Termination Detection for Active Objects[☆]

Frank S. de Boer

CWI, Amsterdam, The Netherlands

Immo Grabe

CWI, Amsterdam, The Netherlands
University of Leiden, The Netherlands

Martin Steffen

University of Oslo, Norway

Abstract

We investigate termination of suitable abstractions of systems of active objects modeled in Creol which focus on the network communciations. In particular we reduce the termination problem of an Actor–like subset of Creol, which restricts the synchronization patterns, to the termination problem of Linda. This reduction involves a termination preserving translation from Creol to Linda, i.e. a Creol program terminates iff the Linda model terminates. Furthermore the semantic consequences of different Creol communication primitivies are illustrated.

1. Introduction

Active objects form a well established model for distributed systems. We present a static technique for termination detection for active objects. Our technique is based on a translation into Linda and the representation of the Linda model als a P / T net [3].

We illustrate our technique in terms of the Creol modeling language. Creol [7, 6] is a modeling language for distributed concurrent systems based on asynchronous communications. In Creol a system consists of active objects communicating via asynchronous calls, futures, and promises [4, 1]. Creol objects encapsulated their data and can only be accessed by their interfaces. Furthermore the objects also encapsulate activity. In opposite to the synchronous case

[☆]Part of this work has been supported by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services* and the German-Norwegian DAAD-NWO exchange project *Avabi* (Automated validation for behavioral interfaces of asynchronous active objects).

where control, i.e. threads, passes object boundaries, each call spawns a new thread. Results are communciated in terms of futures.

This work shows the different approach that is need in the asynchronous setting compared to our previous work on termination detection for concurrent objects communicating synchronously [5]. Due to the severe differences between the asynchronous and the synchronous communication model a fundamently different approach is needed. In the synchronous setting [5] threads were the entities to be modeled whereas in the asynchronous setting the active objects are the entities to be modeled.

In Creol at most one process can be active within an object. Furthermore a blocking request for the result of a computation, i.e. a future, is available. Together these characteristics of Creol give rise to deadlock situations. The main result of this work is decideablility of termination for an Actor–like subset [2] of Creol which focuses on the network communications abstracting from data. This subset restricts the finegrained synchronization pattern within an object to the coarse grained run–to–completion pattern of Actor–like languages.

In this work we have identified Linda as natural model to describe the network communications of our Actor–like language by externalizing the input queues of the objects into the tuple space. We present a translation from our Actor–like language to Linda. For the Linda model we use the work of Busi et al. [3] to decide termination via a representation of the Linda model as a P / T net. Busi et al. [3] investigate the consequences of two different semantics for the message generation. In their work the destinction between the ordered and unordered semantics is crucial. In the ordered semantics a message is generated immediately due to this choice messages occure in the order in which they were send in the tuple space. In the unordered semantics only a sendbox for the message is added to the tuple space which has to be turned into the message 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 for the Actor–like subset of Creol because it does not include testing for a message or conditional branching. For this subset the ordered semantics coincides with the unordered one and both are not Turing powerful.

Finally we briefly discuss the semantic consequences of extending our Actor–like language to a more refined communication primitives and more fine–grained synchronization patterns. Of particular interest is the relation between these extensions and the basic distinction between ordered and unordered semantics.

Outline. This paper is organized as follows. We start with an introduction to Creol in section 2 followed by a presentation of the used Linda dialect in section 3. In section 4 we introduce the translation from Creol to Linda. We investigate the properties of our translation in section 5. We conclude in section 6 and give some insight on directions for future work.

2. Active Objects

A Creol model consists of a set of active objects communicating via asynchronous method calls. Each Creol object is a monitor and allows at most one process to be active within the object. Scheduling among the processes of an object is cooperative. Each object has an unbound process "queue". Processes run to completion, i.e. once scheduled a process keeps exclusive access to the object until termination. If an object is idle any process in its process queue can be scheduled for execution. The "active" behavior of an object is given in terms of a run–method which is the active process after object creation.

We ristrict ourselves to an Actor–like subset C_A of Creol to illustrate the translation from Creol to Linda. Focusing on the communication structure of the model, we abstract from data except for object identities. We assume all objects to be given in advance. Due to the abstraction from data branching (if–then–else) is turned into non–deterministic choice $(e_1 + e_2)$.

2.1. Syntax

We assume a given set of method names M with typical element m and a given set of object definitions O with typical element o. $run \in M$ is the designated *run*–method and defines the object's initial activity. An object is given in terms of its method definitions.

$$\begin{array}{lll} o & ::= & run = e; ret, m = e; ret, \ldots, m = e; ret \qquad \text{object} \\ e & ::= & \tau \mid o.m! \mid f = o.m! \mid f? \mid e; e \mid e + e \qquad \text{expression} \end{array}$$

Here τ denotes an internal (silent) step. $o.m!$ denotes an anonymous, asynchronous method call to method m on object o , i.e. the result of the call is not required by the caller. $f = o.m!$ denotes a future f bound to an asynchronous method call to method m on object o . We require the names of futures to be unique among all method definitions of object o . $f?$ denotes the (blocking) request of the result of a call stored in f . The result is consumed upon request, i.e. $f?; f?$ is a blocking sequence that leads to a deadlock. ret denotes the return-symbol indicating the writing of the result and the termination of the method. We do not have iteration but the anonymous call can be used to program recursion.

By D_o we note the set of method definitions $m = e$ given in o . By $D_o(m)$ we denote the definition given for method m in o , i.e. e for $m = e$. The set of object definitions O defines a program P.

We give the run–to–completion semantics of our language in the next section. Run–to-completion semantics means that a method has no means to release control exept for termination.

Well–formedness. A method is well–formed if each request of a future $(f?)$ appears in the scope of an according declaration $(f = o.m!)$ and if each future is only declared once. Well–typed Creol programs satisfy this requirement. Since futures are local to method invocations and can not 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.

Balancing. A method is balanced if for each future declaration there exists an according request of the future. A program is balanced if all its methods are balanced. The unbound production of runtime labels for futures is a major problem in the automated analysis of Creol program. This property is crucial for the abstraction from the runtime labels of the actual method invocations. Due to the balancing and the run–to–completion semantics there is a clear temporal separation between the results of calls done by different invocations of methods of an object. This allows for a precise semantics with respect to termination without unique runtime labels.

Note that focusing on the network communication we abstract from data in order to be able to decide certain properties like termination.

2.2. Operational Semantics

The operational semantics of a system of active objects is described by a labeled transition relation between configurations Θ which consist of objects and return values.

An object is denoted by a triple (o, a, Γ), where o is the object definition, a is the *active* process and Γ is the input/process queue. The active process is denoted by a labeled expression $\kappa : c@e$, where κ is a runtime label identifying a particular call c and e is an expression denoting the process to execute. A call is either a named call (o, o', m) or an anonymous call (o', m), where o denotes the caller, o' denotes the callee, and m denotes the method name. We call o the name of (o, a, Γ) .

The runtime label κ is generated upon method call and unambigiously identifies a particular call. Due to the run–to–completion semantics there are no partially evaluated processes in the process queue but only "fresh" method invocations. This allows us to represent a pending call by its runtime label $\kappa : c$ only and to look up the method code upon scheduling. Due to the abstraction from data we can also use $\kappa : c$ to represent the result of the call c.

We require a *valid* configuration Θ to contain exactly one element $(o, ., .)$ for each object $o \in O$. The label $\kappa : c@e$ indicates that e is the continuation of the execution of a method call c. The active process represents the method currently executed by the thread. The active process 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 process queue.

$$\theta_o = \{(o, \kappa : (\perp, run)@D_o(run), \emptyset)\}$$

Here $D_o(run)$ denotes the definition of the *run*–method given in o. The active process is given by the anonymous *run*–method is labelled with a fresh label κ. Being the initial activity the result of this process execution is never requested.

The initial configuration Θ_I of the program is the set of the initial configuration of the objects.

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

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

$$\Theta \cup \{(o, \bot, \Gamma \cup \{\kappa : c\})\} \to \Theta \cup \{(o, \kappa : c@D_o(c_m), \Gamma)\}$$

Here $\bot$ indicates the object being idle. $D_o(c_m)$ denotes the definition of the method with name c_m given in the call c.

Method termination. Upon method termination the executing object is set to idle and a future containing the result is created.

$$\Theta \cup \{(o, \kappa : c@ret, \Gamma)\} \to \Theta \cup \{(o, \bot, \Gamma)\} \cup \{\kappa : c\}$$

Here $\bot$ indicates the object being idle. Abstracting from data we only need to communicated the termination of the method and no concrete result value. We do this by adding the future $\kappa : c$ to the configuration.

Choice. Our Actor–like subset of Creol only contains non–deterministic choice.

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

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

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

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

Method call. An anonymous method call adds the call to the process queue of the callee and allows the caller to continue execution.

$$\begin{array}{rl} & \Theta \cup \{(o, \kappa : c@o'.m'!; e, \Gamma)\} \cup \{(o', a', \Gamma')\} \\ \to & \Theta \cup \{(o, \kappa : c@e, \Gamma)\} \cup \{(o', a', \Gamma' \cup \{\kappa' : (o', m')\})\} \end{array}$$

Here κ' is a fresh label identifying the anonymous call to method m' of object o' by o.

Future. A method call adds the call to the process queue of the callee and allows the caller to continue execution.

$$\begin{array}{rl} & \Theta \cup \{(o, \kappa : c@f = o'.m'!; e, \Gamma)\} \cup \{(o', a', \Gamma')\} \\ \to & \Theta \cup \{(o, \kappa : c@\alpha(e, f, \kappa'), \Gamma)\} \cup \{(o', a', \Gamma' \cup \{\kappa' : (o, o', m')\})\} \end{array}$$

Here $\alpha(e, f, \kappa')$ denotes the process e where each (syntactic) occurence of the future f is replaced by the runtime label κ'. This ensures an unambiguous matching between method calls and returns among different invocations of the same method.

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

$$\Theta \cup \{(o, \kappa : c@\kappa'?; e, \Gamma)\} \cup \{\kappa' : c'\} \rightarrow \Theta \cup \{(o, \kappa : c@e, \Gamma)\}$$

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 the object containing the process) is stuck.

3. Linda

We use a process algebra containing coordination primitivies of Linda following Busi et al.[3] to model the commmunication structure of a Creol program. Busi et al. give a process algebra $\mathbf{L}$ and two variations $\mathbf{L}_o$ and $\mathbf{L}_u$. The variations differ in the treatment of output operations.

In $\mathbf{L}_o$ outputs are instantaneous, i.e. on an output the corresponding message is added directly to the tuple space. In $\mathbf{L}_u$ outputs are buffered, i.e. instead of adding the message to the tuple space only a sendbox for the message is added to the tuple space which has to deliver the message by another (internal) step. The different treatment of outputs results in different expressiveness of $\mathbf{L}_o$ and $\mathbf{L}_u$. $\mathbf{L}_o$ is Turing powerful whereas $\mathbf{L}_u$ is not Turing powerful.

In case of $\mathbf{L}_o$ the instantaneous treatment of messages and the conditional choice operator $\mu?C\_C$ allow to model a Random Access Machine. In case of $\mathbf{L}_u$ a finite P/T system for any program in $\mathbf{L}_u$ can be given for which termination is decideable. This construction is based on a complex representation of read arcs and inhibtor arcs.

To model the communication structure of an Actor–like Creol programm not the full power of $\mathbf{L}$ is needed but only a sublanguage $\mathbf{L}_A$. For this sublanguage $\mathbf{L}_{A_o}$ and $\mathbf{L}_{A_u}$ coincide. Termination is decideable for $\mathbf{L}_A$, i.e. $\mathbf{L}_A$ is not Turing powerful.

3.1. Syntax

Let 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:

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

where:

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

Compared to $\mathbf{L}$ from Busi et al. 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 $\mathbf{L}_o$ and $\mathbf{L}_u$ 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.

3.2. Semantics

We follow [3] for the semantics of Linda. Also the rules for testing for messages and conditional branching are presented. In section 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 according Linda primitives at hand. Figure 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 already present the full set of rules as presented in [3]. Though we use rules (3) and (5)–(8) only later since we need neither testing nor conditional branching for the translation of Actor–like Creol.

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

Figure 1: Linda operational sematics (symmetric rules omitted)

Ordered Message Output. In figure 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.

Unordered Message Output. In figure 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

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

Figure 2: Message sending – ordered semantics

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 occure in the tuple space in an arbitrary order.

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

Figure 3: Message sending – unordered semantics

Example 1. Consider the following program $P = \mathrm{out}(a).\mathrm{inp}(a)?\langle b\rangle.0\_\langle c\rangle.0$. 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.

3.3. Expressivness

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

Lemma 1. For a Linda dialect without testing and conditional branching the ordered and the unordered semantics are both not Turing powerful.

The proof for the ordered semantics being Turing powerful in [3] 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 [3] both semantics can be shown to be not Turing powerful.

4. Modeling Creol Programs in Linda

We present the modeling of a Creol program in three steps. First we present the translation of a method in isolation which covers most of 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.

After that we present the modeling of a single object adding the production of return values and the scheduling of methods. Furthermore the modeling of the active behavior in terms of the *run*–method is given.

Finally the model of a Creol program is the parallel composition of the models of the models for the individual objects.

The crucial step in the modeling of the communication is the abstraction from the runtime labels. To use the results from [3] and to get decidability of termination we need to come up with a finite P / T net for the Linda model, i.e. we are restricted to a finite message alphabet. We give an abstraction that forfills this requirement. Instead of creating a unique runtime label for each method invocation we identify the method invocation only with the triple of caller, callee, and method name. For balanced, well-formed programs this identification is sufficient. Due to the balancing the lifespane of a future is restricted to one method invocation only which allows to preserve decidability of termination. for details we refer to section 5.

Pruning. Linda does not provide a primitive for internal steps. To facilitate the translation from Creol to Linda we "prune" the Creol program 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 program and removes as many internal steps as possible. In the end internal steps can only occure in a choice and even there at most in one branch.

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

Since the choice operator is non–deterministic the pruning of the Creol program 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 definitions of the Creol programs to be pruned.

4.1. Modeling 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 will be modeled in the following section 4.2. We introduce messages to deal with anonymous, asynchronous calls (o, m) denoting the callee o and the method name m. To deal with futures we introduce two messages (o, o', m) and $(\overline{o, o', m})$. The message (o, o', m) denotes a call by object o to method m of object o'. The message $(\overline{o, o', m})$ denotes the result of a call (o, o', m).

Modeling method calls and returns this way is ambiguous. Method calls issued by the same method invocation might be mixed up if they involve the same callee and method. Even worse calls and returns issued by different method invocations might be mixed up. The second problem is avoided by restricting to balanced programs only. The first problem is in fact no problem at all in our setting. Due to the abstraction from data two method calls to the same method of the same callee issued in the same method invocation can not be distinguished. Taking the asynchronous communication into account we can find a reordering of the messages such that the two invocations can be exchanged.

Non–deterministic choices are modeled by two processes competing for a designated message $(o, +)$. 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 occure 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.

$$\alpha(\tau) ::= 0$$

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.

$$\alpha(\mathit{ret}) ::= 0$$

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

$$\alpha(o'.m!) ::= \mathrm{out}((o', m))$$

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

$$\alpha(f = o'.m!) ::= \mathrm{out}((o, o', m))$$

Requesting result. The (blocking) request of an result to a method call from object o to method m' of object o' with label f is translated to the consumption of a message $(\overline{o, o', m})$.

$$\alpha(f?) ::= \text{in}((\overline{o, o', m}))$$

Sequential composition. A sequence of Creol statements is translated into a sequence of Linda statements. Please note that this can only occure 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.

$$\alpha(e_1; e_2) ::= \alpha(e_1).\alpha(e_2)$$

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.

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

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

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

4.2. Modeling a Single Object

The caller o' of a method m on object o denoted by a message (o', o, m) is the receiver of the result of the execution of m. To model this relation we model each caller–method–pair. The future to be produced is decided at the time of method reception. For each caller–method–pair 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 *run*–method creates the object token for the first time and adds it to the tuple space.

Caller–Method–Pair. We explicitly model the communciations with each possible caller o' to assign the return value to the caller.

$$\alpha(m) ::= \Pi_{o' \in O} \; !\mathrm{in}((o', o, m)).\alpha(o', e) | !\mathrm{in}((o, m)).\alpha(\bot, e)$$

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 α to reflect the two modes (named and anonymous) of asynchronous calls in our translation function.

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

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.

$$\alpha(o) ::= \Pi_{m \in o} \; \alpha(m)$$

$$\alpha(m) ::= \Pi_{o' \in O} \; !\mathrm{in}((o', o, m)).\mathrm{in}(o).\alpha(o', e) \mid !\mathrm{in}((o, m)).\mathrm{in}(o).\alpha(\bot, e)$$

$$\begin{array}{rcl}
\alpha(\bot, \mathit{ret}) & ::= & \mathrm{out}(o).0 \\
\alpha(o', \mathit{ret}) & ::= & \mathrm{out}((\overline{o', o, m})).\mathrm{out}(o).0
\end{array}$$

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.

$$\alpha(o) ::= \Pi_{m \in o}\alpha(m) \mid \alpha(\bot, D_o(\mathrm{run}))$$

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 terminatio.

4.3. Modeling a Creol program

A Creol program is modeled by a parallel composition of the objects of which the program consists of.

$$\alpha(P) ::= \Pi_{o \in O} \; \alpha(o)$$

5. Termination

In order to prove that our Creol program coincides with our Linda model with respect to termination we give as an intermediate step a new semantics for Creol programs which abstracts from the runtime labels. Then we show that the "concrete" and the "abstract" semantics for Creol programs coincide in case of well–formed, balanced programs. Finally we give a bisimulation between a Creol program in the abstract semantics and its counterpart in Linda.

As explained in section 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 comprehendable to the reader. Instead of storing unique runtime labels we only count the number of computed futures by means of tokens (caller, callee, method name) and the pending calls by means of a so–called decider set of syntax labels f. 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 lifespane of a future is restricted to one method invocation. This makes explicit runtime labels superfluous.

5.1. Abstract Creol Semantics

As an intermediate step we abstract from the runtime labels. Instead we introduce abstract call labels and decider sets. An abstract call label is a triple caller, callee, and method name for a future and a pair callee and method name for an anonymous call. A decider set is a set of (syntax) names of futures. The role of the abstract call labels is to count the number of available futures of a particular caller, callee, and method name combination. The role of the decider sets is to provide information to which calls these futures might belong. A label is added to the decider set upon method call and removed from the set upon the consumption of a result.

Initial configuration. The initial label of the *run*–method is $(\bot, run)$ where $\bot \notin O$. This ensures that the result of the initial *run*–method can not be requested by any object or process. In addition to the initial object we add empty decider sets for all possible callee and method combination with caller o. Please note this could be easily restricted to combination which actually occure in the method definitions of o. In the following χ is of the form (o, m) or (o', o, m) and χ_m denotes the method name m.

$$\theta_o = \{(o, (\bot, run)@D_o(run), \emptyset)\} \cup \bigcup_{o' \in O} \bigcup_{m \in D_{o'}} \{(o, o', m, \emptyset)\}$$

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

$$\Theta \cup \{(o, \bot, \Gamma \cup \{\chi\})\} \rightarrow \Theta \cup \{(o, \chi@D_o(\chi_m), \Gamma)\}$$

Method termination. Upon method termination an abstract call label is added to the configuration.

$$\Theta \cup \{(o, \chi@ret, \Gamma)\} \to \Theta \cup \{(o, \bot, \Gamma)\} \cup \{\chi\}$$

Method call. For an anonymous call only the abstract label is added to the queue.

$$\Theta\cup\{(o, \chi@o'.m'!; e, \Gamma)\}\cup\{(o', a', \Gamma')\} \to \Theta\cup\{(o, \chi@e, \Gamma)\}\cup\{(o', a', \Gamma'\cup\{(o', m')\})\}$$

Future. For a future the abstract label is added to the process queue and the decider set is extended by the future.

$$\begin{array}{rc} & \Theta \cup \{(o, \chi@f = o'.m'!; e, \Gamma)\} \cup \{(o', a', \Gamma')\} \cup \{(o, o', m', \Psi)\} \\ \to & \Theta \cup \{(o, \chi@e, \Gamma)\} \cup \{(o', a', \Gamma' \cup \{(o, o', m')\})\} \cup \{(o, o', m', \Psi \cup \{f\})\} \end{array}$$

Requesting result. Requesting a result to a method call consumes an abstract call label and the future name from the decider set. By passing the get statement ($f?$) we decide that one of the futures had the label f. By removing the future name from the decider set we keep track of this decision.

$$\begin{array}{rc} & \Theta \cup \{(o, \chi@f?; e, \Gamma)\} \cup \{(o, o', m', \Psi \cup \{f\})\} \cup \{(o, o', m')\} \\ \to & \Theta \cup \{(o, \chi@e, \Gamma)\} \cup \{(o, o', m', \Psi)\} \end{array}$$

Please note that f is per convention unique among the method definitions of o.

The decider sets are suited for balanced, well–formed programs. To cover unbalanced, well–formed programs we have to replace the sets with multisets. Please note that we loose pecision in the case of unbalanced, well–formed programs.

For the remainder of this section we assume programs to be balanced and well–formed.

Definition 1. For a configuration Θ the multiset $\vartheta(o, o', m)$ of ready futures for calls from o to method m of o' is defined as:

$$\vartheta(o, o', m) ::= \{(o, o', m) \mid (o, o', m) \in \Theta\}$$

Definition 2. For an object $(o', \chi@e, \Gamma)$ the multiset $\Upsilon_{o'}(o, m)$ of pending calls from o to method m of o' is defined as:

$$\Upsilon_{o'}(o, m) ::= \{(o, o', m) \mid (o, o', m) \in (\Gamma \ \cup \{\chi\})\}$$

Lemma 2. For a program $P$the following invariant holds:

$$\forall o, o', m : |\vartheta(o, o', m)| + |\Upsilon_{o'}(o, m)| = |\Psi_{o,m}|$$

where $(o, o', m, \Psi_{o,m}) \in \Theta$

Proof:

We prove lemmata 2 and 3 together.

Lemma 3. For a program P the following invariant holds:

$$(o, \perp, \Gamma) \in \Theta \Rightarrow \forall o', m : |\vartheta(o, o', m)| = |\Upsilon_{o'}(o, m)| = |\Psi_{o,m}| = 0$$

where $(o, o', m, \Psi_{o,m}) \in \Theta$

Proof:

First we prove that the lemmata hold for one method invocation given that lemma 3 holds initially. We prove the lemmata by induction on the length of the computation η. Furthermore we assume that the object o our method is supposed to run within initially is idle.

Induction start. $|\eta| = 1$ If the first step is made by o it is the scheduling of the method. Since we assume lemma 3 to hold in the initial state and the scheduling of a method neither changes the set of ready futures nor the set of pending calls nor the set of pending decisions we derive that $|\vartheta(o, o', m)| = |\Upsilon_{o'}(o, m)| = |\Psi| = 0$. If the first step is made by another object the mentioned sets stay unchanged since we assumed that initially 3 holds, i.e. there are no pending calls by o which could be processed by another object.

Induction step. $|\eta| = n + 1$ In case the last step was a *ret*–step we know that $|\vartheta(o, o', m)| + |\Upsilon_{o'}(o, m)| = |\Psi|$ holds for the computation up to the *ret*. Since the *ret*–step does not create any pending calls nor futures to calls by o (due to the run–to–completion semantics a named self–call would lead to a deadlock) nor does it change the decision set so lemma 2 holds in this case. Since the program is well–defined each future is declared and since the program is balanced each declared future is consumed. Due to lemma 2 and the properties of our program we know that all pending calls were scheduled and the futures were calculated and that all of these futures were consumed. Due to the validity of lemma 3 in the initial state we derive that lemma 3 holds in the final state again (which is an idle state).

In case the last step was an internal step, a choice, or an anonymous call the lemmata hold due to invariance since these local steps do not change any of the mentioned sets.

In case the last step was the creation of a future f, a new pending call is added to the queue of o'. Since m is well-formed this is the first time that f is declared. Since we assumed lemma 3 to hold initially we can derive that $f \notin \Psi$. So the validity of lemma 2 is preserved.

In case the last step was the request of a result both the number of abstract call labels and the set of decisions are decreased by one leaving the validity of 2 untouched.

In case the last step was a step by another object o' we have to consider the cases that o' schedules a call by o, that o' executes a call by o, and that o' finishes a call by o. In case o' schedules a call by o the call is moved from the input queue to the active process – which must have been idle before –, so the number of pending processes does not change. In case of an execution step for a method called by o the number of pending process stays unchanged. In case of o' finishing a call by o the number of pending processes is decreased by one but also a future is produced. The sum stays unchanged.

Generalization. Due to the object bound labeling of the pending calls, futures, and decision sets the only possibility of interference is the transformation of a pending call into a future. But as seen above our properties are immun to that. So we derive that since lemma 3 holds for the initial configuration Θ_I lemma 3 holds and since lemma 2 holds provided 3 holds we derive that 2 holds. □

Theorem 1. An execution according to the concrete semantics of a Creol program terminates iff an execution according to the abstract semantics terminates.

Proof:

We prove the relation by a bisimulation.

First we relate the configurations. A configuration Θ' of the abstracts simulates a configuration Θ of the concrete semantics $\Theta' \approx \Theta$ iff:

$$\forall o, o', m : |\{(o, o', m) \mid (o, o', m) \in \Theta'\}| = |\{\kappa \mid \kappa_s = o, \kappa_r = o', \kappa_m = m, \kappa \in \Theta\}|$$

$$\forall o : \exists \beta : \beta((o, \chi@e', \Gamma')) = (o, \kappa@e, \Gamma)$$

where $(o, \chi@e', \Gamma') \in \Theta'$, $(o, \kappa@e, \Gamma) \in \Theta$, and β a mapping from an abstract object state to a concrete one.

The mapping β maps each label in the decision set Ψ_o to a future κ. Furthermore β maps each call in Γ' (by another object $o' \neq o$) to a corresponding call (same caller, callee, and method) κ in Γ. This mapping is unique ensuring that Γ and Γ' contain the same amount of pending calls. The mapping is also applied to the active process. If such a mapping exists it means that the open decisions in Θ' depicted by the decision set can match the decisions made in Θ.

The simulation steps are straightforward. The lemmata 2 and 3 ensure that in the abstract semantics no call or future is “abstracted” away and that futures do not “survive” the calling process. These two properties allow for a sound mapping between the futures in the different kinds of configurations. □

Theorem 2. A Creol program terminates iff the corresponding Linda model terminates.

Proof:

We prove this property by a bisimulation between a Creol execution in the abstract semantics and an execution of the corresponding Linda model.

We relate a configuration Θ of the Creol program and a Linda model P by relating subconfigurations and individual processes.

Let $(o, \chi@e, \Gamma)$ be an object in Θ.

Method definitions. First of all the static part of o has to be matched, i.e. for each method definition m in o a process $M = \alpha(m)$ has to be found in P. These processes are of the form $!\mathrm{in}(m).P$ and model the code expansion of the method call.

Active Process. The active process has to be matched by a process $\alpha(o', ret)$ if χ is of the form (o', o, m) and $\alpha(\perp, ret)$ otherwise.

Idle Object. An idle object has to be matched by a process $\mathrm{out}(o).0$, by a message $\langle(o)\rangle$, or the seed for such a message $\langle\langle(o)\rangle\rangle$.

Pending Calls. Each pending anonymous call $(o, m) \in \Gamma$ with method definition $m ::= e$ has to be matched by pending process $\mathrm{in}(o).\alpha(\perp, e)$, a message $\langle(o, m)\rangle$, or the seed for such a message $\langle\langle(o, m)\rangle\rangle$. Each pending named call $(o', o, m) \in \Gamma$ has to be matched by pending process $\mathrm{in}(o).\alpha(o', e)$, by a message $\langle(o', o, m)\rangle$, or the seed for such a message $\langle\langle(o', o, m)\rangle\rangle$.

Return Values. Each return value (o', o, m) in Creol has to be matches by a message $\langle(\overline{o', o, m})\rangle$ or by the seed for a message $\langle\langle(\overline{o', o, m})\rangle\rangle$

The steps for the simulation are straightforward. □

6. Conclusion

6.1. Contribution

We have presented an Actor–like subset of Creol an its translation to Linda. For this translation only a subset of Linda for which the ordered and the unordered semantics coincide is needed. We have presented a semanctis of Actor–like Creol that abstracts from the runtime labels of the futures and provides us with an intermediate step between Creol and Linda. We used this intermediate step to proof the preservation of the termination property between the languages.

6.2. Future Work

We give some directions to advanced Creol concepts and on how they can be integrated to our translation. Furthermore we discuss the semantic consequences of this integration

Conditional Branching. Besides the blocking request Creol also has primitives to poll a future returning just the information whether or not a future has already been calculated. To model such a command we can use the $\mathrm{rdp}(a)?.P\_Q$ command 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.

Our labeling abstraction can still be used and the termination of the Creol program and the Linda model still coincide.

Conditional Scheduling. A condition on the existence of futures can also 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 otherwise the process goes to sleep (waiting for the remaining futures to be calculated). We can model the conditional scheduling on one future by the rdp$(a)?.P_Q$ command of Linda.

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 can not be avoided.

Deadlock Detection. At the moment termination (including normal termination) is decidable. With some modifications to the underlying P / T net we could sharpen the notion of termination to the notion of deadlocks.

Technical Improvement. There is a number of smaller technical translations and refinement of the translation to be done. Among them are the Creol interfaces and co–interfaces. Creol is typed by interfaces and objects can only be accessed by these interfaces. The co–interfaces restrict the set of possible callers of a method by forcing possible callers to implement the co–interfaces of the method. Switching to interfaces and co–interfaces it would suffice to model caller–method–pairs for valid combinations (with respect to the co–interfaces).

Object activation can be realized by activation messages similar to the scheduling tokens. In this case the activation token would block the process for the initial *run*–method until the object creator has send the activation token.

References

[1] 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 (28 pages), 2009. Special issue with selected contributions of NWPT'07. The paper is a reworked version of an earlier UiO Technical Report TR-364, Oct. 2007.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1990.

[3] N. Busi, R. Gorrieri, and G. Zavattaro. On the expressiveness of linda coordination primitives. *Inf. Comput.*, 156(1-2):90–121, 2000.

[4] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proceedings of Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Vienna, Austria.*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2007.

[5] 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, *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2010)*, volume 5901 of *Lecture Notes in Computer Science*, pages 200–211. Springer-Verlag, 2010.

[6] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[7] 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, Nov. 2006.