Static analysis of unbounded structures in object-oriented programs

Grabe, I.

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 $\triangleright f?$ is a blocking request of the result of the call with name $\triangleright f$. The result is consumed upon request, which entails that $\triangleright f?; \triangleright f?$ 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 $o_1\{run = o_1.m_1; ret, m_1 = \triangleright f = o_2.m_4; \triangleright f?; ret, m_2 = ret\}$.
Object $o_2\{run = o_2.m_3; ret, m_3 = \triangleright g = o_1.m_2; \triangleright g?; ret, m_4 = ret\}$.

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. $\triangleright f?$, is in the scope of a corresponding declaration, e.g. $\triangleright f = o.m$, 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 $o_1\{run = o_1.m_1; ret, m_1 = \triangleright f?; ret, m_2 = ret\}$
Object $o_2\{run = o_2.m_3; ret, m_3 = \triangleright g = o_1.m_2; \triangleright g?; ret, m_4 = ret\}$

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:

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

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:

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

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.

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

Here $\bot$ 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.

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

Here $\sigma[f := \kappa']$ denotes the result of assigning the label κ' to the future name $\triangleright f$ 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{aligned}
\Theta_I &= \{(o_1, (\sigma_\perp, \perp : o_1.m_1; ret), \emptyset), (o_2, (\sigma_\perp, \perp : o_2.m_3; ret), \emptyset)\} \\
&\to \{(o_1, (\sigma_\perp, \perp : ret), \{\perp : m_1\}), (o_2, (\sigma_\perp, \perp : o_2.m_3; ret), \emptyset)\} \\
&\to \{(o_1, (\sigma_\perp, \perp : ret), \{\perp : m_1\}), (o_2, (\sigma_\perp, \perp : ret), \{\perp : m_3\})\} \\
&\to \{(o_1, \epsilon, \{\perp : m_1\}), (o_2, (\sigma_\perp, \perp : 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?; ret), \emptyset), (o_2, \epsilon, \{\perp : m_3\}), \perp, \perp\} \\
&\to \{(o_1, (\sigma[f := \kappa], \perp : \rhd f?; ret), \emptyset), (o_2, \epsilon, \{\perp : m_3, \kappa : m_4\}), \perp, \perp\} \\
&\to \{(o_1, (\sigma[f := \kappa], \perp : \rhd f?; ret), \emptyset), (o_2, (\sigma_\perp, \kappa : ret), \{\perp : m_3\}), \perp, \perp\} \\
&\to \{(o_1, (\sigma[f := \kappa], \perp : \rhd f?; ret), \emptyset), (o_2, \epsilon, \{\perp : m_3\}), \kappa, \perp, \perp\} \\
&\to \{(o_1, (\sigma, \perp : 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?; ret), \emptyset), \perp, \perp, \perp\} \\
&\to \{(o_1, \epsilon, \{\kappa' : m_2\}), (o_2, (\sigma'[g := \kappa'], \perp : \rhd g?; ret), \emptyset), \perp, \perp, \perp\} \\
&\to \{(o_1, (\sigma_\perp, \kappa' : ret), \emptyset), (o_2, (\sigma'[g := \kappa'], \perp : \rhd g?; ret), \emptyset), \perp, \perp, \perp\} \\
&\to \{(o_1, \epsilon, \emptyset), (o_2, (\sigma'[g := \kappa'], \perp : \rhd g?; ret), \emptyset), \kappa', \perp, \perp, \perp\} \\
&\to \{(o_1, \epsilon, \emptyset), (o_2, (\sigma', \perp : ret), \emptyset), \perp, \perp, \perp\} \\
&\to \{(o_1, \epsilon, \emptyset), (o_2, \epsilon, \emptyset), \perp, \perp, \perp, \perp\}
\end{aligned}
$$

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.*

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

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.*

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

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

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

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 $\bot$ 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 $\bot$.

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 $\sigma_\bot$ 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 $\bot$ is added to the configuration which can not be consumed.

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

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

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

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

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

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

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

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.

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

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.*

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

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 $\Theta \to \Theta'$ iff $\alpha(\Theta) \to \alpha(\Theta')$. □

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:

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

Compared to **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.

$$(1)\quad \langle a\rangle \xrightarrow{\overline{a}} 0 \qquad (2)\quad \mathrm{in}(a).P \xrightarrow{a} P$$

$$(3)\quad \mathrm{rd}(a).P \xrightarrow{\underline{a}} P \qquad (4)\quad !\mathrm{in}(a).P \xrightarrow{a} P \mid !\mathrm{in}(a).P$$

$$(5)\quad \mathrm{inp}(a)?.P\_Q \xrightarrow{a} P \qquad (6)\quad \mathrm{rdp}(a)?.P\_Q \xrightarrow{\underline{a}} P$$

$$(7)\quad \mathrm{inp}(a)?.P\_Q \xrightarrow{\neg a} Q \qquad (8)\quad \mathrm{rdp}(a)?.P\_Q \xrightarrow{\neg a} Q$$

$$(9)\quad \frac{P \xrightarrow{\alpha} P' \quad \alpha \neq \neg a}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad (10)\quad \frac{P \xrightarrow{\neg a} P' \quad Q \not\xrightarrow{\overline{a}}}{P \mid Q \xrightarrow{\neg a} P' \mid Q}$$

$$(11)\quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad (12)\quad \frac{P \xrightarrow{\underline{a}} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q}$$

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.

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

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 ($\mathrm{rd}(a)$) 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.

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

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.

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

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:

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

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

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

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.

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

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*.*

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

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

Figure 5.4: Congruence rules

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

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 $\mathrm{rdp}(a)?.P\_Q$ 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 $\mathrm{rdp}(a)?.P\_Q$ 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.