaved the deadlock may or may not arise. The number of interleavings of the processes is in general not bound which makes analysis hard. We present a formalism to reason about deadlock in multi-threaded systems. The formalism focuses on the control flow of such systems and abstracts from data.

In the third part of this thesis we extend a calculus to reason about active objects with futures and promises. We present an open semantics for the core of the *Creol* language including first-class futures and promises. A future acts as a proxy for, or reference to, the delayed result of a computation. As the consumer of the result can proceed its own execution until it actually needs the result, futures provide a natural, lightweight, and transparent mechanism to introduce parallelism into a language. A promise is a generalization of a future as it allows for delegation with respect to which process performs the computation. The formalization is given as a typed, imperative object calculus to facilitate the comparison with the multi-threaded concurrency model of object-oriented languages, e.g. *Java*.

We close the third part of this thesis by presenting a technique to detect deadlocks in concurrent systems of active objects. Our technique is based on a translation of the system to analyse into a P/T net and the application of a technique to detect termination in such P/T nets. We illustrate our technique by application to an Actor-like subset of the *Creol* language featuring asynchronous calls using futures as means of communication. The so-called discipline of cooperative multi-tasking within an object as found in *Creol* can lead to deadlock. Our technique can be applied to detect such deadlocks.

Samenvatting

Formele methoden vormen de wiskundige basis voor het redeneren over de correctheid van computer programma's. Optimaal gebruik van recente ontwikkelingen in hardware, met name de ontwikkeling van multicore processoren, vereist software die gestructureerd is op basis van parallelle processen. Dergelijke structurering echter vergroot de complexiteit van software die alleen door gebruik van formele methoden te beheersen is.

In dit proefschrift worden verschillende formele methoden voor de analyse van verschillende vormen van parallelle object georiënteerde software geintroduceerd en bestudeerd. Centraal hierbij staan programmeerconcepten voor de beschrijving van hoe nieuwe processen dynamisch worden gecreëerd en hoe processen communiceren en synchroniseren.

In het eerste gedeelte van dit proefschrift wordt eerst een methode beschreven voor de analyse van hoe objecten gecreëerd worden in talen als Java. Object creatie vormt één van de basismechanismen van object georiënteerde software voor het programmeren van dynamische structuren en ligt ten grondslag aan de creatie van nieuwe parellelle processen in de taal Java .

Vervolgens wordt in dit proefschrift een methode beschreven voor het redeneren over de synchronisatie van parallelle processen in Java. Een belangerijke eigenschap van de correcte synchronisatie van parallelle processen is de afwezigheid van "deadlock", een situatie waarin een aantal processen op elkaar wachten en niet verder uitgevoerd kunnen worden. Voor een volledige automatisering van de beschreven methode voor het detecteren van dergelijke "deadlocks" wordt er geabstraheerd van data en beperkt tot een vast eindig aantal parallelle processen.

De complexiteit van parallelle processen zoals beschreven in de taal Java komt grotendeels doordat deze processen losgekoppeld zijn van de objecten zelf. De data van objecten wordt door de parallelle processen gedeeld waardoor het in het algemeen zeer moeilijk is het effect van deze processen te analyseren. Nieuwe object georiënteerde programmeertalen als Erlang en Scala beschrijven daarentegen processen als het gedrag van een object zelf. Het resulterende model van een object staat bekend als een ”actor” of ”active object”. In het derde en laatste gedeelte van het proefschrift wordt een calculus geintroduceerd voor de algemene beschrijving van het gedrag van systemen bestaande uit verachillende ”actors”. Voorts wordt een methode beschreven voor de analyse van verschillende vormen van communicatie en synchronisatie tussen ”actors”.

Curriculum Vitae

1979 Born on 18th of January in Kiel, Germany

1989-1998 High School (Gymnasium), Kiel, Germany

1999-2006 Diplom (equivalent to master's degree) in Computer Science, Christian-Albrechts-Universität zu Kiel, Germany

Major in Theoretical Computer Science

Thesis title: *Cloning and Processes*

2006-2008 PhD student at Christian-Albrechts-Universität zu Kiel, Germany, supervised by Prof. Dr. Willem-Paul de Roever

2006-2009 Scientific Staff Member of the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services*

2008-2010 PhD student at Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands, supervised by Prof. Dr. Frank S. de Boer

2010 Scientific Staff Member at Universiteit Leiden, The Netherlands

2011- IT-consultant, ITech Progress GmbH, Ludwigshafen, Germany

Bibliography

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, New York, 1st edition, 1996.

[2] M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Proc. 7 th Int. Conf. Theory and Practice of Software*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer, 1997.

[3] E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen. Behavioral interface description of an object-oriented language with futures and promises. Technical Report 364, University of Oslo, Dept. of Computer Science, 2007.

[4] E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen. Behavioral interface description of an object-oriented language with futures and promises. *Journal of Logic and Algebraic Programming*, 78(7):491–518, 2009.

[5] E. Ábrahám, A. Grüner, and M. Steffen. Abstract Interface Behavior of Object-Oriented Languages with Monitors. *Theory of Computing Systems*, 43(3):322–361, 2008.

[6] E. Ábrahám, A. Grüner, and M. Steffen. Heap-Abstraction for an Object-Oriented Calculus with Thread Classes. *Journal of Software and Systems Modelling*, 7(2):177–208, 2008.

[7] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, 1986.

[8] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. Towards a Theory of Actor Computation. In W. R. Cleaveland, editor, *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 565–579. Springer-Verlag, Heidelberg, 1992.

[9] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[10] W. Ahrendt, F. S. de Boer, and I. Grabe. Abstract Object Creation in Dynamic Logic. In A. Cavalcanti and D. Dams, editors, *FM2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 612–627. Springer-Verlag, Berlin, 2009.

[11] Alice project home page. `http://www.ps.uni-saarland.de/alice/`.

[12] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In OOPSLA1999 [85], pages 314–324.

[13] M. Alpuente and G. Vidal, editors. *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, volume 5079 of *Lecture Notes in Computer Science*. Springer, 2008.

[14] R. Alur and P. Madhusudan. Visibly pushdown languages. In L. Babai, editor, *STOC*, pages 202–211. ACM, 2004.

[15] P. America. Issues in the Design of a Parallel Object-Oriented Language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[16] P. America and F. S. de Boer. Reasoning about dynamically evolving process structures. *Formal Asp. Comput.*, 6(3):269–316, 1994.

[17] V. Arslan, P. Eugster, P. Nienaltowski, and S. Vaucouleur. SCOOP — Concurrency Made Easy. In J. Kohlas, B. Meyer, and A. Schiper, editors, *Dependable Systems: Software, Computing, Networks*, volume 4028 of *Lecture Notes in Computer Science*, pages 82–102. Springer-Verlag, Berlin, 2006.

[18] H. C. Baker, Jr. and C. Hewitt. The Incremental Garbage Collection of Processes. In *Proceedings of the 1977 Symposium on Artifical Intelligence and Programming Languages*, volume 12(8) of *ACM SIGPLAN Notices*, pages 55–59. ACM, New York, 1977.

[19] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.

[20] B. Beckert, V. Klebanov, and S. Schlager. Dynamic Logic. In Beckert et al. [19], pages 69–177.

[21] B. Beckert and A. Platzer. Dynamic Logic with Non-rigid Functions. In U. Furbach and N. Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 2006.

[22] N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstraction for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, 2004.

[23] D. L. Botlan, G. Tack, A. Rossberg, A. Rossberg, D. Le, B. G. Tack, T. Brunklaus, T. Brunklaus, G. Smolka, and G. Smolka. Alice Through the Looking Glass. In H.-W. Loidl, editor, *Trends in Functional Programming*, volume 5, chapter 6, pages 79–96. Intellect Books, Bristol, 2006.

[24] A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In R. Ramanujam and S. Sen, editors, *FSTTCS*, volume 3821 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2005.

[25] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *Int. J. Found. Comput. Sci.*, 14(4):551–, 2003.

[26] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In G. Fox, K. Schauser, and M. Snir, editors, *Proceedings of the ACM 1999 conference on Java Grande (JAVA '99)*, pages 129–141. ACM, New York, 1999.

[27] N. Busi, R. Gorrieri, and G. Zavattaro. On the expressiveness of Linda coordination primitives. *Information and Computation*, 156(1-2):90–121, 2000.

[28] D. Caromel. Service, asynchrony and wait-by-necessity. *Journal of Object-Oriented Programming*, 2(4):12–22, 1990.

[29] D. Caromel. Toward a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, 1993.

[30] D. Caromel and L. Henrio. *A Theory of Distributed Objects. Asynchrony — Mobility — Groups — Components.* Springer-Verlag, Berlin, 2005.

[31] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous Sequential Processes. Research Report 4753 (version 2), INRIA Sophia-Antipolis, 2003.

[32] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and Deterministic Objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, volume 39(1) of *ACM SIGPLAN Notices*, pages 123–134. ACM, New York, 2004.

[33] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency, Practice and Experience*, 10(11-13):1043–1061, 1998.

[34] D. Carotenuto, A. Murano, and A. Peron. 2-visibly pushdown automata. In *Developments in Language Theory*, volume 4588 of *Lecture Notes in Computer Science*, pages 132–144. Springer, 2007.

[35] S. Chaki, E. M. Clarke, N. Kidd, T. W. Reps, and T. Touili. Verifying concurrent message-passing c programs with recursive calls. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 334–349. Springer, 2006.

[36] R. Chandra. *The COOL Parallel Programming Language: Design, Implementation, and Performance.* PhD thesis, Stanford University, 1995.

[37] R. Chandra, A. Gupta, and J. L. Hennessy. COOL: A Languange for Parallel Programming. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing: 2nd Workshop Papers*, Research Monographs in Parallel and Distributed Computing, pages 126–148. Pitman Publishing, London, 1990.

[38] M. J. Compton. SCOOP: An Investigation of Concurrency in Eiffel. Bachelor thesis, Australian National University, Dept. of Computer Science, 2000.

[39] The Creol Language. `http://www.ifi.uio.no/~creol`.

[40] F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *FoSSaCS*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 1999.

[41] F. S. de Boer, M. B. Bonsangue, A. Grüner, and M. Steffen. Java Test Driver Generation from Object-Oriented Interaction Traces. *Electronic Notes in Theoretical Computer Science*, 243:33–47, 2009.

[42] F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In R. D. Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, Berlin, 2007.

[43] F. S. de Boer and I. Grabe. Automated Deadlock Detection in Synchronized Reentrant Multithreaded Call-Graphs. In J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, and B. Rumpe, editors, *SOFSEM 2010: Theory and Practice of Computer Science*, volume 5901 of *Lecture Notes in Computer Science*, pages 200–211. Springer-Verlag, Berlin, 2010.

[44] F. S. de Boer, I. Grabe, and M. Steffen. Termination detection for active objects. *Journal of Logic and Algebraic Programming*, 81(4):541 – 557, 2012.

[45] The E Language. `http://www.erights.org`.

[46] C. Engel and R. Hähnle. Generating Unit Tests from Formal Proofs. In Y. Gurevich and B. Meyer, editors, *TAP*, volume 4454 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2007.

[47] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *THEORETICAL COMPUTER SCIENCE*, 256(1), 1998.

[48] C. Flanagan and M. Felleisen. The Semantics of Future. Technical Report TR94-238, Rice University, Dept. of Computer Science, 1994.

[49] C. Flanagan and M. Felleisen. Well-Founded Touch Optimization of Futures. Technical Report TR94-239, Rice University, Dept. of Computer Science, 1994.

[50] C. Flanagan and M. Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.

[51] J. S. Foster, M. Fähndrich, and A. Aiken. A Theory of Type Qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, volume 34(5) of *ACM SIGPLAN Notices*, pages 192–203. ACM, New York, 1999.

[52] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1996)*, pages 372–385. ACM, New York, 1996.

[53] C. Fournet and G. Gonthier. The Join Calculus: A Language for Distributed Mobile Programming. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer-Verlag, Berlin, 2002.

[54] E. Giachino and C. Laneve. Analysis of deadlocks in object groups. In *FMOODS/FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2011.

[55] M. Giese. First-Order Logic. In Beckert et al. [19], pages 21–68.

[56] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[57] P. Haller and M. Odersky. Actors That Unify Threads and Events. In A. L. Murphy and J. Vitek, editors, *Coordination Models and Languages*, volume 4467 of *Lecture Notes in Computer Science*, pages 171–190. Springer-Verlag, Berlin, 2007.

[58] R. H. Halstead, Jr. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[59] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In OOPSLA1999 [85], pages 132–146.

[60] IO: A Small Progamming Language. `http://www.iolanguage.com`.

[61] G. S. Itzstein and D. Kearney. Join Java: An Alternative Concurrency Semantics for Java. Technical Report ACRC-01-001, University of South Australia, 2001.

[62] G. S. Itzstein and D. Kearney. Applications of Join Java. In *Proceedings of the 7th Asia-Pacific Conference on Computer Systems Architecture*, pages 37–46. Australian Computer Society, Inc., Darlinghurst, 2002.

[63] A. Jeffrey and J. Rathke. A Fully Abstract May Testing Semantics for Concurrent Objects. *Theoretical Computer Science*, 338(1-3):17–63, 2005.

[64] A. Jeffrey and J. Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. In M. Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 2005.

[65] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[66] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1-2):23–66, 2006.

[67] JSR 166: Concurrency Utilities. `http://www.jcp.org/en/jsr/detail?id=166`.

[68] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *In CAV*, pages 505–518. Springer, 2005.

[69] N. Kidd, A. Lal, and T. W. Reps. Language strength reduction. In Alpuente and Vidal [13], pages 283–298.

[70] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[71] L. Kornstaedt. Alice in the Land of Oz – an Interoperability-based Implementation of a Functional Language on Top of a Relational Language. *Electronic Notes in Theoretical Computer Science*, 59(1):20–35, 2001.

[72] P. Lammich and M. Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In Alpuente and Vidal [13], pages 205–220.

[73] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, B. Jacobs, G. T. Leavens, and C. Ruby. JML: notations and tools supporting detailed

design in Java. In *In OOPSLA 2000 Companion*, pages 105–106. ACM, 2000.

[74] T. Lev-Ami, N. Immerman, T. W. Reps, S. Sagiv, S. Srivastava, and G. Yorsh. Simulating Reachability Using First-Order Logic with Applications to Verification of Linked Data Structures. In R. Nieuwenhuis, editor, *CADE*, volume 3632 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2005.

[75] H. Liebermann. A Preview of Act 1. AI-Memo AIM-625, MIT, Artificial Intelligence Laboratory, 1981.

[76] H. Liebermann. Concurrent Object-Oriented Programming in Act 1. In Yonezawa and Tokoro [108], pages 9–36.

[77] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, volume 23(7) of *ACM SIGPLAN Notices*, pages 260–267. ACM, New York, 1988.

[78] D. A. Manolescu. Workflow Enactment with Continuation and Future Objects. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, volume 37(11) of *ACM SIGPLAN Notices*, pages 40–51. ACM, New York, 2002.

[79] B. Meyer. Systematic Concurrent Object-Oriented Programming. *Communications of the ACM*, 36(9):56–80, 1993.

[80] L. Moreau. The Semantics of Scheme with Future. In *Proceedings of the first ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, volume 31(6) of *ACM SIGPLAN Notices*, pages 146–156. ACM, New York, 1996.

[81] J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational Semantics for a Concurrent Lambda Calculus with Reference Cells and Futures. *Electronic Notes in Theoretical Computer Science*, 173:313–337, 2007.

[82] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda-calculus with futures. *Theoretical Computer Science*, 364(3):338–356, 2006.

[83] Object Modeling Group. *Object Constraint Language Specification, version 2.0*, June 2005.

[84] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala. A comprehensive step-by-step guide.* Artima Developer, 2008.

[85] *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, volume 34(10) of *ACM SIGPLAN Notices.* ACM, New York, 1999.

[86] A. Poetzsch-Heffter and J. Schäfer. A Representation-Independent Behavioral Semantics for Object-Oriented Components. In M. M. Bonsangue and E. B. Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *Lecture Notes in Computer Science*, pages 157–173. Springer-Verlag, Berlin, 2007.

[87] P. Pratikakis, J. Spacco, and M. Hicks. Transparent Proxies for Java Futures. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, volume 39(10) of *ACM SIGPLAN Notices*, pages 206–223. ACM, New York, 2004.

[88] R. R. Raje, J. I. William, and M. Boyles. An Asynchronous Method Incocation (ARMI) Mechanism for Java. *Concurrency, Practice and Experience*, 9(11):1207–1211, 1997.

[89] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22:2000, 2000.

[90] T. W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998.

[91] M. C. Rinard. Analysis of multithreaded programs. In P. Cousot, editor, *SAS*, volume 2126 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[92] P. Rümmer. Sequential, Parallel, and Quantified Updates of First-Order Structures. In M. Hermann and A. Voronkov, editors, *LPAR*, volume 4246 of *Lecture Notes in Computer Science*, pages 422–436. Springer, 2006.

[93] J. Schwinghammer. A Concurrent λ-Calculus with Promises and Futures. Diploma thesis, Programming Systems Lab, Universität des Saarlandes, 2002.

[94] M. Sirjani. Rebeca: Theory, applications, and tools. In *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, 2007.

[95] M. Steffen. *Object-Connectivity and Observability for Class-Based, Object-Oriented Languages*. Habilitation thesis, Christian-Albrechts-Universität zu Kiel, Technische Faktultät, 2006.

[96] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[97] T. Sysala and J. Janecek. Optimizing Remote Method Invocation in Java. In A. M. Tjoa and R. R. Wagner, editors, *Proceedings of the 13th International Workshop on Database and Expert Systems Applications (DEXA 2002)*, pages 29–36. IEEE Computer Society, Los Alamitos, 2002.

[98] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, volume 38(11) of *ACM SIGPLAN Notices*, pages 27–46. ACM, New York, 2003.

[99] K. Taura, S. Matsuoka, and A. Yoneazawa. ABCL/f: A Future-Based Polymorphic Typed Concurrent Object-Oriented Language — Its Design and Implementation. In G. E. Blelloch, K. M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms*, volume 18 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 275–292. American Mathematical Society, 1994.

[100] J. van den Berg and B. Jacobs. The LOOP Compiler for Java and JML. In T. Margaria and W. Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2001.

[101] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: A component-based language development environment. *Electronic Notes in Theoretical Computer Science*, 44(2):3 – 8, 2001. LDTA'01, First Workshop

on Language Descriptions, Tools and Applications (a Satellite Event of ETAPS 2001).

[102] P. L. Wadler. Linear types can change the world! In C. B. Jones and M. Broy, editors, *Programming Concepts and Methods: PROCOMET '90*, pages 347–359. North-Holland, 1990.

[103] A. Welc, S. Jagannathan, and A. Hosking. Safe Futures for Java. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, volume 40(10) of *ACM SIGPLAN Notices*, pages 439–453. ACM, New York, 2005.

[104] Y. Yokote and M. Tokoro. Concurrent Programming in Concurrent SmallTalk. In Yonezawa and Tokoro [108], pages 129–158.

[105] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* MIT Press, Cambridge, 1990.

[106] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-Oriented Concurrent Programming in ABCL/1. In *Proceedings of the 1st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'86)*, volume 21(10) of *ACM SIGPLAN Notices*, pages 258–268. ACM, New York, 1986.

[107] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. In Yonezawa and Tokoro [108], pages 55–89.

[108] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming.* MIT Press, Cambridge, 1987.