A Petri Net Based Analysis of Deadlocks for Active Objects and Futures*

Frank S. de Boer[1], Mario Bravetti[2], Immo Grabe[1], Matias Lee[3], Martin Steffen[4], and Gianluigi Zavattaro[3]

[1] CWI, Amsterdam, The Netherlands
[2] University of Bologna, Focus Team INRIA, Italy
[3] University of Córdoba, Argentina
[4] University of Oslo, Norway

Abstract. We give two different notions of deadlock for systems based on active objects and futures. One is based on blocked objects and conforms with the classical definition of deadlock by Coffman, Jr. et al. The other one is an extended notion of deadlock based on blocked processes which is more general than the classical one. We introduce a technique to prove deadlock freedom of systems of active objects. To check deadlock freedom an abstract version of the program is translated into Petri nets. Extended deadlocks, and then also classical deadlock, can be detected via checking reachability of a distinct marking. Absence of deadlocks in the Petri net constitutes deadlock freedom of the concrete system.

1 Introduction

The increasing importance of distributed systems demands flexible communication between distributed components. In programming languages like Erlang [3] and Scala [13] asynchronous method calls by active objects have successfully been introduced to better combine object-orientation with distributed programming, with a looser coupling between a caller and a callee than in the tightly synchronized (remote) method invocation model. In [5] so-called futures are used to manage return values from asynchronous calls. Futures can be accessed by means of either a *get* or a *claim* primitive: the first one blocks the object until the return value is available, while the second one is not blocking as the control is released. The combination of blocking and non-blocking mechanisms to access to futures may give rise to complex deadlock situations which require a rigorous formal analysis. In this paper we give two different notions of deadlock for systems based on active objects and futures. One is based on blocked objects and conforms with the classical definition of deadlock by Coffman, Jr. et al [8]. The other one is an extended notion of deadlock based on blocked processes which is

* Part of this work has been supported by the EU-project FP7-231620 HATS (Highly Adaptable and Trustworthy Software using Formal Methods), Eramus Mundus Action 2 Lot 13A EU Mobility Programme 2010-2401/001-001-EMA2 and EU 7FP grant agreement 295261 (MEALS).

more general than the classical one. We introduce a technique to prove deadlock freedom of models of active objects by a translation of an abstraction of the model into Petri nets. Extended deadlocks, and then also classical deadlock, can be detected via checking reachability of a distinct marking. Absence of deadlocks in the Petri net constitutes deadlock freedom of the concrete system.

The formally defined language that we consider is Creol [15] (Concurrent Reflective Object-oriented Language). It is an object oriented modeling language designed for specifying distributed systems. A Creol object provides a high-level abstraction of a dedicated processor and thus encapsulates an execution thread. Different objects communicate only by asynchronous method calls, i.e., similar to message passing in Actor models [12]; however in Creol, the caller can poll or wait for return values which are stored in future variables. An initial configuration is started by executing a *run* method (which is not associated to any class). The active objects in the systems communicate by means of method calls. When receiving a method call a new process is created to execute the method. Methods can have processor release points which define interleaving points explicitly. When a process is executing, it is not interrupted until it finishes or reaches a release point. Release points can be conditional: if the guard at a release point evaluates to true, the process keeps the control, otherwise, it releases the processor and becomes disabled as long as the guard is not true. Whenever the processor is free, an enabled process is *nondeterministically* selected for execution, i.e., scheduling is left unspecified in Creol in favor of more abstract modeling.

In order to define an appropriate notion of deadlock for Creol, we start by considering the most popular definition of deadlock that goes back to an example titled *deadly embrace* given by Dijkstra [7] and the formalization and generalization of this example given by Coffman Jr. et al.[8]. Their characterization describes a deadlock as a situation in a program execution where different processes block each other by denial of resources while at the same time requesting resources. Such a deadlock can not be resolved by the program itself and keeps the involved processes from making any progress.

A more general characterization by Holt [14] focuses on the processes and not on the resources. According to Hold a process is deadlocked if it is blocked forever. This characterization subsumes Coffman Jr.'s definition. A process waiting for a resource held by another process in the circle will be blocked forever. In addition to these deadlocks Holt's definition also covers deadlocks due to infinite waiting for messages that do not arrive or conditions, e.g. on the state of an object, that are never fulfilled.

We now explain our notions of deadlock by means of an example. Consider two objects o_1 and o_2 belonging to classes c_1 and c_2, respectively, with c_1 defining methods m_1 and m_3 and c_2 defining method m_2. Such methods, plus the method *run*, are defined as follows:

- $\mathit{run}() ::= o_1.m_1()$
- $m_1() ::= \mathsf{let}\, x_1 = o_2.m_2()\, \mathsf{in}\, \mathsf{get@}(x_1, \mathit{self}); \mathit{ret}$
- $m_2() ::= \mathsf{let}\, x_2 = o_1.m_3()\, \mathsf{in}\, \mathsf{get@}(x_2, \mathit{self}); \mathit{ret}$
- $m_3() ::= \mathit{ret}$

The variables x_1 and x_2 are futures, accessed (in this case) with the blocking *get* statement. This program clearly originates a deadlock because the execution of m_1 blocks the object o_1 and the execution of m_2 blocks the object o_2. In particular, the call to m_3 cannot proceed because the object o_1 is being blocked by m_1 waiting on its *get*. We call *classical* deadlocks these cases in which there are groups of objects such that each object in the group is blocked by a *get* on a future related to a call to another object in the group.

Consider now the case in which the method m_2 is defined as follows:

– $m_2() ::= \mathsf{let}\, x_2 = o_1.m_3()\,\mathsf{in}\,\mathsf{claim}@(x_2, self); ret$

In this case, object o_2 is not blocked because m_2 releases the control by performing a *claim* instead of a *get*. Nevertheless, the process executing m_2 will remain blocked forever. We call *extended* deadlock this case of deadlock at the level of processes.

After formalization of the notions of *classical* and *extended* deadlock, we prove that the latter includes the former. Moreover, as our main technical contribution, we show a way for proving extended deadlock freedom. The idea is to consider an abstract semantics of Creol expressed in terms of Petri nets. In order to reduce to Petri nets, we abstract away several details of Creol, in particular, we represent futures as quadruples composed of the invoking object, the invoking method, the invoked object, and the invoked method. For instance, the above future x_1 is abstractly represented by $o_1.m_1@o_2.m_2$.

Due to this abstraction, in the abstract semantics a process could access a wrong future simply because it has the same abstract name. Consider, for instance, the following example:

```
– $run() ::= o_1.m_1()$
– $m_1() ::= \mathsf{let}\, x_1 = o_2.m_2(1)\,\mathsf{in}$
                $\mathsf{let}\, x_2 = o_2.m_2(2)\,\mathsf{in}$
                  $\mathsf{get}@(x_2, self); \mathsf{claim}@(x_1, self); ret$
– $m_2(x_1) ::= \mathsf{if}\, x_1 = 1\,\mathsf{then}\, ret\,\mathsf{else}\,\mathsf{let}\, x_2 = o_1.m_3()\,\mathsf{in}\,\mathsf{claim}@(x_2, self); ret$
– $m_3() ::= ret$
```

Both the futures x_1 and x_2 will be represented by the same abstract name $o_1.m_1@o_2.m_2$. For this reason, even if this program originates a deadlock when *get* is performed on x_2, according to the abstract semantics the system could not deadlock. In fact, the return value of the first call could unblock the *get* as the two futures have the same name in the abstract semantics. To overcome this limitation, we add in the abstract semantics marked versions of the methods: when a method m is invoked, the abstract semantics nondeterministically selects either the standard version of m or its marked version denoted with $m?$. Both method versions have the same behavior, but the return value will be stored in two futures with two distinct abstract names. For instance, in the example above, if we consider that the first call to m_2 actually activates the standard version m_2 while the second one activates the marked version $m_2?$, there will be no confusion between the two futures as their abstract names will be $o_1.m_1@o_2.m_2$

and $o_1.m_1@o_2.m_2?$, respectively. In this case, the system will deadlock also under the abstract semantics.

To apply this technique internal choice is an obstacle. We explain this with more details in Section 4. To overcome this problem we move all internal choices up front. During the transformation we make a data abstraction, remove superfluous internal steps and duplicated choices from the program to reduce the size of the Petri net. This transformation can add spurious deadlock but it cannot remove them because the new abstract model is an overapproximation of the original system.

The Petri net based abstract semantics allow us to obtain a decidable way for proving extended deadlock freedom. In fact, reachability problems are decidable in Petri nets, and we show how to reduce extended deadlock to a reachability problem in the abstract Petri net semantics.

Outline. In Section 2 we report the definition of Creol. We present the two notions of deadlock in Section 3. In Section 4 we present the translation into Petri nets. In Section 5 we present the main result of the paper: if in the Petri net associated to a program a particular marking cannot be reached then the program is deadlock free, and we show that such reachability problem is decidable for Petri nets. Section 6 concludes the paper.

Extended version of the paper. Due the lack of space, neither the translation to Petri net nor the complete proof of our main result, Theorem 1, is presented in detail in this paper. These ones can be found in [6]

2 A Calculus for Active Objects

In this section we present a calculus with active objects communicating via *futures*, based on *Creol*. The calculus is a slight simplification of the object calculus as given in e.g. [2], and can be seen as an active-object variant of the concurrent object calculus from [11]. Specific to the variant of the language here and the problem of deadlock detection are the following key ingredients of the communication model:

Futures. Futures are a well-known mechanism to hold a "forthcoming" result, calculated in a separate thread. In Creol, the communication model is based on futures for the results of method calls which results in a communication model based of asynchronously communicating active object. In this paper we do not allow references to futures to be passed around, i.e. the futures in this paper are not first-class constructs. This restriction is enforced (easily) by the type system.

Obtaining the Results and Cooperative Scheduling. Method calls are done asynchronously and the caller obtains the result back when needed, querying the future reference. The model here support two variants of that querying operation: the non-blocking claim-statement, which allows reschedule of the querying code in case the result of not yet there, and the blocking

Table 1. Abstract syntax

$C ::= \mathbf{0} \mid C \parallel C \mid n[\![O]\!] \mid n[n, F, L] \mid \underline{n\langle t\rangle}$	component
$O ::= F, M$	object
$M ::= l = m, \ldots, l = m$	method suite
$F ::= l = f, \ldots, l = f$	fields
$m ::= \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$	method
$f ::= v$	field
$t ::= v \mid \mathsf{stop} \mid \mathsf{let}\, x{:}T = e \,\mathsf{in}\, t$	thread
$e ::= t \mid \mathsf{if}\, e \,\mathsf{then}\, e \,\mathsf{else}\, e \mid n.l(\vec{v}) \mid v.l \mid v.l := v$	expr.
$\mid\ \mathsf{claim@}(n, n) \mid \mathsf{get@}(n, n) \mid \underline{\mathsf{get@}n}$	
$\mid\ \mathsf{suspend}(n) \mid \underline{\mathsf{grab}(n)} \mid \underline{\mathsf{release}(n)}$	
$v ::= x \mid n$	values
$L ::= \bot \mid \top$	lock status

get-statement, which insist on getting the result without a re-scheduling point. In [2], we did not consider the latter as part of the user syntax.

Statically Fixed Number of Objects. In this paper we omit object creation to facilitate the translation to Petri nets.

The type system and properties of the calculus, e.g. subject reduction and absence of (certain) run-time errors, presented in [2] still apply. For brevity we only present explanation for language constructs relevant to the development of deadlocks. Missing details with respect to other language constructs, formalizations and proofs of the mentioned (and further) properties of the calculus can be found in [2].

2.1 Syntax

The abstract syntax is given in Table 1, distinguishing between *user* syntax and *run-time* syntax, the latter underlined. The user syntax contains the phrases in which programs are written; the run-time syntax contains syntactic constituents additionally needed to express the behavior of the executing program in the operational semantics.

The basic syntactic category of names n, represents references to classes, to objects, and to futures/thread identifiers. To facilitate reading, we write o and its syntactic variants for names referring to objects, c for classes, and n for threads/futures, resp. when being unspecific. Technically, the disambiguation between the different roles of the names is done by the type system. x stands for variables, i.e., local variables and formal parameters, but not instance variables. Besides names and variables x, we assume standard data types (such as booleans, integers, etc) and their values without showing them in the syntax of the core calculus. They are unproblematic for the deadlock analysis, which, using data abstraction, concentrates on the analysis of the communication behavior.

A *configuration* C is a collection of classes, objects, and (named) threads, with $\mathsf{0}$ representing the empty configuration. The sub-entities of a configuration are composed using the parallel-construct $\parallel$ (which is commutative and associative, as usual). The entities executing in parallel are the named threads $n\langle t\rangle$, where t is the code being executed and n the name of the thread. Threads are identified with futures, and their name is the reference under which the future result value of t will be available. A class $c[\![O]\!]$ carries a name c and defines its methods and fields in O. An object $o[c, F, L]$ with identity o keeps a reference to the class c it instantiates, stores the current value F of its fields, and maintains a *binary lock* L. The symbols $\top$, resp., $\bot$, indicate that the lock is taken, resp., free. The *initial* configuration consists of a number of classes, one initial thread, and a number of objects (with their locks free); under our restriction that we do not allow object instantiation, and we assume that their identities are known to the initial thread. By convention, the initial thread is assumed to be the body of a (unique) method named *run*.

Besides configurations, the grammar specifies the lower level syntactic constructs, in particular, methods, expressions, and (unnamed) threads, which are basically sequences of expressions, written using the let-construct. The stop-construct denotes termination, so the evaluation of a thread terminates by evaluating to a value or terminating with stop. A method $\varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t$ provides the method body t abstracted over the ς-bound "self" parameter s the formal parameters $\vec{x}$ —the ς-binder is borrowed from the well-known object-calculus of Abadi and Cardelli [1]. Note that the methods are stored in the classes but the fields are kept in the objects.

Methods are called asynchronously, i.e., executing $o.l(\vec{v})$ creates a new thread to execute the method body with the formal parameters appropriately replaced by the actual ones; the corresponding thread identity at the same time plays the role of a future reference, used by the caller to obtain, upon need, the eventual result of the method. The further expressions claim, get, suspend, grab, and release deal with communication and synchronization. As mentioned, objects come equipped with binary locks which assures mutual exclusion. The operations for lock acquisition and release (grab and release) are run-time syntax and inserted before and at the end of each method body code when invoking a method. Besides that, lock-handling is involved also when futures are claimed, using claim or get. The $\mathsf{get}@(n, o)$ operation is easier: it blocks object o (it executes in) if the result of future n is not (yet) available, i.e., if the thread n is not of the form of $n\langle v\rangle$. The $\mathsf{claim}@(n, o)$, is a more "cooperative" version of get: if the value is not yet available, it releases the lock of the object o (it executes in) to try again later, meanwhile giving other threads the chance to execute in that object. By convention, user-syntax commands only refer to the self-parameter *self*, (i.e., the ς-bound variable) in their object-argument, i.e., they are written $\mathsf{claim}@(n, \mathit{self})$, $\mathsf{get}@(n, \mathit{self})$, and $\mathsf{suspend}(\mathit{self})$. We also include a variant $\mathsf{get}@n$ of the get-operation as part of the run-time syntax, for consumption of the return value also when a lock is not held (it is needed to define the semantics of claim). As usual we use sequential composition $t_1; t_2$ as syntactic sugar for $\mathsf{let}\, x{:}T = t_1 \,\mathsf{in}\, t_2$, when x does not occur free in t_2. We refer to [2] for further details on the language constructs, a type system for the language and a comparison with the multi-threading model of *Java*.

2.2 Operational Semantics

Relevant reduction steps of the operational semantics are shown in Table 2, distinguishing between confluent steps $\rightsquigarrow$ and other transitions $\xrightarrow{\tau}$. The $\rightsquigarrow$-steps, on the one hand, do not access the instance state of the objects. The $\xrightarrow{\tau}$-steps, on the other hand, access the instance state, either by reading or by writing it, and may thus lead to race conditions. When not differentiating between the two kinds of transitions, then we replace both symbol by $\rightarrow$. An execution is a sequence of configurations, $C_0, \ldots, C_n$ such that C_{i+1} is obtained from C_i by applying a reduction step. We denote this execution by $C_0 \rightarrow \ldots \rightarrow C_n$.

We omit reduction rules dealing with the basic constructs like substitution, sequential composition (let), conditionals, field access, and lock handling. These rules are straightforward (cf. [2]). For deadlock detection later, most of these constructs will be subject to data abstraction.

Table 2. Operational semantics

$$\begin{array}{l}
c[\![F', M]\!] \parallel o[c, F, L] \parallel n_1\langle \mathsf{let}\, x{:}T = o.l(\vec{v}) \,\mathsf{in}\, t_1\rangle \xrightarrow{\tau} \\
c[\![F', M]\!] \parallel o[c, F, L] \parallel n_1\langle \mathsf{let}\, x{:}T = n_2 \,\mathsf{in}\, t_1\rangle \qquad \textsc{Fut}_i \\
\quad \parallel n_2\langle \mathsf{let}\, x{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v}) \,\mathsf{in}\, \mathsf{release}(o); x\rangle
\end{array}$$

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

$$\frac{t_2 \neq v}{\begin{array}{l} n_2\langle t_2\rangle \parallel n_1\langle \mathsf{let}\, x : T = \mathsf{claim@}(n_2, o) \,\mathsf{in}\, t_1'\rangle \rightsquigarrow \\ \quad n_2\langle t_2\rangle \parallel n_1\langle \mathsf{let}\, x : T = \mathsf{release}(o); \mathsf{get@}n_2 \,\mathsf{in}\, \mathsf{grab}(o); t_1'\rangle \end{array}} \ \textsc{Claim}_i^2$$

$$n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\, x : T = \mathsf{get@}(n_1, o) \,\mathsf{in}\, t\rangle \rightsquigarrow n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\, x : T = v \,\mathsf{in}\, t\rangle \qquad \textsc{Get}_i^1$$

$$n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\, x : T = \mathsf{get@}n_1 \,\mathsf{in}\, t\rangle \rightsquigarrow n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\, x : T = v \,\mathsf{in}\, t\rangle \qquad \textsc{Get}_i^2$$

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

Invoking a method (cf. rule $\textsc{Fut}_i$) creates a new future reference and a corresponding thread is added to the configuration. In the configuration after the reduction step, the meta-mathematical notation $M.l(o)(\vec{v})$ stands for $t[o/s][\vec{v}/\vec{x}]$, when the method suite $[M]$ equals $[\ldots, l = \varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t, \ldots]$. Upon termination, the result is available via the claim- and the get-syntax (cf. the Claim- and Get-rules), but not before the lock of the object is given back again using $\mathsf{release}(o)$. If the thread is not yet terminated, in the case of claim statement, the requesting thread suspends itself, thereby giving up the lock. The rule Suspend releases the lock to allow for interleaving. To continue, the thread has to reacquire the lock.

The above reduction relations are used modulo structural congruence, which captures the algebraic properties of especially parallel composition.

3 Deadlock

We give two different notions of deadlock in Creol. The first one follows [8]. In this case not only processes are blocked but also the objects hosting them.

The second notion resembles the definition of deadlock by Holt [14]. Instead of looking at blocked objects we look at blocked processes. A blocked process does not necessarily block the object hosting it.

To facilitate the definition of deadlock we introduce two notions of the location and state of a process. The notion of a *waiting* process links a process to another process or to an object. In the first case, it is waiting to read a future that the other process has to calculate. In the second case, the process is waiting to obtain the lock of the object.

Definition 1 (Waiting Process). *A process $n_1\langle t\rangle$ is waiting for:*

1. *n_2 iff $\langle t\rangle$ is of the form $\langle \mathsf{let}\, x{:}T = \mathsf{claim@}(n_2, o)\,\mathsf{in}\, t'\rangle$, $\langle \mathsf{let}\, x{:}T = \mathsf{get@}(n_2, o)\,\mathsf{in}\, t'\rangle$, or $\langle \mathsf{let}\, x{:}T = \mathsf{get@}n_2\,\mathsf{in}\, t'\rangle$;*
2. *o iff $\langle t\rangle$ is of the form $\langle \mathsf{let}\, x{:}T = \mathsf{grab}(o)\,\mathsf{in}\, t'\rangle$*

The notion of a *blocking* process links a process that is waiting for a future while holding the lock of the object.

Definition 2 (Blocking Process). *A process $n_1\langle t\rangle$ blocks object o iff $\langle t\rangle$ is of the form $\langle \mathsf{let}\, x : T = \mathsf{get@}(n_2, o)\,\mathsf{in}\, t'\rangle$.*

Note that a process needs to hold the object lock and execute a blocking statement, i.e. get-statement, to block an object. Furthermore note that the process can at most acquire one lock, i.e. the lock of its hosting object.

Our notion of a classical deadlock follows the definition of deadlock by Coffman Jr. et al.[8]. The resource of interest is the exclusive access to an object represented by the object lock. In opposite to the multithreaded setting, e.g. like in *Java*, where a thread can collect a number of these exclusive right, a process in the active object setting can at most acquire the lock of the object hosting it. But by calling a method on another object and requesting the result of that call it requests access to that object indirectly. Or to be more precise a process can derive the information, that the process created to handle its call and access to the callee, by the availability of the result in terms of the future.

Definition 3 (Classical Deadlock). *A configuration Θ is deadlocked iff there exists a set of objects O such that, for all $o \in O$, o is blocked by a process n_1 which is waiting for a process n_2 which is waiting for $o' \in O$.*

Note that the definition of "waiting for" plays a crucial role here, because the process is waiting, the process does not finish its computation. Being blocked by a process, another process can only gain access to the object after the blocking process has made progress. Since each process blocking an object in O is waiting for another process blocking an object in O we have a classical deadlock situation. Note that a blocking process does not necessarily directly wait for another blocking process but can also wait for a process which is waiting to get access to an object in O. But this process can only proceed if the process blocking the object proceeds.

The second notion resembles the definition of deadlock by Holt [14]. Instead of looking at blocked objects we look at blocked processes. A process can be blocked due to the execution of either a get–statement or a claim–statement. In the first case the object is blocked via the active process, in the second case only the process is blocked. Processes that are blocked on a claim–statement are not part of a deadlock according to the first definition since they are not holding any resources. Yet they can be part of a circular dependency that prevents them from making any progress.

Definition 4 (Extended Deadlock). *A configuration Θ is* deadlocked *iff there exists a finite set of processes N such that, for all $n_1 \in N$, n_1 is waiting for $n_2 \in N$, or waiting for o which is blocked by $n_2 \in N$.*

We require the set of processes to be finite in order to guarantee circularity. This notion of deadlock is more general than the classical one.

Corollary 1. *Every classical deadlock is an extended deadlock.*

4 Translation into Petri Nets

We translate Creol programs into Petri nets in such a way that extended deadlocks in a Creol program can be detected by analyzing the reachability of a given class of markings (that we will call *extended deadlock markings*) in the corresponding Petri net.

We first recall the definition of Petri nets. A Petri net is a tuple $\langle P, T, \vec{m_0} \rangle$ such that P is a finite set of places, T is a finite set of transitions, and $\vec{m_0}$ is a marking, i.e. a mapping from P to $\mathbb{N}$ that defines the initial number of tokens in each place of the net. A transition $t \in T$ is defined by a mapping ${}^\bullet t$ (preset) from P to $\mathbb{N}$, and a mapping $t^\bullet$ (postset). A configuration is a marking $\vec{m}$. Transition t is enabled at marking $\vec{m}$ iff ${}^\bullet t(p) \leq \vec{m}(p)$ for each $p \in P$. Firing t at $\vec{m}$ leads to a new marking $\vec{m'}$ defined as $\vec{m'}(p) = \vec{m}(p) - {}^\bullet t(p) + t^\bullet(p)$, for every $p \in P$. A marking $\vec{m}$ is reachable from $\vec{m_0}$ if it is possible to produce it after firing finitely many times transitions in T.

During this translation we apply abstraction with respect to the futures. In Creol a fresh unique label is created for each method invocation; instead, we use abstract labels for the futures only identifying a tuple of caller, calling method, callee, and called method. The reason for this abstraction is to get a Petri nets with finite places. Yet we still allow for an unbounded number of method invocations, i.e. an unbounded number of processes.

In the Petri net, we will have two kinds of places: those representing a method code to be executed by a given object, and those representing object locks. In order to keep the Petri net finite, we assume that only boundedly many objects will be present in a Creol configuration (otherwise we will have to consider unboundedly many places for the object locks). Moreover, in the places representing the method code to be executed, we abstract away from the data that could influence such method (like, e.g., the object fields) otherwise we would need infinitely many places.

Due to the abstraction with respect to the labels of futures, the abstract Petri net semantics could have the following *token confusion* problem. Namely, if there are two concurrent invocations between the same two methods of the same two objects, in the Petri net it could happen that one caller could read the reply generated by the method actually called by the other one. To avoid at least the propagation of the *token confusion* problem, in the Petri net, as soon a caller accesses to a return value in a future, such value is consumed. In this way, we assign the future to a concrete caller and consuming the future prevents it from being claimed by two different processes. To apply this technique in a sound way we have to transform the program. Removing the future upon first claim implies that it is not available for consecutive claims (in opposite to the concrete case). On the other hand consecutive claims do not provide any new information with respect to deadlock detection. Once a future has been claimed in the concrete case all consecutive claims pass. We model this by removing consecutive claims from the program.

But this approach only allows to avoid the token confusion for sequential identical abstract processes. In the case of concurrent identical abstract processes this is not enough. To address this problem each future creation can be marked or not. The Petri net will be defined in such a way that token confusion will not occur between a marked and non-marked call. The deadlock analysis will be done only over the marked processes: if only the method calls directly involved in the deadlock are marked, then there will be no token confusion between the method executions which are involved in the deadlock and those which are not.

Internal choice is an obstacle with respect to this approach. In a sequence of internal choices the kind of a claim (first or consecutive) depends on the choices taken so far and can vary depending on them. To overcome this problem we move all internal choices up front.

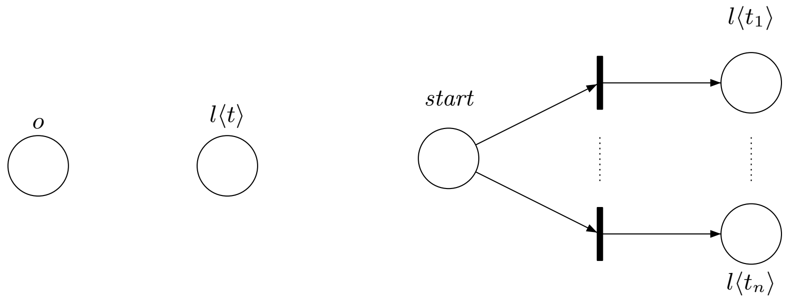
During the transformations we remove superfluous internal steps and duplicated choices from the program to reduce the size of the Petri net. For the technical details of the transformations we refer the reader to [6]. We now describe the Petri net construction more in details.

4.1 Places and Tokens

The resulting Petri net contains two kinds of places:

Locks. Places identifying the locks of the objects. Each object has its designated lock place labeled by the unique name of the object. A token in such a place represents the lock of the corresponding object being available. There is at most one token in such a place.

Process. Places identifying a particular process in execution or the future as a result of the execution of a process. These places are labeled with $l\langle t\rangle$ where l is an abstract label identifying the call and t is abstract method code to be "executed". A token in this place represents one instance of such a process in execution or a future. In case of a future, the token is consumed if the future is claimed.



(a) Places for objects and abstract processes

(b) Transition from the initial place *start*: t_i's are the abstract traces of the initial process *run*

Fig. 1. Places and Initial transitions

4.2 Code Abstractions

In [6] the code abstraction is defined in detail, here we give a quick description. The syntactical transformation is composed of five functions:

Step One s_1**.** It applies data abstraction.

Step Two s_2**.** It removes choices. If t is the code of a method, $s_2(s_1(t))$ is a set of sequential code without branching. We will call these also "traces", as they represent possible (abstract) executions of the method.

Step Three s_3^F**.** It removes the redundant claims of a future, i.e. the claims that are after the first one with respect to particular future. Notice that this function is applied over traces, then, it can be checked when a future is claimed. It also replaces claim–statement by a sequence of release, get and grab statements. We justify this decision below, when we define the transition associated to the claim–statement. The function also replaces suspend–statement by a release and a grab. F is a set used to keep track of the already claimed futures.

Step Four s_4**.** The function takes a trace and returns a set of traces. The set is constructed such that each trace in the set is the received trace with at most one of the future claims marked. In this way, the function takes into account every claim for deadlock analysis (if a deadlock exists, the corresponding trace will be included among the possible non-deterministic ones considered). Notice that a trace without mark, i.e. the received trace, is also included.

Step Five s_5. It applies the abstraction on the futures replacing them with the tuple calling object, calling method, called object, and called method.

Functions s_3^F, s_4 and s_5 are lifted to support sets of traces. Then the code transformation is defined as the composition of all the functions $ST ::= s_5 \circ s_4 \circ s_3^{\emptyset} \circ s_2 \circ s_1$ and it is applied to the method definitions. Suppose m is the method code in a class definition, i.e. in the configuration there is a method suite $[M]$ equals to $[\ldots, l = \varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).m, \ldots]$. Then, $ST(m)$ is a set of traces where each trace represents a possible abstract execution of the method l. A trace in $ST(m)$ is a sequence of abstract statements of the following form: $\mathsf{let}\, x{:}T = o.l$, $\mathsf{claim}@(n, n')$, $\mathsf{get}@(n, n')$, $\mathsf{get}@n$, $\mathsf{suspend}(n)$, $\mathsf{release}(n)$, $\mathsf{grab}(n)$, stop, $\mathsf{claim}@(n?, n')$, $\mathsf{get}@(n?, n')$ and $\mathsf{get}@n?$. Notice that the last three statements include marked calls. Each trace will have at most one marked claim.

4.3 Transitions

The transitions of the Petri net are determined by the translation of the semantic steps. For each object and each method a path for all pairs of caller and calling method is created. We give the translation for the individual execution steps according to the operational semantics in Section 2.2. In case the syntactical transformation affects the operational step we briefly discuss the consequences of the transformation.

Initial Transitions. A Creol program is defined by an initial configuration C_0 composed of a set of classes, a set of objects and an initial thread. We denote the initial thread *run*. This one is the main process in the program, then it is not called by another thread, does not belong to any object, nor class. The code associated to this thread has to be also translated using ST. The election of the trace for the main process is done by the initial transition depicted in Fig. 1(b), to do this we have included an auxiliary place *start*. This place will be the initial place of the Petri net.

Method Calls. We present the Petri net transitions for a method call in Fig. 2. A process place in the Petri net is labeled with a tuple $o_1.l_1@o_2.l_2$ where o_1 denotes the caller, l_1 the calling method, o_2 the callee, and l_2 the called method. We abbreviate parts of the label by $c@o.l$ resp. $o.l@c$ or the whole label by l if details are not needed. Depending on whether the result of the call will be assumed to be part of a deadlock the created process is marked (see Fig. 2(a), notice the symbol "?") or is not (see Fig. 2(b)). The method body of the called process t' is in both cases of the form $\mathsf{grab}(o); t_{o.l}; \mathsf{release}(o)$ where $t_{o.l}$ is an abstract trace execution of the method l of the object o according to the definitions in the associated class. At this point, the abstract execution unifies all the internal choices into one general internal choice that is resolved when the method is called.

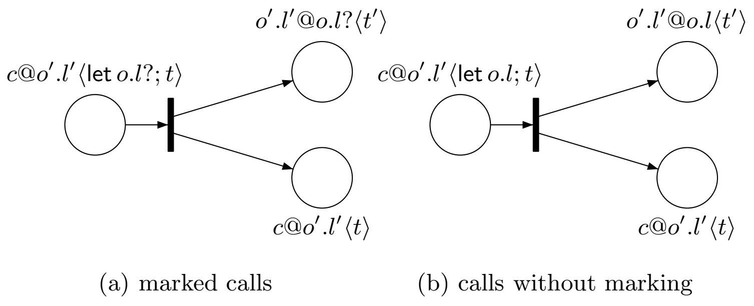

$c@o'.l'\langle \mathsf{let}\, o.l?; t\rangle$
$o'.l'@o.l?\langle t'\rangle$
$c@o'.l'\langle t\rangle$
(a) marked calls
$c@o'.l'\langle \mathsf{let}\, o.l; t\rangle$
$o'.l'@o.l\langle t'\rangle$
$c@o'.l'\langle t\rangle$
(b) calls without marking

Fig. 2. Transitions for method calls

Lock Handling. To execute the $\mathsf{grab}(o)$ statement the object lock of object o must be available. When releasing the lock of an object o by $\mathsf{release}(o)$ a token is added to the place representing the object lock. (Fig. 3)

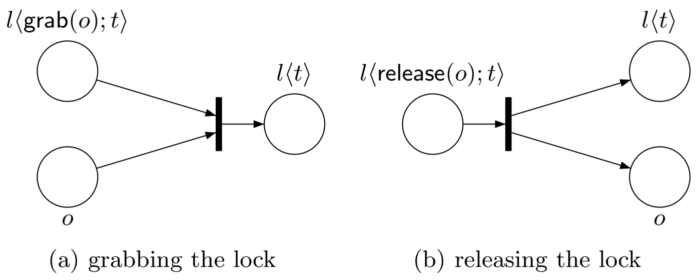


Fig. 3. Transitions for lock handling

Claiming Results. We present the Petri net transitions for claiming the result of a method call in Fig. 4. The notations "$o.l^+$" or "$o.l^*$" denote that $o.l$ can be marked or not: formally, $+$ and $*$ are meta-variables that can be either the empty string or ?. As was explained before, to avoid the token confusion of sequential calls, the tokens are consumed. Notice that removing the result is not problematic with respect to multiple claims of a value because subsequent claims are removed in the syntactical transformation.

Rescheduling. In *Creol* semantics there are two different kinds of rescheduling. Unconditional rescheduling, using keyword $\mathsf{suspend}(o)$, which is translated to $\mathsf{release}(o); \mathsf{grab}(o)$ and covered by the transition rules for lock handling (Fig. 3).

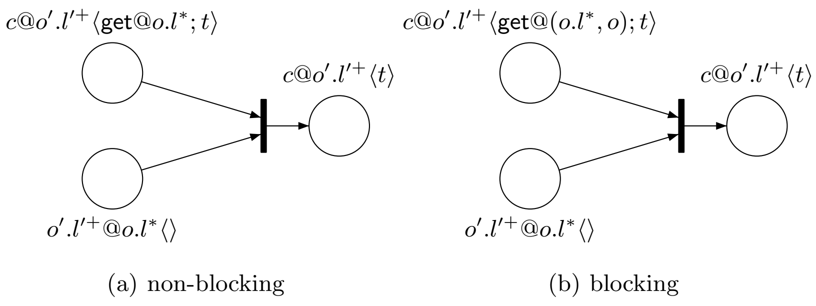

$c@o'.l'^+\langle \mathsf{get}@o.l^*; t\rangle$
$c@o'.l'^+\langle t\rangle$
$o'.l'^+@o.l^*\langle\rangle$
(a) non-blocking
$c@o'.l'^+\langle \mathsf{get}@(o.l^*, o); t\rangle$
$c@o'.l'^+\langle t\rangle$
$o'.l'^+@o.l^*\langle\rangle$
(b) blocking

Fig. 4. Translation of a claim of a result

The translation of conditional rescheduling on the other hand deviates from the operational semantics of the claim statement. In opposite to the concrete case the object lock is always released upon reaching the claim statement. The statement $\mathsf{claim}@(n, o)$ is translated to the sequence $\mathsf{release}(o); \mathsf{get}@n; \mathsf{grab}(o)$ (by function s_3). In the concrete case the lock is only released if the result, that the process is waiting for, is not available. In case the result is available the process continues its execution without rescheduling.

This deviation is justified by the syntactical transformation. In the concrete case the rules for the operational semantics have to cover both the first claim of a result and the subsequent claims. In case of a subsequent result the claim statement has to be executed without rescheduling since the existence of the result has been proven by the previous claim. In the abstract semantics, consecutive claims have been removed, i.e. each claim in the abstract case is the first claim of the result. This justifies the deviation from the operational semantics.

4.4 Petri Net Construction for *Creol* Programs

We complete the definition of the Petri net associated to an initial configuration.

Definition 5. *Given an initial configuration*

$$C_0 = c_0[(F_0, M_0)] \parallel \ldots \parallel o_0[c_{o_0}, F_0, L_0] \parallel \ldots \parallel o_n[c_{o_n}, F_n, L_n] \parallel run\langle t\rangle$$

the corresponding Petri net P_{C_0} has one starting place start, the lock places $o_0, \ldots, o_n$, and the places $n\langle t'\rangle$ with:

1. *$n = run@run$ or $n = run@o_i.l_j$ or $n = o_{i'}.l_{j'}@o_i.l_j$ with l_j and $l_{j'}$ methods of the classes c_j and $c_{j'}$, respectively. Same condition holds for abstract names containing the marker ?;*
2. *if $n = run@run$ then t' is a suffix of one of the traces in $ST(t)$;*
3. *if $n = c@o_i.l_j$ then t' is a suffix of one of the traces in $ST(\mathsf{grab}(o_i); m[o_i/self]; \mathsf{release}(o_i))$, where m is the method definition of l_j, namely, given the class c_i of o_i and $c_i[(F_i, M_i)]$, we have $[M_i] = [\ldots, l_j = \varsigma(self{:}T_0).\lambda(\vec{x}_0{:}\vec{T}_0).m, \ldots]$.*

The initial marking of P_{C_0} has one token in the places start, $o_0, \ldots, o_n$. The transitions are defined as already described in Section 4.3.

Notice that in item 3, statements $\mathsf{grab}(o_i)$ and $\mathsf{release}(o_i)$ are added because processes have to acquire the lock before start running and and it has to be released when the computation is complete. In addition, notice also that keyword *self* is replaced by the appropiate object.

5 Deadlock Freedom

The Petri net translation of a program is an over-approximation of the behavior of the program. Due to the over-approximation the Petri net might contain more deadlocks than the concrete program. By proving the Petri net to be deadlock free we prove the concrete program to be deadlock free.

We give a Petri net representation of the notion of extended deadlock in terms of marking of the Petri net. These markings can be detected by reachability analysis. By proving the absence of the deadlock markings in the Petri net we prove deadlock freedom of the program.

When speaking about a Petri net, we implicitly assume that the Petri net was derived from a program by the above mentioned translation. We only focus in the extended deadlock because it subsumes the classical one (Corollary 1).

5.1 Extended Deadlock Marking

An extended deadlock in the Petri net can be characterized in terms of a marking. This particular marking is just the mapping of Definition 4 to the Petri net context more some extra conditions that we explain after definition.

Definition 6 (Extended Deadlock Marking). *A marking m in a Petri net is an* extended deadlock marking *iff the set of places in the Petri net can be divided in three disjoint sets P_1, P_2 and P_3 such that*

1. *P_1 is a set of places of the form $o.l^{+}@o'.l'^{*}\langle \mathsf{get}@(o'.l'?, o); t\rangle$, $o.l^{+}@o'.l'^{*}\langle \mathsf{get}@o'.l'?; t\rangle$ or $o.l^{+}@o'.l'^{*}\langle \mathsf{grab}(o); t\rangle$ such that*
   - *(a) if $+ = ?$ then there is $p \in P_1$ in the form $c@c'\langle \mathsf{get}@(o.l?, o'); t'\rangle$ or $c@c'\langle \mathsf{get}@o.l?; t'\rangle$;*
   - *(b) if $* = ?$ then there is $p \in P_1$ in the form $c@c'\langle \mathsf{get}@(o'.l'?, o'); t'\rangle$ or $c@c'\langle \mathsf{get}@o'.l'?; t'\rangle$;*
   - *(c) if $t = \mathsf{grab}(o); t'$ then t' does not contain a claim with a question mark and there is $p \in P_1$ with the form $c@c'\langle \mathsf{get}@(o'.l'?, o); t''\rangle$.*

   All the places of P_1 have at least one token in m.
2. *P_2 is a set of places $c@c'\langle t\rangle$ such that one of the following holds*
   - *(a) c, c' and t do not contain question marks;*
   - *(b) c' and t do not contain question marks and if $c = o.l?$ then there is $c''@o.l?\langle t'\rangle \in P_1$.*

   All the places of P_2 have zero or more tokens in m.
3. *All the remaining places, composing the set P_3, have zero tokens in m.*

Conditions in item (1) are the condition defined in Definition 4 adapted to the Petri net context. In addition extra conditions are added to ensure the consistency between the marked calls and the marked abstract names. Conditions in (2) refer to the places that can be used to represent an active process that does do not belong to the deadlock and cannot produce the token confusion. This is evident in condition (2a), because there are no marks. On the other hand, condition (2b) is the abstract representation of a process that was called by another process that belongs to the deadlock. Notice that this process cannot create a token confusion because t has not a marked claim, then it could not do a marked call. Condition (3) is imposed to avoid the token confusion in the marked calls. Notice that P_3 are the places with a question mark that do not belong to P_1 or P_2. Not allowing tokens in these places guarantees that token confusion is not possible.

Theorem 1 (Inclusion of Extended Deadlock). *Given a Creol program, if it has an extended deadlock which is reachable, then the corresponding Petri net has a reachable extended deadlock marking.*

To prove Theorem 1 we define a mapping from both a *Creol* configuration and the *Creol* execution that reaches this configuration, to a set of markings in the Petri net. We prove that the mapping is sound, i.e. if C is a *Creol* configuration reached with an execution α and it reaches *Creol* configuration C' with a step of the operational semantics, then all markings associated to C' are reachable from at least one marking associated to C. Finally, we apply this mapping to a *Creol* execution that reaches a deadlock and we show that there is a marking in the set of reachable markings in the Petri net that satisfies the conditions in Definition 6. The definition of the mapping and the proofs are in [6].

Due to the connection between the extended deadlock in the *Creol* configuration and the extended deadlock marking in the Petri net, we can conclude freedom of extended deadlock of the program from freedom of extended deadlock markings of the Petri net.

As a final remark, we observe that the reachability of an extended deadlock marking is decidable in a Petri net. This is a consequence of the decidability of the *target reachability* problem for Petri nets [4]. Such a problem consists in checking whether a marking is reachable which satisfies some given lower bounds (possibly 0) and upper bounds (possibly 0 or ∞) associated to the places.

Notice that it is not possible to reduce our reachability problem to a coverability problem because we need to check the "absence" of tokens. In particular, we need to check the absence of return values or active methods which are computing the marked return value.

6 Conclusion

In this paper we presented a technique based on Petri net translation and Petri net reachability analysis to detect deadlock in systems made of asynchronously communicating active objects where futures are used to handle return values which can be retrieved via a lock detaining get primitive or a lock releasing claim primitive. We showed soundness of our analysis with respect to extended deadlocks (which encompass also blocked processes in addition to blocked objects considered in the classical notion of deadlock), i.e. if the analysis does not detect any deadlock then we are guaranteed that the original system is deadlock free.

Concerning the other direction, we claim our technique to be complete apart from false positives due to abstraction from data values, i.e. transformation of "if" primitives into non-deterministic choices (which obviously leads to new behavioral possibilities, hence deadlocks, with respect to the original system).

We now make some remark concerning related and future work.

We would like to mention the work in [9,10]. The authors deal with a similar language but use a different technique to discover deadlock: an abstract global system behaviour representation is statically devised from the program code in the form of a transition system whose states are labeled with set of dependencies (basically pairs of objects representing an invocation from an object to another one). The system is, then, deadlock free if no circular dependency is found. With respect to [9,10] our analysis is somehow more precise in that it is process based (i.e. also detecting extended deadlocks) and not just object based. An example of a false positive detected by the [9,10] approach, taken from [10] itself (and translated to our language), follows.

Consider the program consisting of two objects o_1 and o_2 belonging to classes c_1 and c_2, respectively, with c_1 defining methods m_1 and m_3 and c_2 defining method m_2. Such methods, plus the (static) initial method *run* are defined as:

- $run() ::= o_1.m_1()$
- $m_1() ::= \mathsf{let}\, x = o_2.m_2()\ \mathsf{in}$
  $\mathsf{claim@}(x, self); ret$
- $m_2() ::= \mathsf{let}\, x = o_1.m_3()\ \mathsf{in}$
  $\mathsf{get@}(x_2, self); ret$
- $m_3() ::= ret$

This program would originate a deadlock if we had a get instead of a claim in method m_1. This because method m_1 would call method m_2, which in turn would call m_3 which would not be able to proceed because the lock on object o_1 would be kept by m_1 waiting on the get. Differently from [9,10], our analysis correctly detects that the system is deadlock free in that method m_1 is waiting on a claim instead of a get.

Concerning language expressivity, [9,10] additionally considers, with respect to our language, a (finitely bound) "new" primitive for object creation and the capability of accounting for (a finite set of) objects used as values (e.g. passed as parameters or stored in fields) in the analysis. Concerning the former, only objects within a finite set of object names can be created (if invocations to the "new" primitive exceed the amount of available object names, as in the case of recursive object creation, old objects are returned), thus such primitive can be easily encoded in our approach by considering all the objects in the set of object names to be present since the beginning (and then "activated"). Concerning the latter, we can quite easily extend the language abstraction considered in our analysis by considering objects, out of a finite set, passed to methods (by considering object names as part of the method name). Dealing with objects stored in fields would however require an extension of the encoding into the Petri net, where a different place is considered for each possible object to be stored. We plan to do such extensions and to prove our claim about completeness as a future work.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Monographs in Computer Science. Springer (1996)
2. Ábrahám, E., Grabe, I., Grüner, A., Steffen, M.: Behavioral interface description of an object-oriented language with futures and promises. Journal of Logic and Algebraic Programming 78(7), 491–518 (2009)
3. Armstrong, J.: Erlang. Communications of ACM 53(9), 68–75 (2010)
4. Busi, N., Zavattaro, G.: Deciding reachability problems in turing-complete fragments of mobile ambients. Mathematical Structures in Computer Science 19(6), 1223–1263 (2009)
5. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. SIGPLAN Not. 39(1), 123–134 (2004)
6. de Boer, F.S., Bravetti, M., Grabe, I., Lee, M., Steffen, M., Zavattaro, G.: A petri net based analysis of deadlocks for active objects and futures, extended version (2012), http://cs.famaf.unc.edu.ar/~lee/publications/facs12_complete.pdf
7. Dijkstra, E.W.: Cooperating sequential processes. In: Genuys, F. (ed.) Programming Languages: NATO Advanced Study Institute, pp. 43–112. Academic Press (1968)
8. Edward, J., Coffman, G., Elphick, M.J., Shoshani, A.: System deadlocks. ACM Computing Surveys 3(2), 67–78 (1971)
9. Giachino, E., Laneve, C.: Analysis of Deadlocks in Object Groups. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE 2011. LNCS, vol. 6722, pp. 168–182. Springer, Heidelberg (2011)
10. Giachino, E., Laneve, C., Lascu, T.: Deadlock and livelock analysis in concurrent objects with futures. Technical report, University of Bologna (December 2011), http://www.cs.unibo.it/~laneve/publications.html
11. Gordon, A.D., Hankin, P.D.: A concurrent object calculus: Reduction and typing. In: Nestmann, U., Pierce, B.C. (eds.) Proceedings of HLCL 1998. Electronic Notes in Theoretical Computer Science, vol. 16.3, Elsevier Science Publishers (1998)
12. Smith, S.F., Agha, G.A., Mason, I.A., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming (1997)
13. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science 410(2-3), 202–220 (2009)
14. Holt, R.C.: Some deadlock properties of computer systems. ACM Computing Surveys 4(3), 179–196 (1972)
15. Johnsen, E.B., Owe, O.: An Asynchronous Communication Model for Distributed Concurrent Objects. Software and Systems Modeling (2007)