Fundamenta Informaticae 159 (2018) 197–256
DOI 10.3233/FI-2018-1663
IOS Press

A Petri Net Based Modeling of Active Objects and Futures

Frank S. de Boer
Centrum Wiskunde and Informatica
Department of Software Engineering
Amsterdam, The Netherlands
F.S.de.Boer@cwi.nl

Mario Bravetti*
Department of Computer Science and Engineering /
Focus Team INRIA, University of Bologna
Bologna, Italy
mario.bravetti@unibo.it

Matias D. Lee
Faculty of Mathematics,
Astronomy and Physics (FAMAF)
National University of Córdoba
Córdoba, Argentina
lee@famaf.unc.edu.ar

Gianluigi Zavattaro
Department of Computer Science and Engineering /
Focus Team INRIA, University of Bologna
Bologna, Italy
gianluigi.zavattaro@unibo.it

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.

Keywords: Petri nets, active objects, deadlock analysis

*Address for correspondence: Department of Computer Science and Engineering / Focus Team INRIA, University of Bologna, Mura A. Zamboni 7, 40127, Bologna, Italy.

Received January 2018; revised January 2018.

1. Introduction

The increasing importance of distributed systems demands flexible communication between distributed components. In programming languages like Erlang [1] and Scala [2] 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 [3] 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 [4]. The other one is an extended notion of deadlock based on blocked processes which is more general than the classical one. We then show how to encode into Petri nets these systems based on asynchronously communicating active objects in such a way that the problem of checking deadlock can be reduced to the problem of checking the reachability of a specific class of markings in the Petri net. More precisely, the impossibility to reach specific markings representing deadlocks in the Petri net guarantees deadlock freedom of the concrete system.

The formally defined language that we consider is Creol [5] (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 executing threads (one processor for each object). Different objects communicate only by asynchronous method calls, i.e., similar to message passing in Actor models [6]; 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 thread is created to execute the method. Methods can have processor release points that define interleaving points explicitly. When a thread 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 thread 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 thread is *nondeterministically* selected for execution, i.e., scheduling is left unspecified in Creol in favor of more abstract modeling. Since the processor is released only when explicitly requested by the owning thread, this model of concurrency is usually called *cooperative*.

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.[4]. 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 threads from making any progress.

A more general characterization by Holt [8] focuses on the processes and not on the resources. According to Holt 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:

- $run() \triangleq o_1.m_1(5)$
- $m_1(x) \triangleq \mathsf{let}\, x_1 = o_2.m_2(x)\, \mathsf{in}\, (\mathsf{let}\, x_2 = \mathsf{get}@(x_1, \mathit{self})\, \mathsf{in}\, x_2+1)$
- $m_2(x) \triangleq \mathsf{let}\, y_1 = o_1.m_3(x)\, \mathsf{in}\, (\mathsf{let}\, y_2 = \mathsf{get}@(y_1, \mathit{self})\, \mathsf{in}\, y_2+1)$
- $m_3(x) \triangleq x+1$

This program is expected to perform a chain of three method invocations, with the initial value 5 that should be incremented three times, one increment from each method invoked. x_1 and y_1 are future variables that are accessed with the blocking *get* statement: the object specified in the second argument of *get*, i.e. the $self$ (or "this") object, singles out the processor that the thread executing *get* is locking. When the method call associated to the future variable terminates, the *get* statement yields the value returned by the method (which in this example is put in variable x_2 and y_2). 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:

- $m_2(x) \triangleq \mathsf{let}\, y_1 = o_1.m_3(x)\, \mathsf{in}\, (\mathsf{let}\, y_2 = \mathsf{claim}@(y_1, \mathit{self})\, \mathsf{in}\, y_2+1)$

In this case, object o_2 is not blocked because m_2 performs a *claim* instead of a *get*: the *claim* statement implicitly performs a conditional release by checking whether the called method (associated with the future variable y_1), i.e. method m_3, is already terminated. In the case such a guard is not satisfied, the control on the specified $self$ object is released and *claim* waits the guard to become true before trying to regain it. However, the thread executing m_2 will never re-start its execution after having released the control, in that method m_3 will never terminate (actually it does not even start execution). This because, as in the original example, object o_1 is blocked by m_1.[1] We call *extended* deadlock this case of deadlock at the level of threads.

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 finite Petri nets, the abstract semantics abstract away several details of Creol,

[1] Notice that in case also m_1 performs a *claim* statement instead of a *get*, then the deadlock would disappear.

like data manipulation (ranging over infinite sets of possible values) and the precise identification of threads (because unboundedly many distinct threads can be dynamically created). In particular, we follow the idea of representing thread identifiers (hence also the corresponding future variables that will contain the value returned by the thread) 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 thread could access a wrong future simply because it has the same abstract name. Consider, for instance, the following example:

- $run() \triangleq o_1.m_1(1,5)$
- $m_1(y,z) \triangleq \mathsf{let}\, x_1 = o_2.m_2(y)\, \mathsf{in}$
  $\mathsf{let}\, x_2 = o_2.m_2(z)\, \mathsf{in}$
  $\mathsf{let}\, y_2 = \mathsf{get}@(x_2, \mathit{self})\, \mathsf{in}$
  $\mathsf{let}\, y_1 = \mathsf{claim}@(x_1, \mathit{self})\, \mathsf{in}\, y_1 + y_2$
- $m_2(x) \triangleq \mathsf{let}\, y_3 = (\,\mathsf{if}\, x = 1\, \mathsf{then}\, 2\, \mathsf{else}\, (\mathsf{let}\, x_3 = o_1.m_3(x)\, \mathsf{in}\, \mathsf{claim}@(x_3, \mathit{self}))\,)\, \mathsf{in}\, y_3$
- $m_3(x) \triangleq x+1$

Notice that method m_2 is first invoked with parameter 1 and then with parameter 5; in both cases, m_2 is expected to increment the received parameter, but if it is not 1 then the actual increment is delegated to method m_3. But as method m_3 is invoked on object o_1, which is blocked by method m_1, we have that the program deadlocks (method m_1 waits for the second invocation of method m_2, which waits for the execution of m_3, which is itself waiting to acquire the lock of o_1 which is blocked by m_1).

According to the above abstraction, both the futures associated to the two invocations of method m_2, namely 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 the previously described deadlock (namely, when *get* is performed on x_2), according to the abstract semantics no deadlock is generated. In fact, the return value of the first call unblocks the *get* as the two futures have the same name in the abstract semantics. To overcome this limitation, we add in the abstract semantics tagged versions of the methods: when a method m is invoked, the abstract semantics nondeterministically selects either the standard version of m or its tagged version denoted with "$m?$". The intuition behind this technique is that we tag invocations that will be directly involved in the deadlock. In the above example, the idea is to abstractly represent x_1 with $o_1.m_1@o_2.m_2$ while x_2 is represented with $o_1.m_1@o_2.m_2?$: the tag on the second invocation of m_2 indicates that this second invocation will be directly involved in the reached deadlock.

In general, a method m and its tagged version "$m?$", have the same behavior, but the return value will be stored in two futures with two distinct abstract names. In the example above, there will be no swap between the two futures x_1 and x_2 as their abstract names will be $o_1.m_1@o_2.m_2$ and $o_1.m_1@o_2.m_2?$, respectively, and the system will deadlock also under the abstract semantics.

This tagging technique guarantees that, for a deadlocking program, there exists, also under the abstract semantics, a nondeterministic execution in which we tag method executions whose future is waited for by methods involved in the deadlock. In this way futures that are waited for by methods inside the deadlock can never be confused with futures produced by methods that are outside the

deadlock. It is worth to notice that only two versions of the same method (i.e. m and "$m?$") are sufficient because the tagged version is used for invocations directly involved in the deadlock, non tagged for the other ones.

Besides this abstraction technique on future names we also need to abstract from data. Also other more technical transformations are needed, for instance to avoid useless repeated accesses to the same future variables. The adopted abstractions and program transformations, used to define our Petri net semantics for Creol programs, have anyway a fundamental property: the traces of computation that leads a Creol program to an extended deadlock state, are still present in the corresponding Petri net semantics where they lead to particular Petri net markings that we call *extended deadlock markings*. It could happen that the abstract semantics add spurious deadlock, but it cannot remove them; hence, from the point of view of the presence of deadlocks, the abstract model is an over-approximation of the original system.

This allows us to conclude that the abstract semantics makes it possible to obtain a decidable way for proving extended deadlock freedom. In fact, reachability problems are in general decidable in Petri nets, and we show that also our specific case of reachability of an extended deadlock marking is decidable.

As additional results, we show how our technique (i) can be adapted in order to prove classical deadlock freedom, and (ii) how it can be made more precise to faithfully represent fields and passed/returned values of class type. Concerning the first of these additional results, it is justified by the fact that a Creol program could have extended deadlocks, but not classical ones (i.e. it is not possible to have a set of objects completely blocked even if it possible to have a set of blocked threads). The modifications to our technique to deal with classical deadlock are minimal: it is sufficient to consider a slightly different set of reachable markings for exactly the same Petri net. Concerning the second additional result, we define a less abstract Petri net semantics in which fields, parameters and returned values of class type are modeled faithfully. This more concrete semantics could be useful in cases in which circularities among object references are generated by our abstraction technique: in fact, in our initial Petri net semantics we consider that a received object reference (i.e. read from a field/parameter or returned by a method call) could refer to any possible instance of the class corresponding to the type of such reference. This could generate circularities among object references that are not present in the considered concrete Creol semantics.

The outline of the paper is as follows. In Section 2 we report the definition of Creol language. In Section 3 we present the two notions of deadlock. In Section 4 we present the translation of Creol programs into Petri nets. In Section 5 we present the main result of the paper: we characterize the notion of deadlock markings for the Petri net semantics and we prove that if in the Petri net associated to a program deadlock markings cannot be reached, then the program is deadlock free. We also show that such reachability problem is decidable for Petri nets. Section 6 shows how the translation into Petri nets can be extended to also explicitly represent passing/returning of objects and reading/writing of fields of class type. Section 7 concludes the paper. Finally, in Appendices A and B we provide detailed proofs (and related technical machinery) showing that the Petri net translation is sound and that deadlocks in the Creol program are always detected by the Petri net.

This is a technically improved and fully developed version of [9] extended with the additional results (i) and (ii) above, presented in Sections 5.3 and 6, respectively.

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. [10], 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 result in a communication model based on asynchronously communicating active object.

Obtaining the results and cooperative scheduling. Method calls are done asynchronously and the caller obtains the result back when needed, by querying the future reference. The model here supports two variants of that querying operation: the non-blocking claim-statement, which allows reschedule of the querying code in case the result is not yet there, and the blocking get-statement, which insists on getting the result without a re-scheduling point.

Statically fixed number of objects. In this paper we omit object creation to facilitate the translation to Petri nets (according to the Petri net construction that we present, an unbounded number of objects would require an infinite Petri net).

The type system and properties of the calculus, e.g. subject reduction and absence of (certain) run-time errors, presented in [10] still apply to our simplified version of Creol. For brevity we only present explanation for language constructs relevant for deadlocks. Missing details with respect to other language constructs, or the type system, can be found in [10]. Even if we do not recall here the type system, we will consider only Creol programs that are well typed.

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.

Values v in our calculus can be expressed, directly, as names n or by means of variables x. The basic syntactic category of names n, represents references to classes, to objects, and to futures/thread identifiers. To facilitate reading, we write o for names referring to objects and c for classes. We assume names n to also include elements taken from standard data types, such as booleans, integers, etc... For the sake of simplicity, we do not explicitly include their syntactical definition because they are disregarded in our deadlock analysis, which, by using data abstraction, concentrates on the analysis of the communication behavior. Local variables and formal parameters are denoted by the syntactic category of variables x, which we assume to be always syntactically distinguished from names n. Following a similar approach, we left undefined the syntactic category T, used to denote types: the unique assumption we make is that it includes the names c of the classes.

Table 1. Abstract syntax

C	::=	$C \parallel C \mid n[\![F, M]\!] \mid n[n, F, L] \mid n\langle t\rangle$	configuration
M	::=	$[l = m, \ldots, l = m]$	methods
F	::=	$[l = n, \ldots, l = n]$	fields
m	::=	$\varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$	method definition
t	::=	$v \mid \mathsf{let}\, x{:}T = e\, \mathsf{in}\, t$	thread code
e	::=	$t \mid \mathsf{if}\, v = v\, \mathsf{then}\, e\, \mathsf{else}\, e \mid v.l(\vec{v}) \mid v.l \mid v.l := v$	expression
	|	$\mathsf{claim}@(v, n) \mid \mathsf{get}@(v, n) \mid \underline{\mathsf{get}@v}$	
	|	$\mathsf{suspend}(n) \mid \underline{\mathsf{grab}(n)} \mid \underline{\mathsf{release}(n)}$	
v	::=	$x \mid n$	values
L	::=	$\bot \mid \top$	lock status

In general a *configuration* C is a collection of classes, objects, and (named) threads. The sub-entities of a configuration are composed using the parallel-construct $\parallel$ (which is assumed to be 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 $c[\![F, M]\!]$ carries a name c and defines its fields F and methods M. 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.

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. A method $\varsigma(self{:}T).\lambda(\vec{x}{:}\vec{T}).t$ provides the method body t abstracted over the ς-bound "this" parameter $self$, belonging to the syntactic category n, and the formal parameters $\vec{x}$ —the ς-binder is borrowed from the well-known object-calculus of Abadi and Cardelli [12]. Note that the methods are stored in the classes but the fields are kept in the objects. Methods lookup and field reading /modification is denoted as follows: given the method list M containing the label l, $M.l$ yields the definition $\varsigma(self{:}T).\lambda(\vec{x}{:}\vec{T}).t$ associated to l in M; given the field list F containing the label l, $F.l$ yields the value n associated to l in F while $F[l \mapsto n]$ returns a new field list that differs from F simply because the value associated to l becomes n.

We now list some syntactic assumptions on the user syntax, like the initial presence of one thread only, named run, or the presence in the initial configuration of the definitions for all the classes to which the objects belong.

User syntax restrictions. The most significant assumption is that, when a thread accesses a future, we impose that it corresponds to a call previously performed by the same thread. Formally, we impose that commands $\mathsf{claim}@(v, n)$ and $\mathsf{get}@(v, n)$ are such that v (denoting a future, see below) is a variable x that is bound by a corresponding $\mathsf{let}\, x{:}T = e\, \mathsf{in}\, t$, where the e expression result is obtained by: performing a method call $o.l(\vec{v})$ or reading a variable y satisfying, itself, the same property.

Moreover, commands $\mathsf{claim}@(x,n)$, $\mathsf{get}@(x,n)$ and $\mathsf{suspend}(n)$ are such that the name n (denoting an object on which the lock is acquired/released, see below) is the ς-bound name, denoting the self (or "this") object in the method where they occur. For each method definition $\varsigma(n{:}T).\lambda(x_1{:}T_1,\ldots,x_n{:}T_n).t$ we assume that the unique variables that occur free in t are the parameter variables $x_1,\ldots,x_n$.

Finally, we assume that, in *user* syntax, the root C of the syntactical specification, also representing the *initial* configuration at run-time, includes a collection of:

- any number of class definitions $c[\![F, M]\!]$, each one identified with a distinct class name c;
- any number of instantiated objects $o[c, F, L]$, with o being the object name, c the class name the object belongs to (every occurrence having a different object name and referring to the name of an existing class, in such a way that there exists at least an object for each class), F being the same as the F inside the c class definition (i.e. initial value of fields is established according to field definitions) and L being $\perp$ (i.e. all object locks are initially free);
- only one occurrence of a thread $n\langle t\rangle$, with n being the special thread name run (the initial thread). The body t of the initial thread, besides the constraints for standard method body definitions, must also be such that it does not include free variables x (as for a method with no parameters) and occurrences of claim, get and suspend commands (this because, for the sake of simplicity, we do not consider an initial object).

Globally, in the root C we also assume that, in methods definitions and in the initial thread run, only object names for which there exists an object instance in C may occur free. Notice that this corresponds to the restriction, we consider, of not allowing object instantiation at run-time: we thus assume identities of existing objects to be known, i.e. directly referable in the code.

Example 2.1. As an example of user defined syntax, we start with the formal presentation of (a slightly modified version of) a Creol program which has been already informally discussed in the Introduction; namely, the second one used to discuss our technique for detecting deadlocks. Consider two classes c_1 and c_2 and two initial objects o_1 and o_2 belonging to such classes, respectively. Namely, consider the initial configuration

$$\begin{aligned} C_0 \;=\;& c_1[\![[\,], [l_1 = m_1, l_3 = m_3]]\!] \parallel c_2[\![[\,], [l_2 = m_2]]\!] \parallel o_1[c_1, [\,], \perp] \parallel o_2[c_2, [\,], \perp] \parallel \\ & run\langle \mathsf{let}\, x = o_1.l_1() \,\mathsf{in}\, 0\rangle \end{aligned}$$

with methods l_1, l_2 and l_3 defined as:

$$\begin{aligned} m_1 \;\triangleq\;& \varsigma(\mathit{self}).\lambda().\quad \mathsf{let}\, x_1 = o_2.l_2(1)\,\mathsf{in} \\ & \qquad\qquad \mathsf{let}\, x_2 = o_2.l_2(5)\,\mathsf{in} \\ & \qquad\qquad\quad \mathsf{get}@(x_2, \mathit{self})\,;\,(\mathsf{claim}@(x_1, \mathit{self}); 0) \\ m_2 \;\triangleq\;& \varsigma(\mathit{self}).\lambda(x).\quad (\,\mathsf{if}\, x = 1\,\mathsf{then}\, 0\,\mathsf{else}\,(\mathsf{let}\, x_3 = o_1.l_3()\,\mathsf{in}\,\mathsf{claim}@(x_3, \mathit{self}))\,)\,;\,0 \\ m_3 \;\triangleq\;& \varsigma(\mathit{self}).\lambda().\quad 0 \end{aligned}$$

Notice that, for simplicity, we have omitted type declarations and, if compared with the corresponding example in the Introduction, we avoid some useless parameter passings as well as some arithmetic operations. □

Creol programs behaviour. 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 get@(x, o) operation is easier: it blocks if the result of the future n in x is not (yet) available, i.e., if, at run-time, the thread n is not of the form of $n\langle n'\rangle$, with n' being the returned value. The claim@(x, o) is a more "cooperative" version of get@(x, o): 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.

As usual we use sequential composition $e; t$ as syntactic sugar for let $x{:}T = e$ in t, when x does not occur free in t. We refer to [10] 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

Axioms and rules of the operational semantics are shown in Table 2, where reduction steps are denoted by labeled transitions denoted with $\xrightarrow{\lambda}$. The label λ, ranging over the set of possible labels $\{\tau, n, [n_1, n_2, o.l]\}$, is used to carry information that will be useful in Definitions A.1, A.2, and A.3. More specifically, the labels have the following meaning: label n means that the return value of the thread n is accessed, $[n_1, n_2, o.l]$ stands for a method call executed by thread n_1 on method l of object o with creation of a new thread n_2, while τ does not carry specific information (hence it is usually omitted).

As usual in reduction semantics, axioms assume to have the components involved in the step in predefined positions w.r.t. the $\|$ composition operator: commutativity and associativity of such operator are used to readjust the order of the components in such a way the axiom can be applied. A contextual rule is then considered to lift the computation step to a richer configuration that contains also other components besides those directly involved in the reduction.

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 $\xrightarrow{\lambda}$. We denote executions by $C_0 \to \ldots \to C_n$ omitting, for simplicity, the transition labels.

Invoking a method (cf. rule FUT) creates a new future reference and a corresponding thread is added to the configuration. In the configuration after the reduction step, the notation $M.l(o)(\vec{v})$ stands for $t[o/s][\vec{v}/\vec{x}]$, when $M.l$ is $\varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t$. Here and in the following, we use the $t[v/x]$ notation to denote the result of syntactically replacing x by v in the thread code t. As usual, only free variables are replaced: in this context the variable binding operator is the let statement. The notation above is extended to vector replacement $[\vec{v}/\vec{x}]$, standing for replacement of each variable in $\vec{x}$ by the value in $\vec{v}$ that is placed in the same vector position. Similarly, we use $t[n'/n]$ to denote syntactical replacement of a name n by another name n' (notice that in thread code t there are no binders for names). Upon

Table 2. Operational semantics

$$n\langle \mathsf{let}\, x{:}T = n' \,\mathsf{in}\, t\rangle \to n\langle t[n'/x]\rangle \quad \textsc{Red}$$

$$n\langle \mathsf{let}\, x{:}T = (\mathsf{let}\, x'{:}T' = e' \,\mathsf{in}\, t') \,\mathsf{in}\, t\rangle \to n\langle \mathsf{let}\, x'{:}T' = e' \,\mathsf{in}(\mathsf{let}\, x{:}T = t' \,\mathsf{in}\, t)\rangle \quad \textsc{Let}$$

$$n\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, n = n \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2) \,\mathsf{in}\, t\rangle \to n\langle \mathsf{let}\, x{:}T = e_1 \,\mathsf{in}\, t\rangle \quad \textsc{Cond}_1$$

$$n\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, n_1 = n_2 \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2) \,\mathsf{in}\, t\rangle \to n\langle \mathsf{let}\, x{:}T = e_2 \,\mathsf{in}\, t\rangle \quad \text{with } n_1 \neq n_2 \quad \textsc{Cond}_2$$

$$o[c, F, L] \parallel n\langle \mathsf{let}\, x{:}T = o.l \,\mathsf{in}\, t\rangle \to o[c, F, L] \parallel n\langle \mathsf{let}\, x{:}T = n' \,\mathsf{in}\, t\rangle \quad \text{with } n' = F.l \quad \textsc{FLookup}$$

$$o[c, F, L] \parallel n\langle \mathsf{let}\, x{:}T = (o.l := n') \,\mathsf{in}\, t\rangle \to o[c, F', L] \parallel n\langle \mathsf{let}\, x{:}T = n' \,\mathsf{in}\, t\rangle \quad \text{with } F' = F[l \mapsto n'] \quad \textsc{FUpdate}$$

$$c[\![F', M]\!] \parallel o[c, F, L] \parallel n_1\langle \mathsf{let}\, x{:}T = o.l(\vec{v}) \,\mathsf{in}\, t_1\rangle \xrightarrow{[n_1, n_2, o.l]} c[\![F', M]\!] \parallel o[c, F, L] \parallel n_1\langle \mathsf{let}\, x{:}T = n_2 \,\mathsf{in}\, t_1\rangle \parallel n_2\langle \mathsf{let}\, y{:}T_2 = (\mathsf{grab}(o); t_2) \,\mathsf{in}\, \mathsf{release}(o); y\rangle \quad \text{with } n_2 \text{ fresh and } t_2 = M.l(o)(\vec{v}) \quad \textsc{Fut}$$

$$n_1\langle n\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = \mathsf{claim}@(n_1, o) \,\mathsf{in}\, t\rangle \xrightarrow{n_1} n_1\langle n\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = n \,\mathsf{in}\, t\rangle \quad \textsc{Claim}_1$$

$$n_1\langle t_1\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = \mathsf{claim}@(n_1, o) \,\mathsf{in}\, t_2\rangle \xrightarrow{n_1} n_1\langle t_1\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = (\mathsf{release}(o); \mathsf{get}@n_1) \,\mathsf{in}\, \mathsf{grab}(o); t_2\rangle \quad \text{with } \nexists n : t_1 = n \quad \textsc{Claim}_2$$

$$n_1\langle n\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = \mathsf{get}@(n_1, o) \,\mathsf{in}\, t\rangle \xrightarrow{n_1} n_1\langle n\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = n \,\mathsf{in}\, t\rangle \quad \textsc{Get}_1$$

$$n_1\langle n\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = \mathsf{get}@n_1 \,\mathsf{in}\, t\rangle \xrightarrow{n_1} n_1\langle n\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = n \,\mathsf{in}\, t\rangle \quad \textsc{Get}_2$$

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

$$o[c, F, \bot] \parallel n\langle \mathsf{grab}(o); t\rangle \to o[c, F, \top] \parallel n\langle t\rangle \quad \textsc{Grab}$$

$$o[c, F, \top] \parallel n\langle \mathsf{release}(o); t\rangle \to o[c, F, \bot] \parallel n\langle t\rangle \quad \textsc{Release}$$

$$\frac{C_1 \xrightarrow{\lambda} C_1'}{C_1 \parallel C_2 \xrightarrow{\lambda} C_1' \parallel C_2} \quad \textsc{Context}$$

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 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 re-acquire the lock. Other reduction rules are straightforward.

Example 2.2. To show how the operational semantics rules can be applied to a configuration, we now present a couple of reduction steps starting from the initial configuration C_0 defined in the Example 2.1. By applying rules FUT and CONTEXT we have $C_0 \xrightarrow{[run,n_1,o_1.l_1]} C_1$ with

$$\begin{aligned} C_1 \;=\;& c_1[\![[\,],[l_1{=}m_1,l_3{=}m_3]]\!] \parallel c_2[\![[\,],[l_2{=}m_2]]\!] \parallel o_1[c_1,[\,],\bot] \parallel o_2[c_2,[\,],\bot] \parallel \\ & run\langle\ \mathsf{let}\ x{=}n_1\ \mathsf{in}\ 0\ \rangle \parallel \\ & n_1\langle\ \mathsf{let}\ z{=} \\ & \qquad \mathsf{grab}(o_1); \\ & \qquad\quad \mathsf{let}\ x_1{=}o_2.l_2(1)\ \mathsf{in} \\ & \qquad\qquad \mathsf{let}\ x_2{=}o_2.l_2(5)\ \mathsf{in}\ (\ \mathsf{get@}(x_2,o_1);(\mathsf{claim@}(x_1,o_1);0)\,) \\ & \quad \mathsf{in}\ (\mathsf{release}(o_1);z)\ \rangle \end{aligned}$$

By applying rules GRAB and CONTEXT we have $C_1 \to C_2$ with

$$\begin{aligned} C_2 \;=\;& c_1[\![[\,],[l_1{=}m_1,l_3{=}m_3]]\!] \parallel c_2[\![[\,],[l_2{=}m_2]]\!] \parallel o_1[c_1,[\,],\top] \parallel o_2[c_2,[\,],\bot] \parallel \\ & run\langle\ \mathsf{let}\ x{=}n_1\ \mathsf{in}\ 0\ \rangle \parallel \\ & n_1\langle\ \mathsf{let}\ z{=} \\ & \qquad \mathsf{let}\ x_1{=}o_2.l_2(1)\ \mathsf{in} \\ & \qquad\quad \mathsf{let}\ x_2{=}o_2.l_2(5)\ \mathsf{in}\ (\ \mathsf{get@}(x_2,o_1);(\mathsf{claim@}(x_1,o_1);0)\,) \\ & \quad \mathsf{in}\ (\mathsf{release}(o_1);z)\ \rangle \end{aligned}$$

In the continuation of the above computation we have that the method l_2 of o_2 is invoked twice with parameters 1 and 5, respectively. In the first case the method l_2 returns immediately, while in the second one it invokes l_3 on o_1. Namely, the following configuration is reached (assuming n_2, n_3 and n_4 be the fresh thread names used to identify the two invocations of method l_2 on o_2 and method l_3 on o_1, respectively):

$$\begin{aligned} C \;=\;& c_1[\![[\,],[l_1{=}m_1,l_3{=}m_3]]\!] \parallel c_2[\![[\,],[l_2{=}m_2]]\!] \parallel o_1[c_1,[\,],\top] \parallel o_2[c_2,[\,],\top] \parallel \\ & run\langle\ \mathsf{let}\ x{=}n_1\ \mathsf{in}\ 0\ \rangle \parallel \\ & n_1\langle\ \mathsf{let}\ z{=}\ \mathsf{get@}(n_3,o_1);(\mathsf{claim@}(n_2,o_1);0)\ \mathsf{in}\ (\mathsf{release}(o_1);z)\ \rangle \parallel \\ & n_2\langle 0\rangle \parallel \\ & n_3\langle\ \mathsf{let}\ z{=}\ (\mathsf{claim@}(n_4,o_2);0)\ \mathsf{in}\ (\mathsf{release}(o_2);z)\ \rangle \parallel \\ & n_4\langle\ \mathsf{let}\ z{=}\ (\mathsf{grab}(o_1);0)\ \mathsf{in}\ (\mathsf{release}(o_1);z)\ \rangle \end{aligned}$$

This, however, will cause object o_1 to block indefinitely; in fact, thread n_4 will never acquire the o_1 lock because n_1 remains blocked while holding such lock. This happens because n_1 waits for n_3, that waits for n_4, which is blocked by n_1. □

3. Deadlock

As we already explained in the Introduction, we give two different notions of deadlock in Creol. The first one, we call *classical* deadlock, follows [4]. In this case not only threads are blocked but also the

objects hosting them, as it happens in the above Example 2.2. The second notion, we call *extended* deadlock, resembles the definition of deadlock by Holt [8]. In this case, instead of looking at blocked objects we look at blocked threads. A blocked thread does not necessarily block the object hosting it.

Before formally defining these two notions of deadlock, we consider again the first example we presented in the Introduction: the example of classical deadlock and, by modifying one method, of extended deadlock. We now present (a slightly modified version of) the Creol program for such an example and we give a formal argument for it to originate a classical/extended deadlock.

Example 3.1. Consider an initial configuration including two objects o_1 and o_2 both of class C defining methods l_1, l_2 and l_3. Concerning methods, their definitions m_1, m_2 and m_3, respectively, are the following ones (also in this case, for simplicity type declarations are omitted):

$$\begin{aligned} m_1 \triangleq\ & \varsigma(self).\lambda(x).\,\mathsf{let}\, x_1 = o_2.l_2()\ \mathsf{in} \\ & \qquad\qquad \mathsf{let}\, x_2 = (\mathsf{if}\, x = 1\, \mathsf{then}\, \mathsf{get}@(x_1, self)\, \mathsf{else}\, 0)\ \mathsf{in}\ x_2 \\ m_2 \triangleq\ & \varsigma(self).\lambda().\,\mathsf{let}\, y_1 = o_1.l_3()\ \mathsf{in}\ (\mathsf{let}\, y_2 = \mathsf{get}@(y_1, self)\ \mathsf{in}\ y_2) \\ m_3 \triangleq\ & \varsigma(self).\lambda().1 \end{aligned}$$

Calling method l_1 on object o_1 with parameter value 1, that is considering the *run* thread in the initial configuration to be, e.g.,

$$run\langle \mathsf{let}\, x = o_1.l_1(1)\ \mathsf{in}\ 0\rangle$$

originates the deadlock described in the Introduction.

Formally, starting from the initial configuration, and assuming n_1, n_2 and n_3 to be the fresh thread names created by the operational semantics (rule FUT) when executing method calls $o_1.l_1(1)$, $o_2.l_2()$ and $o_1.l_3()$, respectively, the program reaches a configuration whose threads in execution are:

- $run\langle 0\rangle$
- $n_1\langle \mathsf{let}\, x_2 = \mathsf{get}@(n_2, o_1)\ \mathsf{in}\ (\ \mathsf{let}\, z = x_2\ \mathsf{in}\ (\mathsf{release}(o_1); z)\ )\rangle$
- $n_2\langle \mathsf{let}\, y_2 = \mathsf{get}@(n_3, o_2)\ \mathsf{in}\ (\ \mathsf{let}\, z = y_2\ \mathsf{in}\ (\mathsf{release}(o_2); z)\ )\rangle$
- $n_3\langle \mathsf{let}\, z' = \mathsf{grab}(o_1)\ \mathsf{in}\ (\ \mathsf{let}\, z = 1\ \mathsf{in}\ (\mathsf{release}(o_1); z)\ )\rangle$

Since in such a configuration the locks of the objects o_1 and o_2 are both taken (this can also be seen from the fact that the first statement of threads n_1 and n_2 is $\mathsf{get}@(n_2, o_1)$ and $\mathsf{get}@(n_3, o_2)$, thus they own the o_1 and o_2 object lock, respectively) the above threads can no longer proceed in the execution. As we will see from the following Definition 3.4, this configuration is a classical deadlock.

Consider now the case in which method l_2 has, instead, the following definition m_2:

$$m_2 \triangleq \varsigma(self).\lambda().\,\mathsf{let}\, y_1 = o_1.l_3()\ \mathsf{in}\ (\mathsf{let}\, y_2 = \mathsf{claim}@(y_1, self)\ \mathsf{in}\ y_2)$$

In this case, assuming again thread names created during execution to be n_1, n_2 and n_3, the program reaches a configuration where threads *run*, n_1 and n_3 are as above, while n_2 is:

- $n_2\langle \mathsf{let}\, y_2 = \mathsf{get}@n_3\ \mathsf{in}\ (\ \mathsf{grab}(o_2); \mathsf{let}\, z = y_2\ \mathsf{in}\ (\mathsf{release}(o_2); z)\ )\rangle$

As we will see from the following definitions, this configuration is not a classical deadlock (Definition 3.4), in that object o_2 is not actually blocked because thread n_2 does not own the lock (this can be seen from the fact that the first statement of thread n_2 is $\mathsf{get}@n_3$). On the contrary, we will see that this reached configuration is instead an extended deadlock (Definition 3.5). □

The example above of extended deadlock will be our running example throughout the paper.

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

Definition 3.2. (Waiting Thread)
A thread $n_1\langle t\rangle$ is waiting for:

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

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

Definition 3.3. (Blocking Thread)
A thread $n_1\langle t\rangle$ blocks object o iff t is of the form $\mathsf{let}\, x : T = \mathsf{get}@(n_2, o)\,\mathsf{in}\, t'$.

Note that a thread needs to hold the object lock and execute a blocking statement, i.e. get-statement, to block that object. Furthermore note that the threads 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.[4]. The source of interest is the exclusive access to an object represented by the object lock. In opposite to other multithreaded settings, e.g. like in *Java*, where a thread can collect a number of these exclusive rights, a thread in our 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 requires access to that object indirectly. To be more precise, a thread can derive the information that the thread created to handle its call had access to the lock of the called object, by checking the availability of the result inside the corresponding future.

Definition 3.4. (Classical Deadlock)
A configuration C is a *classical deadlock* iff there exists a set of objects O such that the following holds. For all $o \in O$, o is blocked by a thread n_1 that is waiting for a thread n_2 such that: for some object $o' \in O$, either n_2 blocks o' or n_2 is waiting for o'.

Note that the definition of "waiting for" plays a crucial role here, because while a thread is waiting, it cannot finish its computation. Being blocked by threads, the objects in O cannot execute other threads. But also the threads blocking the objects cannot proceed because they are indeed waiting for one of such non-executing threads. This generates a classical deadlock situation. Note that a blocking

thread does not necessarily directly wait for another blocking thread but can also wait for a thread which is simply waiting to have access to its object in O.

The second notion resembles the definition of deadlock by Holt [8]. Instead of looking at blocked objects we look at blocked threads. A thread can be blocked due to the execution of either a get–statement or a claim–statement. In the first case the object is blocked by the thread, in the second case only the thread is blocked. Threads that are blocked on a claim–statement are not part of a deadlock according to the first definition since they are not holding any resource. Yet they can be part of a circular dependency that prevents them from making any progress.

Definition 3.5. (Extended Deadlock)
A configuration C is an *extended deadlock* iff there exists a set of threads N such that, for all $n_1 \in N$, n_1 is waiting for $n_2 \in N$, or waiting for some object o that is blocked by $n_2 \in N$.

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

Proposition 3.6. Every classical deadlock is an extended deadlock.

Proof: Let O be the set of objects involved in a classical deadlock. We denote by $B(O)$ the set of threads blocking an object in O and by $W(B(O))$ the set of threads the threads in $B(O)$ are waiting for. Consider $N = B(O) \cup W(B(O))$: it is a finite set of threads. It remains to show that for all $n_1 \in N$, n_1 is waiting for $n_2 \in N$, or waiting for o which is blocked by $n_2 \in N$.

By definition of classical deadlock and $W(B(O))$ each thread $n_1 \in B(O)$ is waiting for a thread $n_2 \in B(O) \cup W(B(O)) = N$. Thus each thread $n_1 \in B(O)$ is waiting for a thread $n_2 \in N$.

We now consider the remaining threads in N, i.e. the threads $n_1 \in W(B(O))$. By definition of classical deadlock and $W(B(O))$ such threads are either in $B(O)$ or waiting for an object $o \in O$. In the former case $n_1 \in B(O)$ and the above reasoning applies. In the latter, by definition of classical deadlock and $B(O)$, this object o is blocked by a thread $n_2 \in B(O) \subseteq N$, i.e. n_1 is waiting for o which is blocked by a thread $n_2 \in N$. □

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.

4.1. Petri nets preliminaries

We first recall the definition of Petri nets. We adopt a simplified definition without explicit mention of the flow relations, which are replaced by the assumption that each transition $\mathtt{t}$ comes equipped with a preset ${}^\bullet\mathtt{t}$ and a postset $\mathtt{t}^\bullet$, respectively indicating the places from which tokens are consumed, and where tokens are introduced, by the transition $\mathtt{t}$.

Definition 4.1. (Petri Nets)
A Petri net is a tuple $\langle P, T, m_0 \rangle$ such that P is a finite set of places, T is a finite set of transitions, and m_0 is the initial marking, i.e. a function from P to $\mathbb{N}_0$ that defines the initial number of tokens in each place of the net. A transition $\mathtt{t} \in T$ is characterised by a function ${}^\bullet\mathtt{t}$ (preset) from P to $\mathbb{N}_0$, and a function $\mathtt{t}^\bullet$ (postset) from P to $\mathbb{N}_0$: the preset indicates the tokens that must be consumed to fire a transition, the postset indicates the tokens that are produced as effect of such firing.

Concerning Petri net execution, this is represented as a sequence of configurations that are represented by markings m. Formally, transition $\mathtt{t}$ is *enabled* at marking m iff ${}^\bullet\mathtt{t}(p) \leq m(p)$ for each $p \in P$. Enabled transitions can *fire*. Firing $\mathtt{t}$ at m leads to a new marking m' defined as $m'(p) = m(p) - {}^\bullet\mathtt{t}(p) + \mathtt{t}^\bullet(p)$, for every $p \in P$. A marking m is *reachable* in a Petri net $\langle P, T, m_0 \rangle$ if $m = m_0$ or if, starting from m_0, it is possible to produce m by firing finitely many (enabled) transitions in T.

Despite Petri nets are infinite state systems because the number of tokens that can be generated could be unbounded, many interesting properties are decidable, for instance *marking reachability* [13] (i.e. a given marking m is reachable in a given Petri net) and *coverability* [14] (i.e. there exists a marking greater[2] than a given marking m that is reachable). We will use a more expressive form of reachability that was studied in [15]: *target marking reachability*.

Definition 4.2. (Target Marking Reachability)
Let $\mathcal{P} = \langle P, T, m_0 \rangle$ be a Petri net. A *target marking denotation* is a pair of functions $(inf, sup) \in (P \to \mathbb{N}_0) \times (P \to (\mathbb{N}_0 \cup \infty))$ (with $\mathbb{N}_0$ denoting the set of non-negative integers) such that for all $p \in P$ we have $inf(p) \leq sup(p)$ (assuming $n \leq \infty$ for all $n \in \mathbb{N}_0$). We say that a marking m satisfies a target marking denotation (inf, sup) if, for all $p \in P$, we have $inf(p) \leq m(p) \leq sup(p)$. Target marking reachability is the problem of checking, given a Petri net $\mathcal{P}$ and a target marking denotation (inf, sup), whether there exists a marking satisfying (inf, sup) that is reachable in $\mathcal{P}$.

In [15] it is shown that target marking reachability is decidable; intuitively, this follows from the fact that it can be reduced to a combination of reachability and coverability that, as recalled above, are both decidable for Petri nets.

4.2. Informal introduction to the Petri net encoding of Creol

In Creol a fresh unique thread name is created for each method invocation, implying that unboundedly many distinct futures can be created during the computation of a Creol program. As Petri nets are finite, it is not possible to represent such unbounded distinct names faithfully. For this reason, we perform an abstraction: thread names are abstractly identified by a tuple of caller object o, calling method l, callee object o', and called method l'. We denote this tuple with $o.l@o'.l'$. It is interesting to observe that even if we will use in the Petri net finitely many abstract thread names, we still allow for an unbounded number of method invocations. In fact, active threads will be represented with tokens, and there is no a-priori limit to the number of tokens that can be produced within a Petri net.

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. The restriction to a version of Creol without

[2] A marking m' is greater than a marking m if $m'(p) \geq m(p)$, for every place p.

object creation allows us to keep the Petri net finite, 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 or the actual value of the passed parameters) otherwise we would need infinitely many places.

As discussed above, the threads and the corresponding futures are abstractly represented in the Petri net by using the finite set of tuples $o.l@o'.l'$. Due to this abstraction the Petri net semantics could have the following *token swap* problem. If there are two invocations of the same method l' on the object o', performed by the method l of the object o, these two invocations will be indistinguishable in the Petri net as both will be abstractly identified with $o.l@o'.l'$. More precisely, both futures will be placed in the same place (hence generating token swap).

As an example of token swap, consider the Creol program in the Example 2.2. In the reported configuration C we have the threads n_2 and n_3 that correspond to the execution of the method l_2 on object o_2, invoked by the method l_1 on object o_1. This means that the abstract name for both n_2 and n_3 is $o_1.l_1@o_2.l_2$. Having the same name, the thread n_1 which is concretely waiting for n_3, will abstractly wait for any possible return value on $o_1.l_1@o_2.l_2$. A return value on this abstract name is available, namely the return value of the concrete thread n_2, hence thread n_1 will wrongly read such value in the abstract semantics. Token swap identifies this precise phenomenon occurring in the abstract semantics. In this particular case, token swap will avoid the deadlock because thread n_1 unblocks (as a consequence of the token swap) and then it will release the o_1 lock, thus allowing the entire program to complete without any deadlock.

We now discuss two techniques that we will adopt to limit the effect of the token swap problem: *limitation of the propagation* and *abstract name tagging*.

To avoid the *propagation* of the token swap problem, in the Petri net, as soon as 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 threads. 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 subsequent accesses. Nevertheless, subsequent accesses do not provide any new information with respect to deadlock detection because in the Creol semantics once a future has been accessed it remains available for all subsequent accesses that immediately successfully pass. In the Petri net semantics we simply model this by transforming the program by removing the accesses to a future that has been already accessed.

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

But this approach only allows to avoid the token swap for sequential identical abstract thread names. In the case of concurrent identical abstract names this is not enough. To address this problem each method invocation, generating a fresh thread name in the operational semantics, can be nondeterministically *tagged* or not. When a thread tags one of its call, all the subsequent calls will be not tagged. A tagged call to method l of object o will be denoted with $o.l?$.

The intuition behind the tagging of method calls is that the Petri net semantics will surely include a computation that tags only the calls that will be directly involved in the deadlock, in such a way that it will be not possible to have the token swap problem at least on such calls. More precisely, we

will have the guarantee that every computation in Creol that leads to a deadlock will be reproduced in the abstract Petri net semantics by a computation that tags only the calls generating threads on which the deadlocking get statements are executed. For instance, the computation leading to a deadlock discussed at the end of the Example 2.2, will be reproduced by the Petri net execution in which the first call is not tagged while the second one is. In this execution, the deadlocking get will be executed on a place labeled with $o_1.l_1@o_2.l?$ that is not swapped with the place filled by the first call that will be labeled with $o_1.l_1@o_2.l$.

4.3. Generation of abstract statement traces

We now present some preliminary formal machinery necessary to define our Petri net encoding. First of all, we need to introduce an abstract representation for method definitions. As previously discussed, we linearize the code to remove intermediary nondeterminism; more precisely, we will define a way to extract from a Creol method definition a set of possible traces composed of *abstract statements*, each of these sequence represents a possible execution.

Definition 4.3. (Statement Trace)
An *abstract statement ast* is a statement of the following form:

$$\begin{array}{rcl} ast & ::= & \mathsf{let}\, o.l \mid \mathsf{get}@(o.l, o) \mid \mathsf{get}@o.l \\ & \mid & \mathsf{let}\, o.l? \mid \mathsf{get}@(o.l?, o) \mid \mathsf{get}@o.l? \\ & \mid & \mathsf{release}(o) \mid \mathsf{grab}(o) \end{array}$$

An (abstract) *trace* is a ";" separated sequence of abstract statements. On statement traces we consider the following algebra: ";" denotes sequence concatenation and ε denotes the empty sequence, which is the identity element of ";" (i.e. given a sequence w, we assume $w; \varepsilon$ and $\varepsilon; w$ to yield w). Moreover we say that a trace w' is a suffix of trace w whenever there exists w'' such that $w = w''; w'$.

The abstract statements are of four types: (i) creation of abstract future names —tagged (i.e. $o.l?$) or non tagged (i.e. $o.l$)—, (ii) access to an abstract future by means of a get statement (there are two kinds of get, those that block the object o of the executing thread, and those that do not block the object), and (iii) the request to acquire or release the object lock.

Notice that the Creol statements if then else, suspend and claim are no longer present among the abstract statements. The conditional statements are removed as effect of code linearization while, following the Creol operational semantics, the suspend statements are abstractly represented as sequences of release–grab and, similarly, claim commands become sequences of release–get–grab.

It is worth to notice that this last transformation does not faithfully correspond to the operational semantics of claim. In fact, the claim statement follows a conditional release policy: if the claimed future is not yet available the lock is released, on the contrary, if the claimed future is available the thread continues without releasing the lock. Differently, in the abstract interpretation of claim the lock is always released. This discrepancy is not observable on those claim that are preceeded by operations accessing the same future; indeed, as explained above, we completely abstract away from such claim. On the contrary, if a claim is the first command that accesses a future, this difference

becomes observable. Nevertheless, as we will discuss in the following, this is not problematic for two main reasons. The first one is that the concrete computation that accesses the future without releasing the lock can be any way reproduced in the abstract semantics as a sequence of release–grab and access to the future. The second reason is that the additional computations that are present in the abstract semantics due to the lock release will be non problematic for our analysis: as we will show, the abstract semantics is an over-approximation of the possible concrete computations.

Abstract statement traces are obtained by applying to method definitions (and the initial thread) a sequence of syntactical transformations informally defined as follows:

Step one s_1**.** It takes Creol code t and applies data abstraction (and replacement of variables that are not defined by means of method calls) to it, obtaining abstracted code in the form of a tree.

Step two s_2**.** It turns abstracted code (a tree) into a set of sequences, representing possible executions of the method (i.e. turning all intermediate choices into a single initial choice).

Step three s_3**.** For each sequence it removes the redundant claims/gets of a future, i.e. the claims/gets that, for a given future, occur in the code after the first one. It also replaces, as done by Creol operational semantics, each claim statement by a sequence of release, get and grab statements. Similarly it also replaces each suspend statement by a release followed by a grab.

Step four s_4**.** For each sequence it returns a set of sequences (which are then put together to form an overall set of sequences): it yields all possible sequences obtained by tagging, inside it, at most one of its get. Notice that the sequence itself (without any tag) is also included among returned sequences. As we already explained, tagging is used to explicitly tag the case we are getting a future of a method call (thread) that is involved in a deadlock.

Step five s_5**.** For each sequence it applies future abstraction, replacing future parameter x of get occurrences with the pair $o.l$, i.e. called object "o" and called method "l", retrieved from the method call performed inside the definition of variable x.

Concerning sequences (used in the output of the s_2 step and in the input and output of all subsequent steps) we will use the same notation we introduced for abstract statement traces of Definition 4.3. In addition we will consider concatenation ";" of pair of sequences to be extended to sets of sequences, with the usual definition: $w; W = \{w; w' \mid w' \in W\}$ and $W; w = \{w'; w \mid w' \in W\}$, with w a sequence and W a set of sequences. At every step will consider sequences over a different set of statements: such a set will be detailed in the definition of the functions, presented below.

Formally the overall transformation, takes Creol code t, with t being such that

1. *run*$\langle t \rangle$ occurs in the initial configuration or
2. $t = \hat{t}[o/self]$ for some object o, method l and class c, with $c[(F, M)]$ and $o[c, F, \bot]$ occurring in the initial configuration and $M.l = \varsigma(self{:}T).\lambda(\vec{x}{:}\vec{T}).\hat{t}$

and yields a set of abstract statement traces of Definition 4.3. Such a transformation is called ST (Statement Traces) and is defined as the composition of all above functions:

$$\mathsf{ST} \triangleq s_5 \circ s_4 \circ s_3 \circ s_2 \circ s_1$$

In the following we will formally define the s_1, s_2, s_3, s_4 and s_5 functions.

For all trees/sequences considered in the input and output of the various steps (apart from those of Definition 4.3 obtained by applying the final step s_5 where future variables x do not occur) the following property holds: each claim/get statement uses a future parameter x that is bound by a let declaration of variable x occurring previously in the tree/sequence. This is a consequence of the syntactical restrictions we considered over initial configurations (and of the fact that each step preserves this property).

Moreover, we assume, without loss of generality, that in the initial Creol code t (to which the 5 chained steps of ST are applied) we use a different variable name in each variable let declaration occurring inside it: if this is not the case we can just change variable names in the code. As long as this is guaranteed, $\mathsf{ST}(t)$ yields the same set of abstract statement traces, no matter the specific name chosen for the variables.

Finally it is worth observing that the transformation functions (in particular the s_1 function) are implicitly parameterized on the initial configuration Creol code t is taken from. In the following, given a class c occurring in the initial configuration, we will use $O(c)$ to denote the set of all objects o of class c occurring in such a configuration. Notice that, due to the syntactical restrictions that we considered, $O(c)$ cannot be empty.

4.3.1. Step one: s_1 (Data abstraction).

This step receives, as input, Creol code t, obtained from the initial *run* thread or the body of a method, as described above. We remove most data from t, keeping only the objects names "o" (which are all free, due to the syntactical restrictions that we considered and since, in the case of methods bodies, we replaced *self* by the object name that is executing the method) and the future variables "x". Moreover, we replace all variables that are not defined by means of method calls, removing the corresponding let declarations. Thus the only let declarations that are left are future variables defined by means of method calls, which are abstracted by removing parameter data and are simply denoted by let $x{:}T = o.l$. This is a first step in obtaining the abstract statements for method invocations of Definition 4.3. Conditional (on data) branching is replaced by non-deterministic internal choice. Concerning method invocations on object variables x (e.g. being x a method parameter or a let declared variable that is initialized with the object contained inside a field/returned by a method), we replace the concrete invocation by a non-deterministic choice amongst all possible invocations, i.e. given the class type T of x, all objects in the initial configuration belonging to that class. In the definition we make use of the following auxiliary functions. Given a variable x occurring in the Creol code t given as input to s_1, $T(x)$ denotes the type T of the variable (taken from its declaration, i.e. a let declaration or a method formal parameter declaration).

Transformation s_1 outputs abstract code, i.e. a tree whose branches are labeled by statements, ranged over by *st*, and alternative branches represent abstraction of if-then-else statements. Abstract

code, ranged over by ac, belongs to the syntax

$$\begin{array}{rcl} ac & ::= & \varepsilon \mid st; ac \mid ac + ac \\ st & ::= & \mathsf{let}\, x = o.l \mid \mathsf{claim@}(x, o) \mid \mathsf{get@}(x, o) \mid \mathsf{suspend}(o) \end{array}$$

where we use "+" to denote alternative abstract codes,";" to separate a branch from the abstract code (subtree) it leads to, and ε to denote empty abstract code.

We formalize this step via a function s_1, defined as follows.

$$\begin{array}{rcl} s_1(v) & \triangleq & \varepsilon \\ s_1(\mathsf{let}\, x{:}T = v\, \mathsf{in}\, t) & \triangleq & s_1(t[v/x]) \\ s_1(\mathsf{let}\, x{:}T = (\mathsf{let}\, x'{:}T' = e'\, \mathsf{in}\, t')\, \mathsf{in}\, t) & \triangleq & s_1(\mathsf{let}\, x'{:}T' = e'\, \mathsf{in}(\mathsf{let}\, x{:}T = t'\, \mathsf{in}\, t)) \\ s_1(\mathsf{let}\, x{:}T = \mathsf{claim@}(x', o)\, \mathsf{in}\, t) & \triangleq & \mathsf{claim@}(x', o); s_1(t) \\ s_1(\mathsf{let}\, x{:}T = \mathsf{get@}(x', o)\, \mathsf{in}\, t) & \triangleq & \mathsf{get@}(x', o); s_1(t) \\ s_1(\mathsf{let}\, x{:}T = \mathsf{suspend}(o)\, \mathsf{in}\, t) & \triangleq & \mathsf{suspend}(o); s_1(t) \\ s_1(\mathsf{let}\, x{:}T = (v.l := v')\, \mathsf{in}\, t) & \triangleq & s_1(t[v'/x]) \\ s_1(\mathsf{let}\, x{:}T = v.l\, \mathsf{in}\, t) & \triangleq & s_1(t) \\ s_1(\mathsf{let}\, x{:}T = (\mathsf{if}\, v_1 = v_2\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2)\, \mathsf{in}\, t) & \triangleq & s_1(\mathsf{let}\, x{:}T = e_1\, \mathsf{in}\, t) + s_1(\mathsf{let}\, x{:}T = e_2\, \mathsf{in}\, t) \\ s_1(\mathsf{let}\, x{:}T = o.l(\vec{v})\, \mathsf{in}\, t) & \triangleq & \mathsf{let}\, x = o.l; s_1(t) \\ s_1(\mathsf{let}\, x{:}T = x'.l(\vec{v})\, \mathsf{in}\, t) & \triangleq & \Sigma_{o \in O(T(x'))}\, \mathsf{let}\, x = o.l; s_1(t[o/x']) \end{array}$$

Notice that the sum in the last defining equation has at least one argument (since class $T(x')$, due to the syntactical restrictions, must occur in the initial configuration, we have that, as observed above, $O(T(x'))$ cannot be empty). In the case such a sum has one argument only, then it is assumed to just yield such an argument; otherwise, in the case of $n \geq 1$ arguments $ac_1, \ldots, ac_n$, it yields $ac_1 + \cdots + ac_n$, where the way in which the binary $+$ is associated is not significant.

Example 4.4. Consider the extended deadlock example presented in Example 3.1 that we use as a running example.

Let t_1, t_2 and t_3 be the Creol code of methods l_1, l_2 and l_3, respectively, of class C. That is, for each $i \in \{1, 2, 3\}$, t_i is such that $m_i = \varsigma(self{:}T_i).\lambda(\vec{x_i}{:}\vec{T_i}).t_i$. Observe that $\vec{x_1}$ just contains variable x, while $\vec{x_2}$ and $\vec{x_3}$ are empty.

We have (for simplicity, since we do not have method calls on variables and function $T()$ is not used, type declarations are omitted):

- $t_1 \triangleq \mathsf{let}\, x_1 = o_2.l_2()\, \mathsf{in}\, (\mathsf{let}\, x_2 = (\mathsf{if}\, x = 1\, \mathsf{then}\, \mathsf{get@}(x_1, self)\, \mathsf{else}\, 0)\, \mathsf{in}\, x_2)$
- $t_2 \triangleq \mathsf{let}\, y_1 = o_1.l_3()\, \mathsf{in}\, (\mathsf{let}\, y_2 = \mathsf{claim@}(y_1, self)\, \mathsf{in}\, y_2)$

- $t_3 \triangleq 1$

Moreover let t be the Creol code of the *run* thread, i.e.:

- $t \triangleq \mathsf{let}\, x = o_1.l_1(1)\, \mathsf{in}\, 0$

We now consider the Creol code of methods l_1, l_2 and l_3 when executed by some object o of class C, i.e. $t_1[o/self]$, $t_2[o/self]$ and $t_3[o/self]$, respectively.

By applying step 1 to our running example, we get:

- $s_1(t_1[o/self]) = \mathsf{let}\, x_1 = o_2.l_2; (\ (\mathsf{get}@(x_1, o); \varepsilon) + \varepsilon\ )$
- $s_1(t_2[o/self]) = \mathsf{let}\, y_1 = o_1.l_3; (\mathsf{claim}@(y_1, o); \varepsilon)$
- $s_1(t_3[o/self]) = \varepsilon$
- $s_1(t) = \mathsf{let}\, x = o_1.l_1; \varepsilon$

Notice that all variables were removed except for future variables x_1, y_1 and x. □

4.3.2. Step two: s_2 (Unification of choice).

This step receives, as input, abstracted code *ac*, i.e. a tree whose branches are labeled over *st* statements. Since all choices are internal we can anticipate them and perform a single global choice at the beginning, which selects among possible execution traces.

This step outputs a set of sequences, representing execution traces, over *st* statements. Notice that such sequences are finite because method code, itself, does not include looping constructs (looping is expressed/encoded by recursion). The idea in turning the *ac* tree into a set of sequences is that, for each of them it is clear whether or not the result of a method call is read later by a claim/get or not.

We formalize this step via a function s_2, defined as follows.

$$s_2(\varepsilon) \triangleq \{\varepsilon\} \qquad s_2(ac_1 + ac_2) \triangleq s_2(ac_1) \cup s_2(ac_2)$$

$$s_2(st; ac) \triangleq st; s_2(ac)$$

Example 4.5. By applying step 2 to our running example, we get:

- $s_{1,2}(t_1[o/self]) = \{\ \mathsf{let}\, x_1 = o_2.l_2; \mathsf{get}@(x_1, o)\ ,\ \mathsf{let}\, x_1 = o_2.l_2\ \}$
- $s_{1,2}(t_2[o/self]) = \{\ \mathsf{let}\, y_1 = o_1.l_3; \mathsf{claim}@(y_1, o)\ \}$
- $s_{1,2}(t_3[o/self]) = \{\ \varepsilon\ \}$
- $s_{1,2}(t) = \{\ \mathsf{let}\, x = o_1.l_1\ \}$

where, given Creol code t', we use $s_{1,2}(t')$ to stand for $s_2(s_1(t'))$. In general in the following we will use, for the sake of conciseness, $s_{1,k}(t')$, with $k \leq 5$, to stand for $s_k(\ldots(s_1(t'))\ldots)$. □

4.3.3. Step three: s_3 (Transformation of communication).

This step receives, as input, a set of sequences over *st* statements. For each sequence it removes the redundant claims/gets of a future, i.e. the claims/gets that, for a given future, occur in the code after the first one. Since we are abstracting from data (including data returned by methods) we can just remove claims/gets occurring after the first one. This allows us to represent, in Petri nets, reading of futures as token consumption from the place representing the future. This because, by guaranteeing that future reading can occur at most once, the Petri net will not block in trying to consume a token that has already been consumed by a previous reading of the same future.

In addition this step replaces, as done by Creol operational semantics, each claim statement by a sequence of release, get and grab statements. This is done in order to move from static syntax to run-time syntax (which actually represents, at a lower level, the code behavior) and to do a further step in the direction of obtaining abstract statements of Definition 4.3. Similarly it also replaces each suspend statement by a release followed by a grab. Transformation s_3 outputs a set of strings over *run-time* statements *rst* defined by the grammar

$$rst \quad ::= \quad \mathsf{let}\, x = o.l \mid \mathsf{get@}(x, o) \mid \mathsf{get@}x \mid \mathsf{release}(o) \mid \mathsf{grab}(o)$$

We formalize this step via a function s_3^F, with F being a set of future variables x, that transforms a single *st* sequence into a *rst* sequence. The s_3^F function is defined inductively and, while F is assumed to be initially empty ($F = \emptyset$), during the induction it contains the set of variables whose futures have been already consumed. Function s_3^F is defined as follows.

$$\begin{aligned}
s_3^F(\mathsf{let}\, x = o.l; t) &\triangleq \mathsf{let}\, x = o.l; s_3^F(t) \\
s_3^F(\mathsf{claim@}(x, o); t) &\triangleq \begin{cases} \mathsf{release}(o); \mathsf{get@}x; \mathsf{grab}(o); s_3^{F \cup \{x\}}(t) & \text{if } x \notin F \\ s_3^F(t) & \text{if } x \in F \end{cases} \\
s_3^F(\mathsf{get@}(x, o); t) &\triangleq \begin{cases} \mathsf{get@}(x, o); s_3^{F \cup \{x\}}(t) & \text{if } x \notin F \\ s_3^F(t) & \text{if } x \in F \end{cases} \\
s_3^F(\mathsf{suspend}(o); t) &\triangleq \mathsf{release}(o); \mathsf{grab}(o); s_3^F(t) \\
s_3^F(\varepsilon) &\triangleq \varepsilon
\end{aligned}$$

The s_3 transformation is obtained by lifting s_3^F to work over sets of strings, so that it can be chained with the s_2 transformation.

$$s_3^F(W) \quad \triangleq \quad \{s_3^F(t) \mid t \in W\}$$

The s_3 transformation is therefore defined as $s_3^\emptyset(W)$, where W is the set of *st* sequences outputed by s_2. For the sake of simplicity, we will just write s_3 to stand for $s_3^\emptyset$.

Example 4.6. By applying step 3 to our running example, we get:

- $s_{1,3}(t_1[o/self]) = \{\ \mathsf{let}\, x_1 = o_2.l_2; \mathsf{get@}(x_1, o)\ ,\ \mathsf{let}\, x_1 = o_2.l_2\ \}$

- $s_{1,3}(t_2[o/self]) = \{$ let $y_1 = o_1.l_3$; release(o); get@y_1; grab(o) $\}$
- $s_{1,3}(t_3[o/self]) = \{ \varepsilon \}$
- $s_{1,3}(t) = \{$ let $x = o_1.l_1$ $\}$

The claim statement was replaced in the second set. Notice that, this is the only change in the traces because all of them only have at most one get/claim statement. □

4.3.4. Step four: s_4 (Adding tags).

This step receives, as input, a set of sequences over *rst* statements. For each sequence it produces a set of sequences: all possible sequences obtained by tagging, inside it, *at most* one of its get with the "?" symbol. This means that the sequence itself (without any tag) is also included among the produced sequences. The idea is that, if the input code (an execution sequence) is involved in a deadlock by means of one of its get, i.e. it is blocked on such a get, then that it is obviously not possible that, in the context of the same deadlock, it is blocked also on other get. Of course it can be that it is not involved at all in the deadlock, i.e. it has no tag. More formally, given an extended deadlock reached by a Creol program, we have that each thread n belonging to the set N of Definition 3.5 may block on a grab or on a get of a future/thread n'. In the latter case, in order to detect such a deadlock in the Petri net, we consider a tagged abstract identifier in the form $c@o.l?$ for the thread n'. Thus we have to consider a possible sequence for the code of n where get is correspondingly tagged. Of course, whenever we tag a get, we have also to tag with the "?" symbol the let statement binding it, which performs the $o.l$ method call (generating the thread with the $c@o.l?$ abstract identifier).

Transformation s_4 outputs a set of strings over a tagged run-time statements *mst* defined by the grammar

$$\textit{mst} \quad ::= \quad \textit{rst} \mid \text{let } x = o.l? \mid \text{get}(x?, o) \mid \text{get@}x?$$

We formalize this step via a function s_4, which transforms a single *rst* sequence into a set of *mst* sequences, defined as follows.

$$\begin{aligned} s_4(t) \triangleq \{t\} \cup \{t_1; \text{let } x = o'.l?; t_2; \text{get}(x?, o); t_3 \mid t = t_1; \text{let } x = o'.l; t_2; \text{get}(x, o); t_3\} \\ \cup \{t_1; \text{let } x = o'.l?; t_2; \text{get@}x?; t_3 \mid t = t_1; \text{let } x = o'.l; t_2; \text{get@}x; t_3\} \end{aligned}$$

This definition is based on the fact (discussed before) that, in traces produced by previous steps: for each get statement using a future parameter x there is one and only one let declaration of x occurring previously in the sequence.

The s_4 transformation is obtained by lifting function s_4 to work over sets of strings, so that it can be chained with the s_3 transformation.

$$s_4(W) \triangleq \cup_{t \in W} s_4(t)$$

where W is the set of *rst* sequences outputed by s_3.

Notice that the sets of sequences produced, for each input sequence, by the s_4 function, are put together to form an overall set of sequences, which is the output of the s_4 transformation.

Example 4.7. By applying step 4 to our running example, we get:

- $s_{1,4}(t_1[o/self]) =$
  $\{\ \mathsf{let}\, x_1 = o_2.l_2; \mathsf{get@}(x_1, o)\ ,\ \mathsf{let}\, x_1 = o_2.l_2?; \mathsf{get@}(x_1?, o)\ ,\ \mathsf{let}\, x_1 = o_2.l_2\ \}$
- $s_{1,4}(t_2[o/self]) =$
  $\{\ \mathsf{let}\, y_1 = o_1.l_3; \mathsf{release}(o); \mathsf{get@}y_1; \mathsf{grab}(o)\ ,$
  $\mathsf{let}\, y_1 = o_1.l_3?; \mathsf{release}(o); \mathsf{get@}y_1?; \mathsf{grab}(o)\ \}$
- $s_{1,4}(t_3[o/self]) = \{\ \varepsilon\ \}$
- $s_{1,4}(t) = \{\ \mathsf{let}\, x = o_1.l_1\ \}$

□

4.3.5. Step five: s_5 (Future abstraction).

This step receives, as input, a set of sequences over *mst* statements. In this last step we apply future abstraction, thus removing future variables. Up until now, they were preserved because they were necessary to define the function s_4. More precisely, we replace future parameter x of get occurrences with the pair $o.l$, i.e. called object "o" and called method "l", retrieved from the method call performed inside the definition of variable x. Transformation s_5 outputs a set of statement traces (see Definition 4.3), i.e. sequences over abstract statements *ast*.

We formalize this step via a function s_5 that transforms a single *mst* sequence into an *ast* sequence. Function s_5 is defined as follows.

$$
\begin{aligned}
s_5(\mathsf{let}\, x = o.l^*; t) &\triangleq \mathsf{let}\, o.l^*; s_5(t)[o.l/x] \\
s_5(\mathit{mst}; t) &\triangleq \mathit{mst}; s_5(t) \qquad \nexists x, o, l : \mathit{mst} \in \{\mathsf{let}\, x = o.l, \mathsf{let}\, x = o.l?\}
\end{aligned}
$$

where "$*$" is a meta-variable that can be either the empty string or "?" (used to denote that "$\mathsf{let}\, x = o.l$" can be tagged or not) and "$t[o.l/x]$" is the result of replacing "x" by "$o.l$" in "t".

The s_5 transformation is obtained by lifting function s_5 to work over sets of sequences, so that it can be chained with the s_4 transformation.

$$s_5(W) \triangleq \{s_5(t) \mid t \in W\}$$

Example 4.8. By applying step 5 to our running example, we get:

- $s_{1,5}(t_1[o/self]) = \mathsf{ST}(t_1[o/self]) =$
  $\{\ \mathsf{let}\, o_2.l_2; \mathsf{get@}(o_2.l_2, o)\ ,\ \mathsf{let}\, o_2.l_2?; \mathsf{get@}(o_2.l_2?, o)\ ,\ \mathsf{let}\, o_2.l_2\ \}$
- $s_{1,5}(t_2[o/self]) = \mathsf{ST}(t_2[o/self]) =$
  $\{\ \mathsf{let}\, o_1.l_3; \mathsf{release}(o); \mathsf{get@}o_1.l_3; \mathsf{grab}(o)\ ,$
  $\mathsf{let}\, o_1.l_3?; \mathsf{release}(o); \mathsf{get@}o_1.l_3?; \mathsf{grab}(o)\ \}$
- $s_{1,5}(t_3[o/self]) = \mathsf{ST}(t_3[o/self]) = \{\ \varepsilon\ \}$
- $s_{1,5}(t) = \mathsf{ST}(t) = \{\ \mathsf{let}\, o_1.l_1\ \}$

□

Figure 1. Places for objects and abstract threads.

4.4. Petri net construction for creol programs

We are now in a position to present the definition of the Petri net associated to an initial configuration. We first discuss the two kinds of places that we will consider in the Petri net (see Fig. 1):

Object 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.

Abstract Threads. Places identifying a particular thread in execution or the future resulting from the execution of a thread. These places are labeled with $c@c'\langle t\rangle$ where $c@c'$ is an abstract label with c' identifying the method (or initial *run*) in execution, c its caller, and t being an abstract statement trace (as defined in Definition 4.3). A token in this place represents one instance of such a thread in execution or a future. In the latter case the place is of the kind $c@c'\langle\varepsilon\rangle$ (where the trace still to be executed is empty), simply denoted by $c'@c\langle\rangle$. In this case, the token is consumed if the future is accessed.

We now discuss the transitions that will be considered in the Petri net semantics.

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 with *run*. This is the main thread in the program, thus it is not called by another thread, does not belong to any object, nor class. Due to this lack of information, i.e. no object and method of the caller and no called object and method, we will use as abstract name for the initial thread the tuple *run*@*run*. As discussed above, we will define a way to extract from the thread code a set of abstract statement traces: let $t_1, \ldots, t_n$

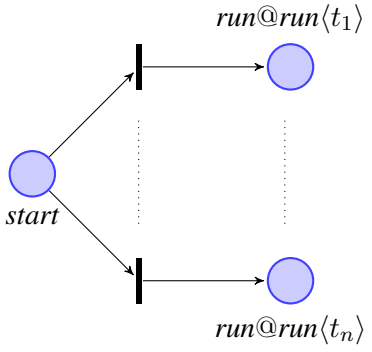
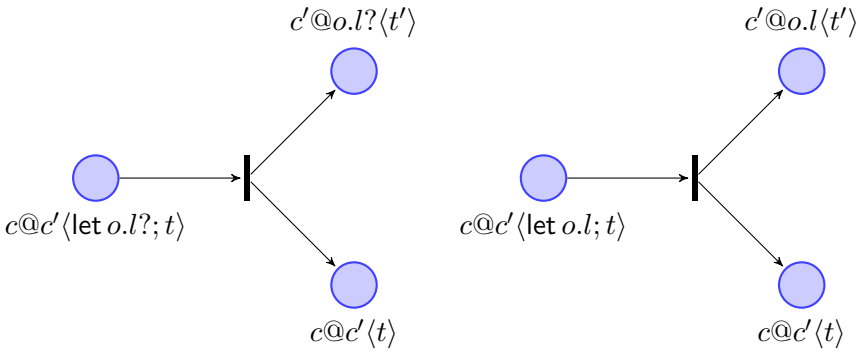


Figure 2. Transition from the initial place *start*: t_i are the abstract traces of the initial thread *run*.

be such traces. Hence we can have different abstract representations in the Petri net for the initial thread: $run@run\langle t_1\rangle, \ldots, run@run\langle t_n\rangle$. To activate nondeterministically one of them, we consider an auxiliary place *start*, that will contain a token in the initial configuration, and n alternative transitions that move such token in one of these n places (see Fig. 2).

Method Calls. We present the Petri net transitions for a method call in Fig. 3. Depending on whether the result of the call will be assumed to be part of a deadlock or not, the created thread is tagged (see Fig. 3.a, notice the symbol "?") or it is not (see Fig. 3.b). In the first case, the new thread will be abstractly named with $c'@o.l?$ (where c' represents the caller) while in the second case with $c'@o.l$. In both cases, the new thread will execute one of the traces t' extracted from the definition of method l in the class of object o. More precisely, the Petri net will include two transitions, like those reported in Fig. 3, for each trace t' obtained as abstraction of the Creol code of the l method definition.



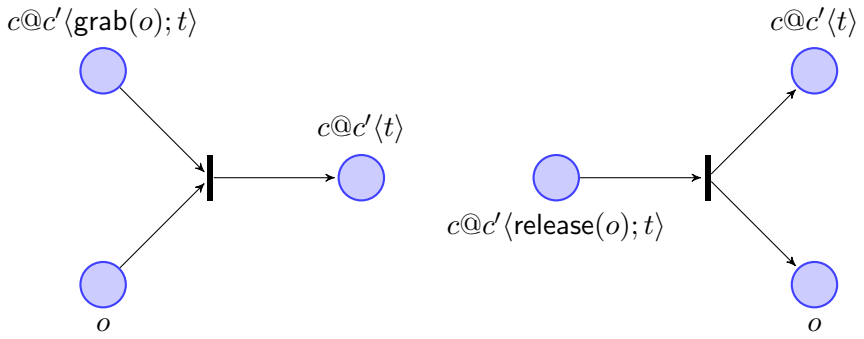
(a) tagged calls (b) calls without tag

Figure 3. 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. 4)

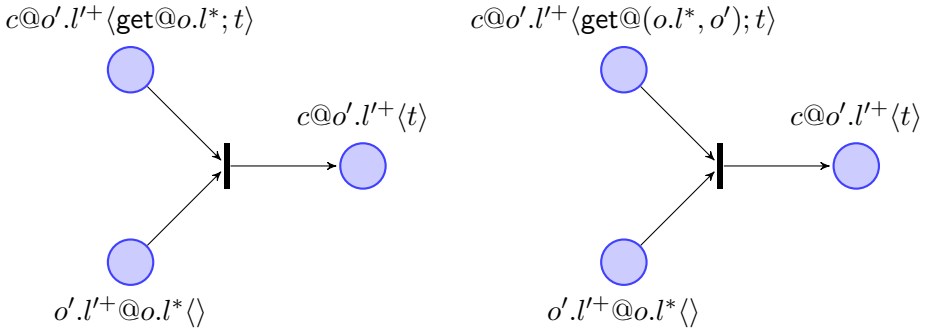
Access to Results. We present the Petri net transitions for accessing the result of a method call in Fig. 5. According to the abstract statements syntax, we have four distinct primitives: $\mathsf{get}@(o.l, o)$ and $\mathsf{get}@(o.l?, o)$, for a get executed by a thread holding the lock of its object o, $\mathsf{get}@o.l$ and $\mathsf{get}@o.l?$, for threads that released their lock and try to access the future before re-acquiring it. In the Figure we unify pairs of cases by adopting the following meta-notation: "$o.l^+$" or "$o.l^*$" denote that $o.l$ can be tagged or not; formally, "$+$" and "$*$" are meta-variables that can both be either the empty string or "$?$".

In both the depicted transitions, the access to the returned value is represented by the consumption of one token in the place $o'.l'^+@o.l^+\langle\rangle$: the fact that the trace to be executed is empty indicates that method execution has completed, hence the return value is available. Moreover, notice that the token in



(a) grabbing the lock (b) releasing the lock

Figure 4. Transitions for lock handling.



(a) get by a thread not holding the lock (b) get by a thread holding the lock

Figure 5. Translation of get of a result.

such place (representing the availability of the return value) is consumed by the transition: in this way the return value is no longer available for subsequent access. Nevertheless, has already commented, our abstract traces will remove subsequent accesses, hence avoiding this possible problem.

We are now ready to report the full definition of our Petri net construction. To simplify the notation, we denote markings, i.e. functions from places to natural numbers, simply by listing the places with a non zero associated number. This is sufficient because we simply consider the natural numbers 0 and 1 in all considered markings. Moreover, as above, we make use of the meta-variables "$+$" and "$*$" that can both be either the empty string or "?".

Definition 4.9. Given an initial configuration

$$C_0 = c_1[(F_1, M_1)] \,\|\, \ldots \,\|\, c_m[(F_m, M_m)] \,\|\, o_1[c_{k_1}, F_{k_1}, \bot] \,\|\, \ldots \,\|\, o_n[c_{k_n}, F_{k_n}, \bot] \,\|\, \mathit{run}\langle \bar{t}\rangle$$

the Petri net associated to C_0 is defined by $P_{C_0} = \langle P, T, m_0\rangle$ where

- $P = \{start, o_1, \ldots, o_n\} \cup \mathcal{S}$ with $\mathcal{S}$ being the minimal set of places $c@c'\langle t\rangle$ satisfying the following conditions:
  - $c@c'$ is of the form *run*@*run*, *run*@$o_i.l$ or $o_j.l'^{+}@o_i.l^*$ with l and l' methods of the classes c_{k_i} and c_{k_j}, respectively.
  - if $c@c' =$ *run*@*run* then t is a suffix of one of the traces in $\mathsf{ST}(\bar{t})$
  - if $c@c'$ is $c@o_i.l$ or $c@o_i.l?$ then t is a suffix of a trace in the set

    $$\mathsf{grab}(o_i); \mathsf{ST}(\hat{t}[o_i/self]); \mathsf{release}(o_i)$$

    where $\hat{t}$ is the code of the method l with "*self*" denoting the self object, namely, being c_{k_i} the class of o_i, we have $M_{k_i}.l = \varsigma(self{:}T).\lambda(\vec{x}{:}\vec{T}).\hat{t}$
- T is the minimal set including the following transitions.

  T-1 For each $t' \in \mathsf{ST}(\bar{t})$ there is $\mathsf{t} \in T$ such that
  ${}^\bullet\mathsf{t} = \{start\}$ and $\mathsf{t}^\bullet = \{run@run\langle t'\rangle\}$ (see Fig. 2).

  T-2 For each $c@c'\langle \mathsf{let}\ o_i.l^*; t\rangle \in P$ and $t' \in \mathsf{grab}(o_i); \mathsf{ST}(\hat{t}[o_i/self]); \mathsf{release}(o_i)$, where $M_{k_i}.l = \varsigma(self{:}T).\lambda(\vec{x}{:}\vec{T}).\hat{t}$, there is $\mathsf{t} \in T$ such that
  ${}^\bullet\mathsf{t} = \{c@c'\langle \mathsf{let}\ o_i.l^*; t\rangle\}$ and $\mathsf{t}^\bullet = \{c@c'\langle t\rangle, c'@o_i.l^*\langle t'\rangle\}$ (see Fig. 3.a for the case of a tagged method call and Fig. 3.b for a non tagged one).

  T-3 For each $c@c'\langle \mathsf{grab}(o); t\rangle \in P$ there is $\mathsf{t} \in T$ such that
  ${}^\bullet\mathsf{t} = \{c@c'\langle \mathsf{grab}(o); t\rangle, o\}$ and $\mathsf{t}^\bullet = \{c@c'\langle t\rangle\}$ (see Fig. 4.a).

  T-4 For each $c@c'\langle \mathsf{release}(o); t\rangle \in P$ there is $\mathsf{t} \in T$ such that
  ${}^\bullet\mathsf{t} = \{c@c'\langle \mathsf{release}(o); t\rangle\}$ and $\mathsf{t}^\bullet = \{c@c'\langle t\rangle, o\}$ (see Fig. 4.b).

  T-5 For each $c@o'.l'^{+}\langle \mathsf{get}@o.l^*; t\rangle \in P$ there is $\mathsf{t} \in T$ such that
  ${}^\bullet\mathsf{t} = \{c@o'.l'^{+}\langle \mathsf{get}@o.l^*; t\rangle, o'.l'^{+}@o.l^*\langle\rangle\}$ and $\mathsf{t}^\bullet = \{c@o'.l'^{+}\langle t\rangle\}$ (see Fig. 5.a).

  T-6 For each $c@o'.l'^{+}\langle \mathsf{get}@(o.l^*, o'); t\rangle \in P$ there is $\mathsf{t} \in T$ such that
  ${}^\bullet\mathsf{t} = \{c@o'.l'^{+}\langle \mathsf{get}@(o.l^*, o'); t\rangle, o'.l'^{+}@o.l^*\langle\rangle\}$ and $\mathsf{t}^\bullet = \{c@o'.l'^{+}\langle t\rangle\}$ (see Fig. 5.b).
- The initial marking m_0 is defined by $\{start, o_1, \ldots, o_n\}$.

Notice that, for methods called by the *run* thread we just consider places in the form *run*@$o_i.l$, i.e. whose thread abstract identifier is not tagged, because the *run* thread code $\bar{t}$ cannot include get or claim statements (so, when evaluating $\mathsf{ST}(\bar{t})$ step s_4 does not produce any tagged statement). Moreover, notice that Definition 4.9 is well-defined, i.e. it produces a finite number of places and transitions. This because the number of objects and methods in C_0 is finite and, for any Creol code t, $\mathsf{ST}(t)$ produces a finite amount of (finite) traces.

Example 4.10. In Fig. 6 we show the Petri net obtained from the Creol program considered in Example 3.1 (case of extended deadlock), i.e. our running example. For clarity, and due to space limitations, we omit places and transitions that will be never involved in computations starting from the initial marking $\{start, o_1, o_2\}$. Moreover, for each method, we depict places and transitions for only one of

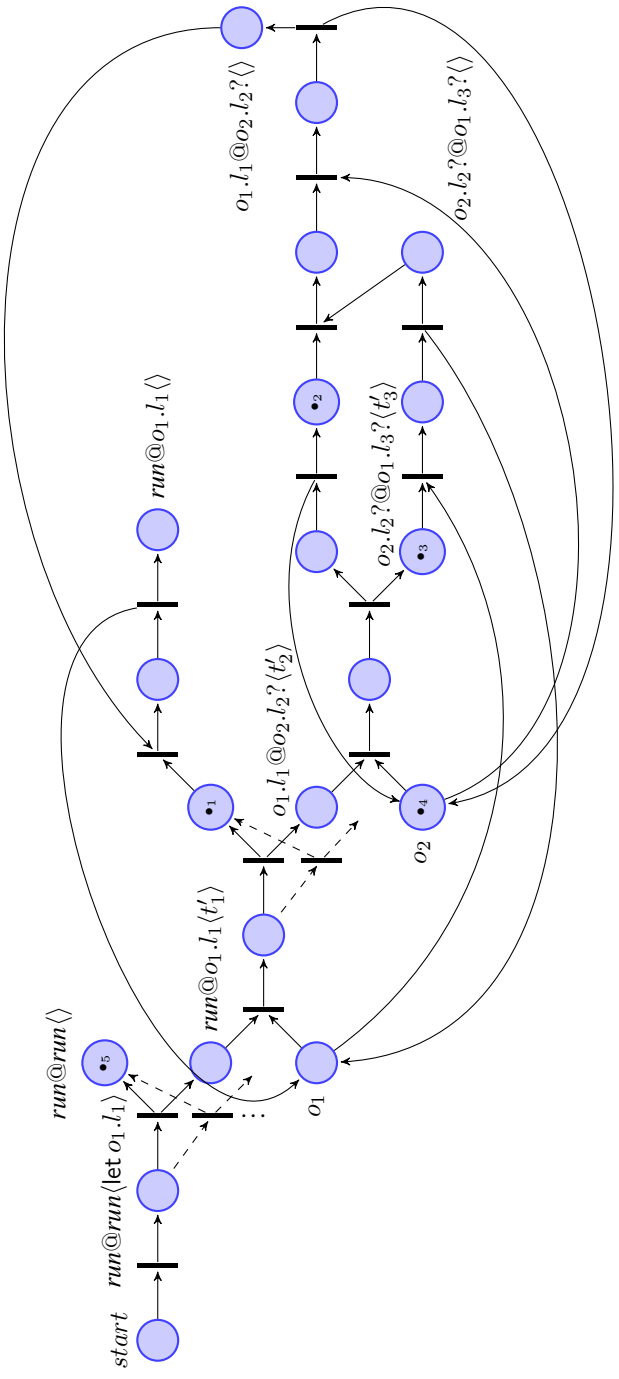


Figure 6. Translation of the extended deadlock example.

the possible corresponding abstract traces, as the places and transitions for the other ones are similar. Namely, the methods l_1, l_2 and l_3, that are executed by objects o_1, o_2 and o_1, respectively, generate the sets of abstract traces $W_1 = \mathsf{grab}(o_1); \mathsf{ST}(t_1[o_1/self]); \mathsf{release}(o_1)$, $W_2 = \mathsf{grab}(o_2); \mathsf{ST}(t_2[o_2/self]); \mathsf{release}(o_2)$ and $W_3 = \mathsf{grab}(o_1); \mathsf{ST}(t_3[o_1/self]); \mathsf{release}(o_1)$, with t_i being the code presented in Example 4.4 for $i \in \{1, 2, 3\}$. We just consider one abstract trace t'_1, t'_2 and t'_3 for each set of abstract traces W_1, W_2 and W_3, respectively (the last one being actually a singleton), that is

- $t'_1 \triangleq \mathsf{grab}(o_1); \mathsf{let}\, o_2.l_2?; \mathsf{get@}(o_2.l_2?, o_1); \mathsf{release}(o_1)$
- $t'_2 \triangleq \mathsf{grab}(o_2); \mathsf{let}\, o_1.l_3?; \mathsf{release}(o_2); \mathsf{get@}o_1.l_3?; \mathsf{grab}(o_2); \mathsf{release}(o_2)$
- $t'_3 \triangleq \mathsf{grab}(o_1); \mathsf{release}(o_1)$

In the picture we use dashed lines and dots to indicate the presence of additional places and transitions (similar to those that we depict) corresponding to the other alternative threads.

Notice that in the Figure we show only the labels for the places corresponding with the beginning of the t'_i trace execution, i.e. labeling a place that is in the postset of a *T-2* transition representing a method call, and for the places corresponding with the end of the t'_i trace execution, i.e. labeling a place that represents a future, for $i \in \{1, 2, 3\}$.

The initial marking of the Petri net consists of one token in the $start$ place, plus one token in both the lock places o_1 and o_2. In the picture we consider an example of a reachable marking represented by the tokens $\bullet_1$, $\bullet_2$, $\bullet_3$, $\bullet_4$ and $\bullet_5$. We associated a number with each token just for the aim of singling them in the following. This marking is reachable by performing the following transitions. The first transition moves the token from the *start* place into the initial place of the *run* thread, i.e. the *run*@*run*$\langle \mathsf{let}\, o_1.l_1 \rangle$ place. Since *run* makes a $o_1.l_1$ call, a transition can be performed that puts: one token into the *run*@$o_1.l_1\langle t'_1 \rangle$ place and one token in the place *run*@*run*$\langle\rangle$ (the place with the $\bullet_5$ token)). The transition from *run*@$o_1.l_1\langle t'_1 \rangle$ (representing the initial grab of the t'_1 trace) removes the token from the o_1 place and the subsequent transition (one of the multiple transitions, all representing the $o_2.l_2$ call, the Petri net can perform) puts: one token into the $o_1.l_1@o_2.l_2?\langle t'_2 \rangle$ place and one token in the place with the $\bullet_1$ token. In turn, the transition from $o_1.l_1@o_2.l_2?\langle t'_2 \rangle$ (representing the initial grab of the t'_2 trace) removes the token from the o_2 place and the subsequent transition (representing the $o_1.l_3$ call) puts: one token into the $o_2.l_2?@o_1.l_3?\langle t'_3 \rangle$ place, i.e. the place with the $\bullet_3$ token, and one token in a place from which a transition (representing the first release of the t'_2 trace) is performed that puts a token into the o_2 place (the $\bullet_4$ token) and a token into the place with the $\bullet_2$ token. □

4.5. Soundness of the Petri net construction

The Petri net P_{C_0} is a sound over-approximation of the possible behaviors of the initial configuration C_0. The proof of this result is rather technical hence we have reported the details in Appendix A. Here we only present the main theorem formalizing this soundness result.

Consider a Creol program with initial configuration C_0. The idea is to associate to each execution $C_0 \to \ldots \to C_k$ a set of corresponding reachable markings in the Petri net P_{C_0}. We denote this set of markings with $M(C_0 \to \ldots \to C_k)$.[3] Notice that this set of reachable markings is computed

[3]The formal definition of the mapping $M(C_0 \to \ldots \to C_k)$ is in Appendix A.

considering the entire computation $C_0 \to \ldots \to C_k$ and not only the reached configuration C_k: this is a consequence of the fact that C_k contains partially executed threads, and the Petri net representation of such threads requires knowledge of previous computation steps. For instance, if a partially executed thread in C_k accesses the future n, in the corresponding Petri net marking we have to replace n with the pair $o.l$ where o and l are the object and method invoked when the future n was created.

Intuitively, the mapping function $M(C_0 \to \ldots \to C_k)$ returns a set of markings having the following property: one token is present for each active thread in C_k (threads whose return value was not yet read) and one token is present for each available object lock in C_k. It is necessary to consider a set of markings instead of a unique marking representing the configuration C_k, because several aspects of the Creol program are abstracted away in the Petri net semantics, like the selected branch in an if-then-else construct or the non deterministic application of the tag "?" to the abstract thread identifiers $c@c'$. These abstractions introduce nondeterminism which is not present in the Creol program, hence each active thread in C_k could have different possible representations in the Petri net (e.g. representations that differ only in the branches selected in the if-then-else constructs or in the way the tags "?" are applied).

The main theorem concerning the correctness of the Petri net encoding proved in Appendix A, where we also formally define the mapping function $M(C_0 \to \ldots \to C_k)$, is as follows.

Theorem 4.11. (Marking Reachability)
Let $C_0 \to \cdots \to C_k$, with $k \geq 0$, be an execution in the operational semantics of the initial configuration C_0. Each marking in $M(C_0 \to \ldots \to C_k)$ is reachable from the initial marking of P_{C_0}.

5. Deadlock freedom

This section presents a procedure for detecting deadlock in Creol programs. We details out a technique for detecting extended deadlocks because they subsume the classical ones (Proposition 3.6) but, as we will discuss in Section 5.3, it can be easily modified to deal with classical deadlock.

In the previous Sections we have presented that for every computation $C_0 \to \cdots \to C_k$ of a Creol program, there exists a set of markings $M(C_0 \to \ldots \to C_k)$ that are reachable in the corresponding Petri net P_{C_0}. In this Section we present a characterization of extended deadlock configurations for Creol programs in terms of Petri net markings, that we call *extended deadlock markings*. Extended deadlocks are preserved by the translation function $M(C_0 \to \ldots \to C_k)$, in the sense that if C_k is an extended deadlock then $M(C_0 \to \ldots \to C_k)$ contains some extended deadlock markings. This result entails a procedure for proving deadlock freedom in the Creol program: if no extended deadlock marking is reachable in the Petri net, then the program is guaranteed to be deadlock free. This condition can be algorithmically checked by means of reachability analysis on the Petri net.

5.1. Extended deadlock marking

Extended deadlock markings essentially correspond to that of Creol extended deadlock configurations given in Definition 3.5, where the set N of (waiting) threads is expressed as a set D of places (abstractly representing such threads) and the abstract identifier of threads that are waited for, from a thread in N, is tagged by "?".

The definition of extended deadlock marking follows. In the definition we consider an abstract trace t to be *?-free* if it does not include "?" tags in any of the commands included in its sequence. Similarly, $c = o.l^*$ is considered to be *?-free* if the meta-variable $*$ is the empty string (and not "?"). Finally, a place p is considered to be *?-free* if its label does not include any "?" tag, i.e. it is either an object lock place labeled with o or it is a thread place labeled with $c@c'\langle t\rangle$ such that t is *?-free* and both c and c' are *?-free*.

Definition 5.1. (Extended Deadlock Marking)
A marking m in a Petri net is an *extended deadlock marking* iff there exists a set of places D such that:

1. For each $p \in D$ we have: $m(p) \geq 1$, p is in the form $c@c'\langle t\rangle$, with t being one of "$\mathsf{get}@(o''.l''?, o'); t'$", "$\mathsf{get}@o''.l''?; t'$" or "$\mathsf{grab}(o'); t'$", and p satisfies all the following conditions
   (a) if $c = o.l?$ then there is $p' \in D$ 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 $c' = o'.l'?$ then there is $p' \in D$ 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' is ?-free and there is $p' \in D$ in the form $c@c'\langle \mathsf{get}@(o'''.l'''?, o'); t''\rangle$.
2. For each $p \notin D$ we have: $m(p) \geq 1$ implies
   (a) p is ?-free or
   (b) p is in the form $c@c'\langle t\rangle$, with c', t being ?-free and $c = o.l?$ for some o,l such that there is $p' \in D$ in the form $c''@o.l?\langle t'\rangle$.

Notice that, in order for an extended deadlock marking to be reachable from the initial marking it must hold that: if D includes a place $c@c'\langle t\rangle$ with t being one of "$\mathsf{get}@(o''.l''?, o'); t'$" or "$\mathsf{get}@o''.l''?; t'$", then D must also include a place $c'@o''.l''?\langle t''\rangle$ for some t'' (resembling the condition in Definition 3.5, i.e. a thread that is waited for by a thread in N is also in N). This because for any abstract trace produced by the ST function, it holds that any occurrence of "$\mathsf{get}@(o''.l''?, o'); t'$" or "$\mathsf{get}@o''.l''?; t'$" is always preceded by a corresponding $\mathsf{let}\ o''.l''?$ (due to transformation s_4). This let generates a token in the thread $c'@o''.l''?$ reaching some place $c'@o''.l''?\langle t''\rangle$, but this last place must belong to D because by item 2 the places $c@c'\langle t\rangle \notin P$ whose c' is not *?-free* are required to have 0 tokens.

In general, therefore, the conditions imposed on the set of places D, correspond to constraints Definition 3.5 adapted to the Petri net context, where we also require that the abstract identifier of threads that are waited for, from a place in D, is tagged by "?". In relation to this requirement, here, extra conditions (notably 1a, 1b and 2b) are considered in order to ensure the consistency between the tagged get statements and the tagged abstract thread identifiers. Item 2 allows for non-zero tokens to occur in places that represent threads that do not belong to the deadlock (not in N of Definition 3.5). These places are of two kinds: either they are completely *?-free* and totally unrelated to the deadlock (condition 2a) or they are the abstract representation of threads that were called by a thread that belongs to the deadlock (condition 2b). In both cases, thread places $c@c'\langle t\rangle$ cannot create a token

swap, because both c' and t are required to be *?-free*, thus they, respectively, cannot identify a tagged thread/future and they cannot include a tagged get: this guarantees that in the abstract representation no thread that is inside the deadlock (in N of Definition 3.5) can be confused for a thread that is outside the deadlock (not in N of Definition 3.5) or vice versa. In practice, by imposing *?-free* constraints in item 2, we avoid the token swap problem: no token is allowed in places $c@c'\langle t\rangle$ that are not in D and have some tag "?" in c' or t. Not allowing tokens in these places guarantees that token swap is not possible.

Example 5.2. We show a reachable extended deadlock marking for the Petri net constructed in Example 4.10. We consider the marking composed of the tokens $\bullet_1$, $\bullet_2$, $\bullet_3$, $\bullet_4$ and $\bullet_5$ shown in the Petri net depicted in Figure 6. More formally, the marking m we consider has 1 token for each of the following places:

1. $\mathit{run}@o_1.l_1\langle \mathsf{get}@(o_2.l_2?, o_1); \mathsf{release}(o_1)\rangle$
2. $o_1.l_1@o_2.l_2?\langle \mathsf{get}@o_1.l_3?; \mathsf{grab}(o_2); \mathsf{release}(o_2)\rangle$
3. $o_2.l_2?@o_1.l_3?\langle t'_3\rangle$
4. o_2
5. $\mathit{run}@\mathit{run}\langle\rangle$

with t'_3 being that of Example 4.10. We have already commented, at the end of the presentation of Example 4.10, that this marking is reachable in the Petri net.

We now show that this (reachable) marking m is an extended deadlock marking. We just take D of Definition 5.1 to be the set of places 1, 2 and 3 above (i.e. those containing tokens $\bullet_1$, $\bullet_2$ and $\bullet_3$). It is immediate to observe that all constraints in item 1 of Definition 5.1 for places in D are satisified: they all have some token, are in the required format, and satisfy all the constraints $1a$–$1c$ (constraint $1a$ applies to place 3, constraint $1b$ applies to place 2 and 3 and constraint $1c$ applies to place 3). Moreover, constraints in item 2 of Definition 5.1 for places not in D having some token are also satisfied: both places o_2 (with $\bullet_4$) and $\mathit{run}@\mathit{run}\langle\rangle$ (with $\bullet_5$) are *?-free*. □

5.2. Deadlock analysis

We now prove that extended deadlocks are preserved by the mapping function $M(C_0 \to \ldots \to C_k)$: given a Creol program execution $\alpha = C_0 \to \ldots \to C_k$ such that the configuration C_k is an extended deadlock (Definition 3.5), we have that the set of markings produced by the mapping function $M(\alpha)$ includes an extended deadlock marking (Definition 5.1) of P_{C_0}. This, in combination with the marking reachability theorem of the previous section (Theorem 4.11), will allow us to prove the main theorem (Theorem 5.4) about extended deadlock marking reachability for deadlocking Creol programs. Finally, we will show (Theorem 5.6) decidability for the extended deadlock marking reachability problem for P_{C_0}. The proof of the following Lemma is in Appendix B because it is based on definitions and results reported in Appendix A (where we also formally define the mapping function $M(C_0 \to \ldots \to C_k)$).

Lemma 5.3. Let C_0 be an initial configuration and let $\alpha = C_0 \to \ldots \to C_k$ be an execution in the operational semantics. Let P_{C_0} be the Petri net associated to C_0. If C_k is an extended deadlock then $M(\alpha)$ includes an extended deadlock marking of P_{C_0}.

We are now in a position to present the main theorem about extended deadlock marking reachability for deadlocking Creol programs. We say that a Creol program has a reachable extended deadlock configuration if, assumed C_0 to be the initial configuration of the program, there exists an execution $C_0 \to \cdots \to C_k$, with $k \geq 0$, such that C_k is an extended deadlock configuration.

Theorem 5.4. (Extended Deadlock Marking Reachability)
Given a Creol program, if it has a reachable extended deadlock configuration, then the corresponding Petri net has a reachable extended deadlock marking.

Proof: Let $C_0 \to \cdots \to C_k$, with $k \geq 0$, be an execution in the operational semantics of the initial configuration C_0 such that C_k is an extended deadlock. We just apply Theorem 4.11 to such an execution and we, then, use Lemma 5.3 on the hypothesis that C_k is an extended deadlock. □

We finally 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 that we have recalled in Section 4.1. In fact, the following lemma shows how to reduce the problem of checking the existence of a reachable extended deadlock marking to a finite amount of instances of the (decidable) target marking reachability problem.

Lemma 5.5. Given a Petri net $\mathcal{P}$, there exists a finite set of target marking denotations (inf_1, sup_1), $\ldots, (inf_n, sup_n)$ such that, for all markings m of $\mathcal{P}$ it holds: m is an extended deadlock marking if and only if there exists $i \in \{1 \ldots n\}$ such that m satisfies (inf_i, sup_i).

Proof: By Definition 5.1 we have that m is an extended deadlock marking if and only if there exists a set of places D such that conditions 1 and 2 of the definition are satisfied. We now show how to define a target marking denotation (inf_D, sup_D) as a function of D. It is sufficient to consider $inf_D(p) = 1$ and $sup_D(p) = \infty$ for every $p \in D$, $inf_D(r) = 0$ and $sup_D(r) = \infty$ for every $r \notin D$ that satisfies 2.(a) or 2.(b), and $inf_D(s) = 0$ and $sup_D(s) = 0$ for the remaining places s. It is immediate to see that, given a set of places D satisfying condition 1 of Definition 5.1 we have that, for all markings m of $\mathcal{P}$: m satifies the conditions 1 and 2 of Definition 5.1 if and only if m satisfies (inf_D, sup_D).

We conclude by observing that the possible distinct sets of places D that satisfy condition 1 of Definition 5.1 are obviously finite (it follows directly from the finiteness of the Petri net). Let $\{D_1, \ldots, D_n\}$ be such finitely many sets of places. The thesis holds by considering the set of target marking denotations $(inf_{D_1}, sup_{D_1}), \ldots, (inf_{D_n}, sup_{D_n})$. □

Theorem 5.6. Let C_0 be an initial configuration and let P_{C_0} be the Petri net associated to C_0. It is decidable whether in P_{C_0} there exists a reachable extended deadlock marking.

Proof: In the light of Lemma 5.5 deciding the reachability of an extended deadlock marking can be reduced to deciding the reachability of one among a finite set of target marking denotations. The thesis hence follows from the decidability of target marking reachability. □

5.3. Detecting classical deadlocks

We now discuss how the technique previously described to proof extended deadlock freedom, can be modified to deal with *classical deadlocks*. In fact, it could be the case that a Creol program has extended deadlocks but it has no classical ones (see, for instance, Example 2.1 where the thread in execution within o_2, involved in the deadlock, does not hold the object lock). Hence, one could be interested in proving this specific property, i.e. classical deadlock free but not extended deadlock free for his own Creol program.

The first modification to the proposed technique is in the definition of extended deadlock marking. Differently from extended deadlocks, in classical deadlock the focus is on deadlocked objects instead of deadlocked threads. Nevertheless, given the set of deadlocked objects, we can infer a set of involved threads: it is sufficient to consider those threads that block the considered objects by performing a get–statement, plus the threads from which such threads are expecting to receive a return value. By definition of classical deadlock, these additional threads are executing a grab–statement.

Given this observation, we can characterize Petri net markings corresponding to classical deadlocks, following the same approach used for extended deadlocks, but considering only threads performing get–statement that blocks their objects and threads executing grab–statements.

Definition 5.7. (Classical Deadlock Marking)
A marking m in a Petri net is a *classical deadlock marking* iff there exists a set of places D such that:

1. For each $p \in D$ we have: $m(p) \geq 1$, p is in the form $c@c'\langle t\rangle$, with t being one of "$\mathsf{get}@(o''.l''?, o')\,; t'$" or "$\mathsf{grab}(o'); t'$", and p satisfies all the following conditions
   (a) if $c = o.l?$ then there is $p' \in D$ in the form $c''@c'''\langle \mathsf{get}@(o.l?, o'''); t''\rangle$;
   (b) if $c' = o'.l'?$ then there is $p' \in D$ in the form $c''@c\langle \mathsf{get}@(o'.l'?, o'''); t''\rangle$;
   (c) if $t = \mathsf{grab}(o'); t'$ then t' is ?-free and there is $p' \in D$ in the form $c@c'\langle \mathsf{get}@(o'''.l'''?, o'); t''\rangle$.
2. For each $p \notin D$ we have: $m(p) \geq 1$ implies
   (a) p is ?-free or
   (b) p is in the form $c@c'\langle t\rangle$, with c', t being ?-free and $c = o.l?$ for some o,l such that there is $p' \in D$ in the form $c''@o.l?\langle t'\rangle$.

Following the same proof presented for Lemma 5.3 we can prove the following result. The unique additional observation to be done within the proof, consists of extracting the set of threads N from the set of objects O that are blocked in the considered classical deadlock. As discussed above the set of involved threads N is obtained by taking those threads that are blocking the objects in O plus the threads on which such threads are waiting.

Lemma 5.8. Let C_0 be an initial configuration and let $\alpha = C_0 \rightarrow \ldots \rightarrow C_k$ be an execution in the operational semantics. Let P_{C_0} be the Petri net associated to C_0. If C_k is a classical deadlock then $M(\alpha)$ includes a classical deadlock marking of P_{C_0}.

Following precisely the same proof presented for Theorem 5.6, but exploiting Lemma 5.8 instead of Lemma 5.3, we can prove what follows.

Theorem 5.9. (Classical Deadlock Marking Reachability)
Given a Creol program, if it has a reachable classical deadlock configuration, then the corresponding Petri net has a reachable classical deadlock marking.

We conclude by observing that what is stated in Lemma 5.5, i.e. it is possible to define a finite set of target marking denotations characterizing all the extended deadlock markings, obviously holds also for classical deadlocks: it is indeed sufficient to consider, in the proof, only sets of places D that satisfy Definition 5.7. Hence, we can conclude the decidability of reachability of classical deadlock markings precisely following the same reasoning reported in the decidablity proof for extended deadlock markings (Theorem 5.6).

6. Explicitly dealing with objects

We now show how the presented translation into Petri nets can be extended to also explicitly represent passing/returning of objects and reading/writing of fields of class type.

6.1. Generation of abstract statement traces

The machinery in Section 4.3 is modified, to also explicitly represent passing/returning of objects and reading/writing of fields of class type, as follows. We first assume, without loss of generality, that in the initial Creol specification C all parameters and variables x (including declarations binding them) that are of class type are distinguished from the other parameters/variables (they use distinguished names x) and are ranged over by $\hat{x}$. Notice that x still ranges over all variables, thus also $\hat{x}$ variables. We assume $\hat{v}$ to range over $\hat{x}$ parameters/varibales and objects o. Remember that we also assume Creol code to type check: e.g. method calls are correct with respect to formal parameters number and type. In relation to this, we have that type of fields are simply determined by initialization values n inside field declarations F (if n is an object, the type is the class it has in the initial configuration). Similarly, the return type of a method is given by the type of the value returned by method code.

We modify the definition of statement traces (Definition 4.3) by:

- Enriching method call statements to also include objects passed as parameters (non class type parameters are filtered out), i.e. we now consider let $o.l(o_1, \ldots, o_n)^+$ statements (with "+" being either the empty string or "?"): this also means that the abstract representation of futures, which are now identified by means of methods with a specific set of object parameters, gets correspondingly enriched.

- Enriching get statements related to calls of methods having class return type, so to represent the returned objects: they are now denoted by $\mathsf{let}\,\hat{x} = \mathsf{get@}\hat{v}.l(\vec{\hat{v}})^+$ and $\mathsf{let}\,\hat{x} = \mathsf{get@}(\hat{v}.l(\vec{\hat{v}})^+, o)$, where $\hat{x}$ represents the returned object and can be used in the remainder of the trace, e.g. in a method call statement, whose general form, thus, is $\mathsf{let}\,\hat{v}.l(\vec{\hat{v}})^+$ (remember that a $\hat{v}$ can be an $\hat{x}$).
- Adding new statements $\hat{v}.l := \hat{v}'$ and $\mathsf{let}\,\hat{x} = \hat{v}.l$ that represent writing and reading of fields, respectively.
- Adding new statements "$\hat{v}$" that represent the returned object in traces of a method having class return type.

In get statements related to calls of methods not having class return type the "$\mathsf{let}\,\hat{x} =$" prefix is not present, as before. The new definition of statement traces thus becomes as follows.

Definition 6.1. (Statement Trace)
An *abstract statement ast* is a statement of the following form:

$$\begin{array}{rcl} \mathit{ast} & ::= & \mathsf{let}\,\hat{v}.l(\vec{\hat{v}}) \mid \mathsf{get@}(\hat{v}.l(\vec{\hat{v}}), o) \mid \mathsf{get@}\hat{v}.l(\vec{\hat{v}}) \\ & \mid & \mathsf{let}\,\hat{x} = \mathsf{get@}(\hat{v}.l(\vec{\hat{v}}), o) \mid \mathsf{let}\,\hat{x} = \mathsf{get@}\hat{v}.l(\vec{\hat{v}}) \\ & \mid & \mathsf{let}\,\hat{v}.l(\vec{\hat{v}})? \mid \mathsf{get@}(\hat{v}.l(\vec{\hat{v}})?, o) \mid \mathsf{get@}\hat{v}.l(\vec{\hat{v}})? \\ & \mid & \mathsf{let}\,\hat{x} = \mathsf{get@}(\hat{v}.l(\vec{\hat{v}})?, o) \mid \mathsf{let}\,\hat{x} = \mathsf{get@}\hat{v}.l(\vec{\hat{v}})? \\ & \mid & \hat{v}.l := \hat{v}' \mid \mathsf{let}\,\hat{x} = \hat{v}.l \mid \mathsf{release}(o) \mid \mathsf{grab}(o) \mid \hat{v} \end{array}$$

Moreover we need to modify functions s_1-s_5 composing function ST that builds abstract traces from Creol code t. Code t can be either the code of thread $\mathit{run}\langle t\rangle$ in the initial configuration (as before) or be an instantiation of the code of a method with a certain object for self (as before) and a certain set of actual parameter objects replacing the class type method formal parameters, i.e.

$$t = \hat{t}[o/\mathit{self}\,][o'_1/\hat{x}_1]\ldots[o'_k/\hat{x}_k]$$

for some objects o, o'_1, …, o'_k, method l and class c, with $c[(F, M)]$ and $o[c, F, \bot]$ occurring in the initial configuration and $M.l = \varsigma(\mathit{self}{:}T).\lambda(\vec{x}{:}\vec{T}).\hat{t}$, with $\hat{x}_1, \ldots, \hat{x}_k$ being the parameters in $\vec{x}$ having class type.

Notice that, similarly as for future variables x in sequences (over *st*, *rst* and *mst* statements) of Section 4.3, we just obtain sequences where object variables $\hat{x}$ are bound by a previous statement beginning with "$\mathsf{let}\,\hat{x} =$" (declaring them). In particular this holds for the *ast* traces obtained by applying the whole ST function. In this regard, we introduce some terminology that will be used in the following. *Free variables* of an *ast* trace are variables $\hat{x}$ that are not bound by a previous statement beginning with "$\mathsf{let}\,\hat{x} =$". A *closed ast trace* is an *ast* trace with no free variables. Moreover, an *ast* trace has *distinguished binders* if it does not include two statements beginning with "$\mathsf{let}\,\hat{x} =$" for any variable $\hat{x}$. Therefore, given a Creol code t obtained from the initial configuration as described above and such that we use a different variable name in each variable let declaration (as we also assumed in Section 4.3), we have that $\mathsf{ST}(t)$ is a set of closed *ast* traces with distinguished binders.

We start from function s_1, which builds abstact code *ac*, i.e. a tree whose branches are labeled by *st* statements that are now defined as:

$$\begin{array}{rcl} st & ::= & \mathsf{let}\, x = \hat{v}.l(\vec{\hat{v}}) \mid \mathsf{claim@}(x,o) \mid \mathsf{get@}(x,o) \\ & \mid & \mathsf{let}\, \hat{x} = \mathsf{claim@}(x,o) \mid \mathsf{let}\, \hat{x} = \mathsf{get@}(x,o) \\ & \mid & \hat{v}.l := \hat{v}' \mid \mathsf{let}\, \hat{x} = \hat{v}.l \mid \mathsf{suspend}(o) \mid \hat{v} \end{array}$$

Function s_1 is now defined by:

- the old defining equations *apart from the last two ones*, where: $s_1(v) \triangleq \varepsilon$ is now applied only if there is no $\hat{v}$ such that $v = \hat{v}$; and the equations for claim, get, field writing and field reading are applied only if there is no $\hat{x}$ such that $x = \hat{x}$;
- the new defining equations:

$$\begin{array}{rcl} s_1(\hat{v}) & \triangleq & \hat{v} \\ s_1(\mathsf{let}\, \hat{x}{:}T = \mathsf{claim@}(x,o)\, \mathsf{in}\, t) & \triangleq & \mathsf{let}\, \hat{x} = \mathsf{claim@}(x,o); s_1(t) \\ s_1(\mathsf{let}\, \hat{x}{:}T = \mathsf{get@}(x,o)\, \mathsf{in}\, t) & \triangleq & \mathsf{let}\, \hat{x} = \mathsf{get@}(x,o); s_1(t) \\ s_1(\mathsf{let}\, \hat{x}{:}T = (\hat{v}.l := \hat{v}')\, \mathsf{in}\, t) & \triangleq & \hat{v}.l := \hat{v}'; s_1(t[v'/\hat{x}]) \\ s_1(\mathsf{let}\, \hat{x}{:}T = \hat{v}.l\, \mathsf{in}\, t) & \triangleq & \mathsf{let}\, \hat{x} = \hat{v}.l; s_1(t) \\ s_1(\mathsf{let}\, x{:}T = \hat{v}.l(\vec{v})\, \mathsf{in}\, t) & \triangleq & \mathsf{let}\, x = \hat{v}.l(objects(\vec{v})); s_1(t) \end{array}$$

  where $objects(\vec{v})$ yields the sequence of $\hat{v}$ elements obtained by filtering out elements v of $\vec{v}$ for which there is no $\hat{v}$ such that $v = \hat{v}$ (and preserving ordering).

Definition of function s_2, producing sets of sets of sequences over *st* statements is unchanged.

Concerning function s_3, it produces sets of sets of sequences over *rst* statements. Here such statements are those obtained from *st* statements above by removing $\mathsf{suspend}(o)$, $\mathsf{claim@}(x,o)$ and $\mathsf{let}\, \hat{x} = \mathsf{claim@}(x,o)$ statements and by adding $\mathsf{release}(o)$, $\mathsf{grab}(o)$, $\mathsf{get@}x$ and $\mathsf{let}\, \hat{x} = \mathsf{get@}x$ statements. Function s_3 definition is simply modified by: replacing the first defining equation with equations that skip in the same way the statements $\mathsf{let}\, x = \hat{v}.l(\vec{\hat{v}})$, $\hat{v}.l := \hat{v}'$, $\mathsf{let}\, \hat{x} = \hat{v}.l$, and $\hat{v}$; adding a defining equation for $\mathsf{let}\, \hat{x} = \mathsf{claim@}(x,o)$ that is like that of $\mathsf{claim@}(x,o)$, but produces $\mathsf{let}\, \hat{x} = \mathsf{get@}x$ instead of $\mathsf{get@}x$; and, similarly, adding a defining equation for $\mathsf{let}\, \hat{x} = \mathsf{get@}(x,o)$ that is like that of $\mathsf{get@}(x,o)$, but produces $\mathsf{let}\, \hat{x} = \mathsf{get@}(x,o)$ instead of $\mathsf{get@}(x,o)$.

We now consider function s_4 that produces sets of sets of sequences over *mst* statements. Here such statements are those obtained by adding, to *rst* statements above, variants where futures are marked, i.e.: $\mathsf{let}\, x = \hat{v}.l(\vec{\hat{v}})?$, $\mathsf{get@}(x?,o)$, $\mathsf{let}\, \hat{x} = \mathsf{get@}(x?,o)$ $\mathsf{get@}x?$ and $\mathsf{let}\, \hat{x} = \mathsf{get@}x?$. Function s_4 definition is simply modified as follows. In the definition of $s_4(t)$ instead of considering just two sets of traces to be added to the original trace t, we now consider other two additional sets, which are defined in the same way apart from $\mathsf{let}\, \hat{x} = \mathsf{get@}(x^+,o)$ and $\mathsf{let}\, \hat{x} = \mathsf{get@}x^+$ statements replacing $\mathsf{get@}(x^+,o)$ and $\mathsf{get@}x^+$ statements, respectively (with "+" being either the empty string or "?").

Concerning the defining equation of the tagging function m, it is the same apart from "let $x = \hat{v}.l(\vec{\hat{v}})$" replacing "let $x = o'.l$" and "let $x = \hat{v}.l(\vec{\hat{v}})?$" replacing "let $x = o'.l?$".

Finally, definition of function s_5, producing sets of sets of sequences over *ast* statements, is modified by just considering "$\hat{v}.l(\vec{\hat{v}})$" instead of "$o.l$", wherever the latter occurs (even if tagged) in the s_5 defining equations (the existential quantifier in the side condition must now consider both $\hat{v}$ and $\vec{\hat{v}}$ instead of o).

6.2. Petri net construction for creol programs

We are now in a position to present the extended Petri net construction. We first need some preliminary definitions. Given a variable $\hat{x}$ occurring in a closed *ast* trace t with distinguished binders, we define the *type of* $\hat{x}$ *in* t as follows:

- If $\hat{x}$ is declared by a $\mathsf{let}\ \hat{x} = \mathsf{get@}(\hat{v}.l(\vec{\hat{v}})^+, o)$ or a $\mathsf{let}\ \hat{x} = \mathsf{get@}\hat{v}.l(\vec{\hat{v}})^+$ statement, then the type of $\hat{x}$ is the return type of method l of the class of $\hat{v}$. The latter is determined as follows: if $\hat{v}$ is an object o' then it is simply the class of o' (given by the initial configuration); if, instead, $\hat{v}$ is a variable $\hat{y}$ then it is the type of $\hat{y}$ in t.

- If $\hat{x}$ is declared by a $\mathsf{let}\ \hat{x} = \hat{v}.l$ statement, then the type of $\hat{x}$ is the type of field l of the class of $\hat{v}$. The latter is determined as in the previous case.

Notice that the statement declaring $\hat{y}$ must occur in t before the statement declaring $\hat{x}$, so the above inductive definition is well-formed.

We define a *closed suffix* of a closed *ast* trace t with distinguished binders to be an *ast* trace $t'[o'_1/\hat{x}_1] \ldots [o'_k/\hat{x}_k]$ where: t' is any suffix of t and, supposing $\hat{x}_1, \ldots, \hat{x}_k$ to be the free variables of t', $o'_1, \ldots, o'_k$ are any objects such that for all i, with $1 \leq i \leq k$, the class of o'_i is the type of $\hat{x}_i$ in t. Notice that we use the $t[v/x]$ notation, similarly as we did for Creol code, to denote the result of syntactically replacing free variable x by v in an *ast* trace t.

The Petri net $P_{C_0} = \langle P, T, m_0 \rangle$ associated to an initial configuration

$$C_0 = c_1[(F_1, M_1)] \,\|\, \ldots \,\|\, c_m[(F_m, M_m)] \,\|\, o_1[c_{k_1}, F_{k_1}, \bot] \,\|\, \ldots \,\|\, o_n[c_{k_n}, F_{k_n}, \bot] \,\|\, \mathit{run}\langle \bar{t} \rangle$$

is defined as in Definition 4.9 apart from the following modifications.

Concerning Petri net places P, those that are in the form $c@c'\langle t \rangle$ must now satisfy the following conditions:

- the same requirement about the form of $c@c'$ considered in Definition 4.9 except that, for both c and c', we here consider, instead of pairs $o_i.l$, pairs $o_i.l(o'_1, \ldots, o'_k)$, where: l is a method of the class c_{k_i} whose parameters $\vec{x} : \vec{T}$ are such that $objects(\vec{x})$ has length k; and $o'_1, \ldots, o'_k$ are such that $\{o'_1, \ldots, o'_k\} \subseteq \{o_1, \ldots, o_n\}$ and the class of o'_i is the type of the i-th element of $objects(\vec{x})$

- if $c@c' = \mathit{run}@\mathit{run}$ then t is a closed suffix of one of the traces in $\mathsf{ST}(\bar{t})$

- if $c@c'$ is $c@o_i.l(o'_1,\ldots,o'_k)^+$ then t is a closed suffix of a trace in the set

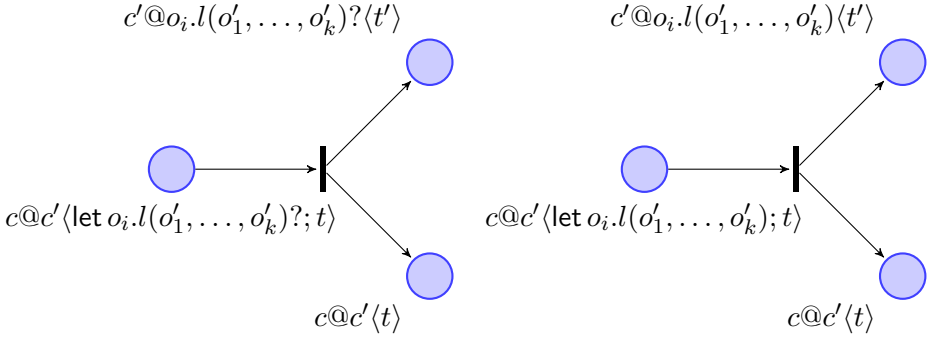
$$\mathsf{grab}(o_i);\mathsf{ST}(\hat{t}[o_i/self\,][o'_1/\hat{x}_1]\ldots[o'_k/\hat{x}_k]);\mathsf{release}(o_i)$$

  where $(\hat{x}_1,\ldots,\hat{x}_k) = objects(\vec{x})$, with $\vec{x}$ being the parameters of method l, and $\hat{t}$ is the code of the method l, with "*self*" denoting the self object; namely, being c_{k_i} the class of o_i, we have $M_{k_i}.l = \varsigma(self{:}T).\lambda(\vec{x}{:}\vec{T}).\hat{t}$

Moreover, we add to P totally new places $o_i.l\langle o_j\rangle$, for any object o_i, field l defined by class c_{k_i}, and o_j whose class is the type of field l of c_{k_i}. Place $o_i.l\langle o_j\rangle$ represents field l of o_i to be currently set to object o_j.

Concerning Petri net transitions T, the following transitions are redefined (we recall that "$*$" and "$+$" are meta-variables being both either the empty string or "?"):

T-2 For each $c@c'\langle\mathsf{let}\ o_i.l(o'_1,\ldots,o'_k)^*;t\rangle \in P$ and trace t' in the set $\mathsf{grab}(o_i);\mathsf{ST}(\hat{t}[o_i/self\,][o'_1/\hat{x}_1]\ldots[o'_k/\hat{x}_k]);\mathsf{release}(o_i)$ with $\hat{t}$ being class c_{k_i} method l code (in which "self" denotes the self object and $\vec{\hat{x}}$ are its class type parameters, as above), there is $\mathsf{t}\in T$ such that
${}^\bullet\mathsf{t} = \{c@c'\langle\mathsf{let}\ o_i.l(o'_1,\ldots,o'_k)^*;t\rangle\}$ and $\mathsf{t}^\bullet = \{c@c'\langle t\rangle, c'@o_i.l(o'_1,\ldots,o'_k)^*\langle t'\rangle\}$ (see Fig. 7.a for the case of a tagged method call and Fig. 7.b for a non tagged one).



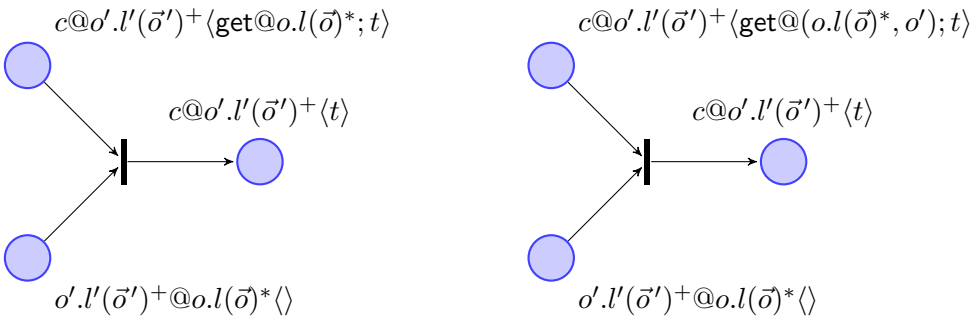
(a) tagged calls (b) calls without tag

Figure 7. Transitions for method calls with explicit objects.

T-5 For each $c@o'.l'(\vec{o}')^+\langle\mathsf{get}@o.l(\vec{o})^*;t\rangle \in P$ there is $\mathsf{t}\in T$ such that
${}^\bullet\mathsf{t} = \{c@o'.l'(\vec{o}')^+\langle\mathsf{get}@o.l(\vec{o})^*;t\rangle, o'.l'(\vec{o}')^+@o.l(\vec{o})^*\langle\rangle\}$ and
$\mathsf{t}^\bullet = \{c@o'.l'(\vec{o}')^+\langle t\rangle\}$ (see Fig. 8.a).

T-6 For each $c@o'.l'(\vec{o}')^+\langle\mathsf{get}@(o.l(\vec{o})^*,o');t\rangle \in P$ there is $\mathsf{t}\in T$ such that
${}^\bullet\mathsf{t} = \{c@o'.l'(\vec{o}')^+\langle\mathsf{get}@(o.l(\vec{o})^*,o');t\rangle, o'.l'(\vec{o}')^+@o.l(\vec{o})^*\langle\rangle\}$ and
$\mathsf{t}^\bullet = \{c@o'.l'(\vec{o}')^+\langle t\rangle\}$ (see Fig. 8.b).

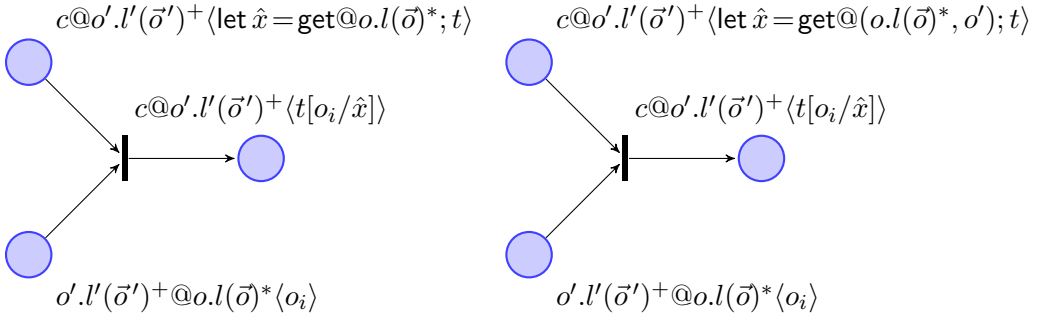
The following transitions are added:



(a) get by a thread not holding the lock (b) get by a thread holding the lock

Figure 8. Translation of get returning a non object value.

T-7 For each $c@o'.l'(\vec{o}')^+\langle \mathsf{let}\ \hat{x} = \mathsf{get}@o.l(\vec{o})^*; t\rangle \in P$ and object o_i whose class c_{k_i} is the return type of method l of object o, there is $\mathsf{t} \in T$ such that
${}^\bullet\mathsf{t} = \{c@o'.l'(\vec{o}')^+\langle \mathsf{let}\ \hat{x} = \mathsf{get}@o.l(\vec{o})^*; t\rangle, o'.l'(\vec{o}')^+@o.l(\vec{o})^*\langle o_i\rangle\}$ and
$\mathsf{t}^\bullet = \{c@o'.l'(\vec{o}')^+\langle t[o_i/\hat{x}]\rangle\}$ (see Fig. 9.a).

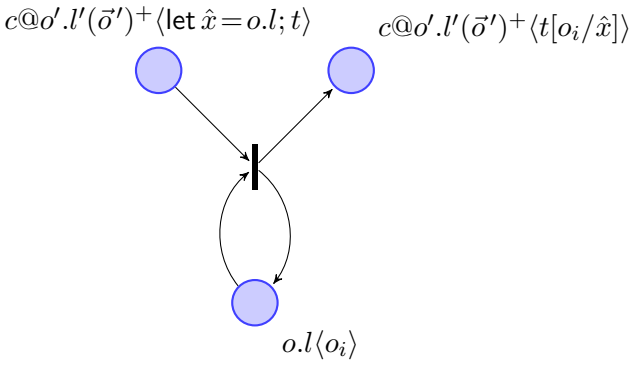


(a) get by a thread not holding the lock (b) get by a thread holding the lock

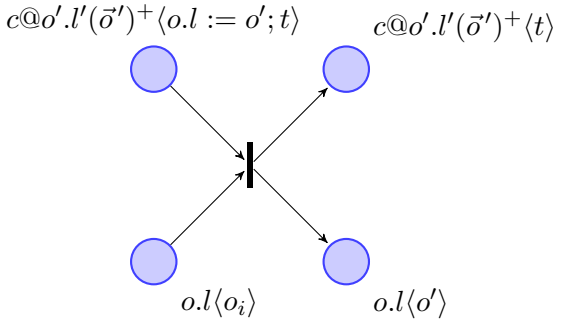
Figure 9. Translation of get returning an object.

T-8 For each $c@o'.l'(\vec{o}')^+\langle \mathsf{let}\ \hat{x} = \mathsf{get}@(o.l(\vec{o})^*, o'); t\rangle \in P$ and object o_i whose class c_{k_i} is the return type of method l of object o, there is $\mathsf{t} \in T$ such that
${}^\bullet\mathsf{t} = \{c@o'.l'(\vec{o}')^+\langle \mathsf{let}\ \hat{x} = \mathsf{get}@(o.l(\vec{o})^*, o'); t\rangle, o'.l'(\vec{o}')^+@o.l(\vec{o})^*\langle o_i\rangle\}$ and
$\mathsf{t}^\bullet = \{c@o'.l'(\vec{o}')^+\langle t[o_i/\hat{x}]\rangle\}$ (see Fig. 9.b).

T-9 For each $c@o'.l'(\vec{o}')^+\langle \mathsf{let}\ \hat{x} = o.l; t\rangle \in P$ and object o_i whose class c_{k_i} is the type of field l of object o, there is $\mathsf{t} \in T$ such that
${}^\bullet\mathsf{t} = \{c@o'.l'(\vec{o}')^+\langle \mathsf{let}\ \hat{x} = o.l; t\rangle, o.l\langle o_i\rangle\}$ and
$\mathsf{t}^\bullet = \{c@o'.l'(\vec{o}')^+\langle t[o_i/\hat{x}]\rangle, o.l\langle o_i\rangle\}$ (see Fig. 10.a).
Notice that we have just to read the field, so the token must not be removed from place $o.l\langle o_i\rangle$.



(a) field reading



(b) field writing

Figure 10. Translation of field reading and writing.

T-10 For each $c@o'.l'(\vec{o}')^+\langle o.l := o'; t\rangle \in P$ and object o_i whose class c_{k_i} is the type of field l of object o, there is $\mathsf{t} \in T$ such that
${}^\bullet\mathsf{t} = \{c@o'.l'(\vec{o}')^+\langle o.l := o'; t\rangle, o.l\langle o_i\rangle\}$ and $\mathsf{t}^\bullet = \{c@o'.l'(\vec{o}')^+\langle t\rangle, o.l\langle o'\rangle\}$ (see Fig. 10.b).
Notice that we have to remove the token from place $o.l\langle o_i\rangle$ representing the old field value.

Concerning Petri net initial marking m_0, places $o_i.l\langle o\rangle$ are added (with one token), for any i, with $1 \leq i \leq n$, and any field l defined in F_{k_i} (containing field definitions for object o_i in the initial configuration) such that $F_{k_i}.l = o$.

6.3. Soundness and deadlock analysis

Theorem 4.11 concerning soundness of the Petri net construction still holds from the modified construction above when a properly extended definition of the mapping function $M(C_0 \to \ldots \to C_k)$ (from computations to sets of markings) is considered. M is modified by generating abstracted thread identifiers in the form $o.l(\vec{o})@o'.l'(\vec{o}')$, i.e. taking into account also objects that are actually passed at method invocation, instead of $o.l@o'.l'$. Moreover each marking of M is extended so to include, for any i, with $1 \leq i \leq n$, and any field l defined in F_{k_i} (containing field definitions for object o_i in

the initial configuration), one token in the place $o_i.l\langle o\rangle$, with o being such that, considered subterm $o_i[c_{k_i}, F, L]$ of configuration C_k, it holds $F.l = o$ (i.e. o is the current value of the field l of object o_i).

Moreover, concerning deadlock analysis, the definition of extended and classical deadlock marking (Definitions 5.1 and 5.7) are the same apart from considering $o.l(\vec{o})$ $(o'.l'(\vec{o}'), \dots)$ instead of $o.l$ $(o'.l', \dots,$ respectively) everywhere. Notice that the new places $o_i.l\langle o\rangle$ we introduced here are considered to be *?-free*.

Soundness Theorem 4.11 and all lemmas and theorems about deadlock analysis considered in Section 5 can be shown to still hold by following the same proof techniques we use in Appendices A and B.

7. Related work and conclusion

In this paper we presented a Petri net modeling of the concurrent object-oriented language Creol [5]. Creol is based on asynchronous method invocations: such invocations generate concurrent threads, that compute a return value that is accessed by the invoker through a future variable. Threads are executed according to a cooperative concurrency model: an object executes only one of its threads at a time, and the executing thread continues until it explicitly indicate a release point. Distinct objects can execute in parallel. We exploited our Petri net model of Creol to define a formal approach to the analysis of deadlock. We showed soundness of our analysis with respect to extended deadlocks (which encompass also blocked threads 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 Creol system is deadlock free.

Concerning the other direction, a Creol system could be deadlock free while our analysis detect the presence of deadlocks: but this is obvious as deadlock is an undecidable property. We had to perform several abstractions to resort to a decidable model like Petri nets, like data abstraction. This implies that some of the computations present in the Petri net model do not have a counterpart in the corresponding Creol system.

The relationship between Petri nets and concurrent object-oriented programming has been widely investigated since the 90s, when a couple of thematic workshops have been organized on this specific topic. The outcomes of such meetings and the main achievements of the research community were subsequently collected in a reference book in the area [16]. Most of the research effort was focused on how to successfully combine the benefits and advantages of both Petri nets and object-oriented programming: many languages and notations have been proposed among which Object Petri Nets (OPN) [17], Concurrent Object-Oriented Petri Nets (CO-OPN) [18], CoOperative Objects (COO) [19], and the Object Based Petri Net Programming Notation (OB(PN)^2) [20]. Differently from these approaches, here we simply use standard finite place/transition Petri nets to abstractly model the concurrency mechanisms adopted in Creol. In particular, we make use of the rather unique combination of expressivity and amenability for analysis that characterizes Petri nets, which are sufficiently expressive to model concurrent infinite-state systems while keeping decidable interesting properties like coverability and reachability.

We now report some remarks on other related work on formally grounded deadlock analysis for cooperative concurrent object-oriented languages.

In [21, 22] 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 their approach, our analysis is somehow more precise in that it is thread based and not just object based. An example of a false positive detected by their approach, taken from [23] (and rephrased in 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 initial method run are as follows:

- $run() \triangleq o_1.m_1()$
- $m_1() \triangleq \mathsf{let}\, x = o_2.m_2()\, \mathsf{in}$
  $\mathsf{claim@}(x, self); 1$
- $m_2() \triangleq \mathsf{let}\, x = o_1.m_3()\, \mathsf{in}$
  $\mathsf{get@}(x, self); 1$
- $m_3() \triangleq 1$

This program would originate a deadlock if we had a *get* instead of a *claim* in method m_1. This because, as for the first example of the Introduction, method m_2 would be stuck waiting for m_3 that cannot proceed because the lock on object o_1 is kept by m_1 that now performs the *get*. The technique in [21, 22] correctly reports such a deadlock, but also wrongly detects the presence of a deadlock in the above program (where the use of *claim* instead of *get* causes the o_1 object lock to be released allowing m_3 to proceed). Our analysis is, instead, more precise, in that it distinguishes the two cases and correctly detects that the program above is deadlock free. Concerning language expressivity, the language in [21, 22] additionally considers, with respect to our language, a "new" primitive for object creation. We also could easily encode in our analysis the creation of a finite set of objects by considering all such objects to be present since the beginning and then only "activated". More precisely, given a bound k, we could extend our Petri net semantics with additional places (and transitions) to model k additional objects for each class, that will be exploited to deal with the first k executions of the "new" primitive on each class. All our results would continue to hold under the assumption that at most k objects are created during a considered computation. This technique allows us to detect only deadlocks that occur in computations with a predefined limited number of object creations (notice that a similar limitation applies also to [21]). Nevertheless, there are cases (like the case of *linear* recursion discussed in [22]) in which it is possible to predefine a bound with the guarantee that: if no deadlock has been found in computations with at most this bounded number of object creation, then it will not appear afterwards. In the case of a language with a mechanism for destroying objects (explicitly or by garbage collection), this approach could find a broader application by extending it with a suitable reuse of released objects (see e.g. name reuse techniques in [24]): the bound would then concern the number of contemporaneously active objects.

We conclude by commenting that our notion of extended deadlock has been a source of inspiration for a paper [25] that, by exploiting static analysis instead of resorting to Petri nets, proposes a technique for detecting deadlocks in an actor-based language with futures.

Acknowledgements. We thank the anonymous reviewers for their useful comments and suggestions that helped us to significantly improve the paper.

References

[1] Armstrong J. Erlang. *Communications of ACM*, 2010. **53**(9):68–75. doi:10.1145/1810891.1810910.

[2] Haller P, Odersky M. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 2009. **410**(2-3):202–220. URL `https://doi.org/10.1016/j.tcs.2008.09.019`.

[3] Caromel D, Henrio L, Serpette BP. Asynchronous and deterministic objects. *SIGPLAN Not.*, 2004. **39**(1):123–134. doi:10.1145/982962.964012.

[4] Edward G Coffman J, Elphick MJ, Shoshani A. System Deadlocks. *ACM Computing Surveys*, 1971. **3**(2):67–78.

[5] Johnsen EB, Owe O. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and Systems Modeling*, 2007. 2007. **6**(1):39–58. doi:10.1007/s10270-006-0011-2.

[6] Gul A Agha SFS Ian A Mason, Talcott CL. A foundation for actor computation. *Journal of Functional Programming*, 1997. **7**(1):1–72. doi:10.1017/S095679689700261X.

[7] Dijkstra EW. Cooperating sequential processes. In: Genuys F (ed.), Programming Languages: NATO Advanced Study Institute, pp. 43–112. Academic Press, 1968.

[8] Holt RC. Some Deadlock Properties of Computer Systems. *ACM Computing Surveys*, 1972. **4**(3):179–196. doi:10.1145/356603.356607.

[9] de Boer FS, Bravetti M, Grabe I, Lee MD, Steffen M, Zavattaro G. A Petri Net Based Analysis of Deadlocks for Active Objects and Futures. In: Proceedings of 9th International Symposium on Formal Aspects of Component Software (FACS), volume 7684 of *Lecture Notes in Computer Science*. Springer, 2013 pp. 110–127. doi:10.1007/978-3-642-35861-6_7.

[10] 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*, 2009. **78**(7):491–518. URL `https://doi.org/10.1016/j.jlap.2009.01.001`.

[11] Gordon AD, Hankin PD. A Concurrent Object Calculus: Reduction and Typing. In: Proceedings of 3rd International Workshop on High-Level Concurrent Languages (HLCL 1998), volume 16 no 3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998 pp. 248–264. URL `https://doi.org/10.1016/S1571-0661(04)00145-8`.

[12] Abadi M, Cardelli L. A Theory of Objects. Monographs in Computer Science. Springer-Verlag, 1996. ISBN:0387947752.

[13] Mayr E. Persistence of Vector Replacement Systems is Decidable. *Acta Informatica*, 1981. **15**:309–318. URL `https://doi.org/10.1007/BF00289268`.

[14] Karp R, Miller R. Parallel Program Schemata. *Journal of Computer and system Sciences*, 1969. **3**(2):147–195. URL https://doi.org/10.1016/S0022-0000(69)80011-5.

[15] Busi N, Zavattaro G. Deciding reachability problems in Turing-complete fragments of Mobile Ambients. *Mathematical Structures in Computer Science*, 2009. **19**(6):1223–1263. URL https://doi.org/10.1017/S0960129509990181.

[16] Agha G, de Cindio F, Rozenberg G (eds.). Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets, volume 2001 of *Lecture Notes in Computer Science*. Springer, 2001. doi:10.1007/3-540-45397-0.

[17] Valk R. Object Petri Nets: Using the Nets-within-Nets Paradigm. In: Advances in Petri Nets, volume 3098 of *Lecture Notes in Computer Science*. Springer, 2003 pp. 819–848. URL https://doi.org/10.1007/978-3-540-27755-2_23.

[18] Buchs D, Guelfi N. CO-OPN: a Concurrent Object Oriented Petri Net Approach. In: Proceedings of the 12th International Conference on Application and Theory of Petri Nets (ICATPN). 1991 pp. 432–454.

[19] Sibertin-Blanc C. CoOperative Objects: Principles, Use and Implementation. In: Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets, volume 2001 of *Lecture Notes in Computer Science*. Springer, 2001 pp. 216–246. URL https://doi.org/10.1007/3-540-45397-0_7.

[20] Lilius J. OB(PN)^2: An Object Based Petri Net Programming Notation (Extended Abstract). In: Proceedings of the 2nd International Euro-Par Conference (Euro-Par), volume 1123 of *Lecture Notes in Computer Science*. Springer, 1996 pp. 660–663.

[21] Giachino E, Laneve C. Analysis of Deadlocks in Object Groups. In: Proceedings of the International Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE), volume 6722 of *Lecture Notes in Computer Science*. Springer, 2011 pp. 168–182. URL https://doi.org/10.1007/978-3-642-21461-5_11.

[22] Giachino E, Laneve C, Lienhardt M. A framework for deadlock detection in core ABS. *Software and System Modeling*, 2016. **15**(4):1013–1048. URL https://doi.org/10.1007/s10270-014-0444-y.

[23] Giachino E, Laneve C, Lascu T. Deadlock and Livelock Analysis in Concurrent Objects with Futures. Technical report, University of Bologna, 2011. URL http://www.cs.unibo.it/laneve/publications.html.

[24] Bravetti M, Gorrieri R. Deciding and axiomatizing weak ST bisimulation for a process algebra with recursion and action refinement. *ACM Transactions on Computational Logic*, 2002. **3**(4):465–520. doi:10.1145/566385.566386.

[25] Flores-Montoya A, Albert E, Genaim S. May-Happen-in-Parallel Based Deadlock Analysis for Concurrent Objects. In: Proceedings of the International Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE), volume 7892 of *Lecture Notes in Computer Science*. Springer, 2013 pp. 273–288. URL https://doi.org/10.1007/978-3-642-38592-6_19.

A. Proof of Theorem 4.11

In this section we define a mapping M from executions in the operational semantics of C_0 to sets of markings of P_{C_0} such that, for all executions $C_0 \to \cdots \to C_k$, each marking in $M(C_0 \to \ldots \to C_k)$ is reachable from the initial marking of P_{C_0}.

A.1. Generation of abstract traces for marking construction

We now present the formal machinery that, given an execution $\alpha = C_0 \to \ldots \to C_k$ of an initial configuration C_0, transforms codes t of threads $n\langle t\rangle$ occurring in C_k into abstract traces of Definition 4.3. Such traces will be used, in Section A.2, by function M to generate a set of markings $M(\alpha)$ of the Petri net P_{C_0}.

In order to define such a transformation we will resort to extensions of the (chained) transformation functions s_1, s_2, s_3, s_4 and s_5 we introduced in Section 4.3. This because code t of threads of (non-initial) configurations may additionally include: names n of (created) threads and run-time syntax commands $\mathsf{get@}n$, $\mathsf{grab}(o)$ and $\mathsf{release}(o)$.

Notice that, since extended functions are defined (w.r.t. the definition we gave in Section 4.3) over an enlarged domain and they preserve their definition over the original domain, we will keep using for them the same name s_i we used in Section 4.3. Concerning function s_3, we will denote the corresponding extended function with s_3^F, with F being a set of thread/future names n (the original function s_3 being obtained for $F = \emptyset$). The parameter F, which is expected to include the names n of the futures already consumed (read) by some get/claim statement according to the historical information recorded in the execution α, is needed to deal with get/claim on future names n (instead of future variables x as in the case of static code considered in Section 4.3) that may now additionally occur in a sequence inputed to the s_3 function. In particular, this information is used to establish if a get/claim on n is the first consumption of the future or not (if it is not, then s_3^F, consistently with its original definition, removes it).

Moreover, we will add a new transformation function, denoted by om^α, that will be chained at the end of functions of Section 4.3 as a final transformation step. Such a function is needed in order to abstract from (free) thread/future names n that now occur in sequences: similarly as for function s_5, which just operates on future variables x, they are replaced by om^α with the pair $o.l$, i.e. called object "o" and called method "l". However, while s_5 retrieves this information from the definition of variable x, om^α gets it from historical information recorded in the execution α.

In order to formally define such transformation functions, we first need to introduce an auxiliary function.

The function $\mathcal{C}$ (**c**onsumed futures) takes a trace β of the operational semantic and returns a set of thread/future names denoted with $\mathcal{C}^\beta$. A thread/future name is in the set iff the future was read at least once in the trace.

Definition A.1. Given an execution β we denote with $\mathcal{C}^\beta$ a set of future names defined as follows, where $\alpha = C_0 \to \ldots \to C_k$ and $\alpha' = C_0 \to \ldots \to C_k \to C_{k+1}$:

- $\mathcal{C}^{C_0} = \emptyset$
- for $k \geq 0$ we have:
  - $\mathcal{C}^{\alpha'} = \mathcal{C}^\alpha \cup \{n\}$ if $C_k \xrightarrow{n} C_{k+1}$;
  - $\mathcal{C}^{\alpha'} = \mathcal{C}^\alpha$ otherwise.

Formally, given an execution $\alpha = C_0 \to \ldots \to C_k$ of the operational semantics, we define transformation $s_{1,4}^\alpha$ that takes Creol thread code t, with t being such that $n\langle t\rangle$ occurs in C_k for some

thread name n, and yields a set of sequences over tagged run-time statements. Such a transformation is defined as the composition of extended functions:

$$s^{\alpha}_{1,4} \triangleq s_4 \circ s_3^{\mathcal{C}^{\alpha}} \circ s_2 \circ s_1$$

As we will see in Section A.2, while transformation steps s_1, s_2, s_3^F and s_4 are directly chained, we will apply transformations s_5 and om^{α} only after a particular (single) sequence over the set of $s^{\alpha}_{1,4}(t)$ tagged run-time statement sequences has been chosen, thus finally obtaining an abstract statement trace of Definition 4.3.

In the following we will formally define the extended s_1, s_2, s_3^F, s_4, s_5 functions and the new om^{α} functions.

As in Section 4.3 we have that, for all trees/sequences considered in the input and output of the various steps, each claim/get statement using a future parameter x is bound by a let declaration of variable x occurring previously in the tree/sequence. We also make the same assumption that, in the initial Creol thread code t, we use a different variable name in each variable let declaration occurring inside it (after the final transformation step the same abstract statement traces will be obtained, no matter the specific name chosen for the variables).

A.1.1. s_1 extension

It transforms the code t of a thread $n\langle t\rangle$ into an abstract code tree *ac*, with branches labeled by *st* statements with extended syntax:

$$\begin{array}{rcl} \mathit{st} & ::= & \mathsf{let}\, x = o.l \mid \mathsf{claim@}(v,o) \mid \mathsf{get@}(v,o) \mid \mathsf{suspend}(o) \\ & \mid & \mathsf{get@}n \mid \mathsf{grab}(o) \mid \mathsf{release}(o) \end{array}$$

Formally, the definition of the extended s_1 function is the same as that of the s_1 function, with x' replaced by v in the defining equations of claim and get, i.e. obtaining equations

$$\begin{array}{rcl} s_1(\mathsf{let}\, x{:}T = \mathsf{claim@}(v,o)\, \mathsf{in}\, t) & \triangleq & \mathsf{claim@}(v,o); s_1(t) \\ s_1(\mathsf{let}\, x{:}T = \mathsf{get@}(v,o)\, \mathsf{in}\, t) & \triangleq & \mathsf{get@}(v,o); s_1(t) \end{array}$$

after the modification (so to encompass also the case of claim and get on future names n), and with the addition of the following defining equations:

$$\begin{array}{rcl} s_1(\mathsf{let}\, x{:}T = \mathsf{get@}n\, \mathsf{in}\, t) & \triangleq & \mathsf{get@}n; s_1(t) \\ s_1(\mathsf{let}\, x{:}T = \mathsf{grab}(o)\, \mathsf{in}\, t) & \triangleq & \mathsf{grab}(o); s_1(t) \\ s_1(\mathsf{let}\, x{:}T = \mathsf{release}(o)\, \mathsf{in}\, t) & \triangleq & \mathsf{release}(o); s_1(t) \end{array}$$

A.1.2. s_2 extension

It transforms an abstract code tree *ac* into a set of strings over *st* statements with the extended syntax above.

Formally, the definition of the extended s_2 function is the same as that of s_2 function, taking into account that now we have an extended syntax for *st* statements.

A.1.3. s_3 extension

It takes as a parameter a set of thread/future names F and it transforms a set of sequences over *st* statements into a set of sequences over *rst* statements with extended syntax:

$$\mathit{rst} \quad ::= \quad \mathsf{let}\, x = o.l \mid \mathsf{get@}(v, o) \mid \mathsf{get@}v \mid \mathsf{release}(o) \mid \mathsf{grab}(o)$$

Formally, the definition of the extended s_3^F function is the same as that of the s_3^F function with F now being a set of future variables and names (variables being added to the initial set of future names during the induction), with x replaced by v in the defining equations for $\mathsf{claim@}(x, o)$ and $\mathsf{get@}(x, o)$, and with the addition of the following defining equations:

$$\begin{aligned} s_3^F(\mathsf{get@}n; t) &\triangleq \mathsf{get@}n; s_3^{F\cup\{n\}}(t) \\ s_3^F(\mathsf{grab}(o); t) &\triangleq \mathsf{grab}(o); s_3^F(t) \\ s_3^F(\mathsf{release}(o); t) &\triangleq \mathsf{release}(o); s_3^F(t) \end{aligned}$$

Notice that a run-time get on a future name n may appear only if the future has still to be consumed (read for the first time). This because, if $n \in \mathcal{C}^\alpha$ then the operational semantics of Creol does not transform a claim into a run-time get (it just directly reads the result of the future). Therefore in the corresponding defining equation we do not need to check if $n \notin F$.

As we did for the s_3^F function, we lift the extended s_3^F function to work over sets of sequences, so to obtain the described s_3^F transformation (that can be chained with the s_2 transformation).

A.1.4. s_4 extension

It transforms a set of sequences over *rst* statements into a set of sequences over tagged run-time statements *mst* with extended syntax

$$\mathit{mst} \quad ::= \quad \mathit{rst} \mid \mathsf{let}\, x = o.l? \mid \mathsf{get}(v?, o) \mid \mathsf{get@}v?$$

Formally, the definition of the extended s_4 function is as follows. Given a single *rst* sequence t, the set of *mst* sequences $s_4(t)$ is the union of the set yielded by the original s_4 function with

$$\{t_1; \mathsf{get}(n?, o); t_2 \mid t = t_1; \mathsf{get}(n, o); t_2\} \cup$$
$$\{t_1; \mathsf{get@}n?; t_2 \mid t = t_1; \mathsf{get@}n; t_2\}$$

As we did for the s_4 function, we lift the extended s_4 function to work over sets of sequences, so to obtain the described s_4 transformation (that can be chained with the s_3^F transformation).

A.1.5. s_5 extension

It transforms a sequence over *mst* statements into a sequence over extended abstract statements with the following syntax

$$\mathit{east} \quad ::= \quad \mathit{ast} \mid \mathsf{get}(n, o) \mid \mathsf{get}(n?, o) \mid \mathsf{get@}n \mid \mathsf{get@}n?$$

with *ast* being abstract statements defined in Definition 4.3. This means that, differently from future variables, (free) future names occurring in the input sequences are not abstracted in this step. This because the corresponding method calls were performed in the past, thus such sequences do not include the information needed (called object and called method) to abstract them.

Formally, the defining equations of the extended s_5 function are the same as those of the original s_5 function (with *mst* statements now belonging to the extended syntax above).

A.1.6. Additional step (Future name abstraction)

It takes as a parameter an execution α and it transforms a sequence over *east* statements into a sequence over *ast* abstract statements. It works by extracting, from the execution history α, the pair called object-called method that generated a given future name, and by replacing it, inside the input *east* sequence, with the retrieved pair.

We formalize this additional step via function om (**o**bject-**m**ethod) that takes an execution β of the operational semantics and yields a partial function from thread/future names to $o.l$ object-method pairs that we denote with om^β: $n \to o.l \in \mathsf{om}^\beta$ means that the thread/future n was created when calling method l on the object o.

Definition A.2. Given an execution β we denote with om^β a partial function from future names to object-method pairs defined as follows, where $\alpha = C_0 \to \ldots \to C_k$ and $\alpha' = C_0 \to \ldots \to C_k \to C_{k+1}$:

- $\mathsf{om}^{C_0} = \emptyset$
- for all $k \geq 0$
  - $\mathsf{om}^{\alpha'} = \mathsf{om}^\alpha \cup \{n_2 \to o.l\}$ if $C_k \xrightarrow{n_1,n_2,o.l} C_{k+1}$;
  - $\mathsf{om}^{\alpha'} = \mathsf{om}^\alpha$, otherwise.

The described transformation is obtained by lifting partial function om^α to work as a substitution over *east* sequences. Given an *east* sequence t, we define

$$\mathsf{om}^\alpha(t) \triangleq t[\mathsf{om}^\alpha(n)/n \mid n \in dom(\mathsf{om}^\alpha)]$$

that is $\mathsf{om}^\alpha(t)$ yields the *ast* sequence obtained by replacing each occurrence of a future name n in $dom(\mathsf{om}^\alpha)$ with $\mathsf{om}^\alpha(n)$. Notice that an *ast* sequence is obtained because, since Creol code in the initial configuration cannot include future names, all future names n occurring in t must belong to $dom(\mathsf{om}^\alpha)$.

A.2. From executions to sets of Petri net markings

In this section we present the definition of the mapping M from executions $\alpha = C_0 \to \cdots \to C_k$ in the operational semantics of C_0 to sets of markings of P_{C_0}.

Intuitively, given an execution $\alpha = C_0 \to \cdots \to C_k$, the set of markings $M(\alpha)$ is constructed as follows. We consider all threads $n\langle t_n\rangle$ in the C_k configuration such that $n \notin \mathcal{C}^\alpha$, i.e. we disregard those whose execution is already terminated and the obtained future n has been already consumed. For each such thread n we fix a sequence $t'_n \in s^\alpha_{1,4}(t_n)$. Once such an overall sequence selection (determining a set of sequences, one for each of the above considered threads n) is performed, we build a corresponding marking of P_{C_0} by putting one token in each place $c'_n@c_n\langle \mathsf{om}^\alpha(s_5(t'_n))\rangle$, with $c'_n@c_n$ being the abstract identifier of the thread n (i.e. both c_n and c'_n are a, possibly tagged, $o.l$ pair).

Before presenting the formal definition of the mapping M, we need to introduce some auxiliary functions.

Function c (**c**aller) takes an execution β in the operational semantics and returns a partial function from thread names to thread names denoted with c^β: $\mathsf{c}^\beta(n) = n'$ means that, in β, the caller of thread n is the thread n'.

Definition A.3. Given an execution β we denote with c^β a partial function from thread names to thread names defined as follows, where $\alpha = C_0 \to \ldots \to C_k$ and $\alpha' = C_0 \to \ldots \to C_k \to C_{k+1}$:

- $\mathsf{c}^{C_0} = \emptyset$
- $\mathsf{c}^{\alpha'} = \mathsf{c}^\alpha \cup \{n_2 \to n_1\}$ if $C_k \xrightarrow{n_1,n_2,o.l} C_{k+1}$;
- $\mathsf{c}^{\alpha'} = \mathsf{c}^\alpha$ otherwise.

Function acc (**a**bstract **c**aller and **c**allee) yields the abstract identifier $c'_n@c_n$ of the thread n based on the execution α and the (tagged run-time statement) sequences $t'_{n'}$ and $t'_{n''}$ chosen for: the thread $n' = \mathsf{c}^\alpha(n)$ that called n and (in case $n' \neq \mathit{run}$) the thread $n'' = \mathsf{c}^\alpha(n')$ that called n'. In particular, c_n and c'_n are both in the form $o.l$ (untagged pair) or $o.l?$ (tagged pair) with $o.l$ being determined by applying function om^α defined in the previous section, i.e. by computing $\mathsf{om}^\alpha(n)$ and $\mathsf{om}^\alpha(n')$, respectively. Moreover, in order to establish if such pairs are tagged, the two sequences $t'_{n'}$ and $t'_{n''}$ are inspected for get of n and n', respectively: in the case get of n (n') is found (due to s_3 there is at most one such a get) and it is tagged, the pair c_n (c'_n, respectively) is tagged. In the following we assume $\mathsf{n}(C)$ to denote the set of names n of all threads in the configuration C.

Definition A.4. Let $\alpha = C_0 \to \cdots \to C_k$ be an execution of the operational semantics and assume t'_n, for each $n \in \mathsf{n}(C_k)$, to range over sequences such that $t'_n \in s^\alpha_{1,4}(t_n)$ with $n\langle t_n\rangle$ in C_k. The function acc^α is defined by:

- $\mathsf{acc}^\alpha_{\cdot,t'_{\mathit{run}}}(n) = \mathit{run}@\mathsf{om}^\alpha(n, t'_{\mathit{run}})$ if $\mathsf{c}^\alpha(n) = \mathit{run}$
- $\mathsf{acc}^\alpha_{t'_{\mathsf{c}^\alpha(n')},t'_{n'}}(n) = \mathsf{om}^\alpha(n', t'_{\mathsf{c}^\alpha(n')})@\mathsf{om}^\alpha(n, t'_{n'})$ if $\mathsf{c}^\alpha(n) = n' \wedge n' \neq \mathit{run}$

where $\mathsf{om}^\alpha(n,t) = \mathsf{om}^\alpha(n)?$ if t includes "$\mathsf{get}@n?$" or "$\mathsf{get}(n?, o)$"; $\mathsf{om}^\alpha(n,t) = \mathsf{om}^\alpha(n)$, otherwise.

We are now in a position to define the mapping function M.

As we already explained, given an $\alpha = C_0 \to \ldots \to C_k$, M returns a set of markings, where each marking is obtained by performing an overall selection of sequences $t'_n \in s^{\alpha}_{1,4}$, one for each thread $n \in \mathsf{n}(C_k)$.

In step 1, based on such a set of selected sequences, we single out, for each of the above threads n such that $n \notin \mathcal{C}^{\alpha}$, a place where to add one token. This means that no token is added for threads n whose execution is already terminated and the obtained future n has been already consumed: in this way we are consistent with the behavior of Petri net where tokens representing futures are removed upon consumption. In step 2 we, instead, add tokens corresponding to locks based on the current status of the objects in C_k.

Definition A.5. Let C_0 be an initial configuration and let $\alpha = C_0 \to \ldots \to C_k$ be an execution in the operational semantics. Let P_{C_0} be the Petri net associated to C_0. Let us assume t_n, for each $n \in \mathsf{n}(C_k)$, to be such that $n\langle t_n \rangle$ is in C_k. Once fixed a trace for each thread n of C_k, i.e. considered a $\{t'_n \mid n \in \mathsf{n}(C_k)\}$ such that $\forall n \in \mathsf{n}(C_k).\, t'_n \in s^{\alpha}_{1,4}(t_n)$, we build a marking m of P_{C_0} as follows:

1. for each $n \in \mathsf{n}(C_k)$ such that $n \notin \mathcal{C}^{\alpha}$ add one token in place
   (a) $\mathit{run}@\mathit{run}\langle \mathsf{om}^{\alpha}(s_5(t'_{\mathit{run}}))\rangle$ if $n = \mathit{run}$
   (b) $\mathsf{acc}^{\alpha}_{\_,t'_{\mathit{run}}}(n)\langle \mathsf{om}^{\alpha}(s_5(t'_n))\rangle$ if $\mathsf{c}^{\alpha}(n) = \mathit{run}$
   (c) $\mathsf{acc}^{\alpha}_{t'_{\mathsf{c}^{\alpha}(n')},t'_{n'}}(n)\langle \mathsf{om}^{\alpha}(s_5(t'_n))\rangle$ if $\mathsf{c}^{\alpha}(n) = n' \wedge n' \neq \mathit{run}$
2. for each $o[c, F, L]$ in C_k, add one token in place o if $L = \bot$.

Let $M(\alpha)$ be the set of all possible markings built in this way.

Notice that the above is well-defined in that a finite set of tokens is produced for each marking in $M(\alpha)$. This is obvious because the number of objects and threads in a configuration (which is reached after a finite number of steps) is always finite.

We now show that each marking in $M(C_0 \to \ldots \to C_k)$ is reachable from the initial marking of the Petri net P_{C_0}. First, we present a technical lemma that we will use, as the induction step, to prove the above by induction on the length k.

Lemma A.6. Let $C_0 \to \cdots \to C_{k+1}$, with $k \geq 0$, be an execution in the operational semantics of the initial configuration C_0. For each marking $m_{k+1} \in M(C_0 \to \cdots \to C_{k+1})$ there is a marking $m_k \in M(C_0 \to \cdots \to C_k)$ such that m_{k+1} is reachable from m_k.

Proof: The proof proceeds by case analysis on the operational semantics rule used to infer the last transition $C_k \xrightarrow{\lambda} C_{k+1}$.

We first list three general properties useful to deal with several cases in our case analysis. Let $\alpha = C_0 \to \cdots \to C_k$ and $\alpha' = C_0 \to \cdots \to C_{k+1}$:

(f1) the functions $s^{\alpha'}_3$ and $\mathcal{C}^{\alpha'}$ change w.r.t. s^{α}_3 and $\mathcal{C}^{\alpha}$, respectively, if and only if the last transition consumes one future;

(f2) the functions $\mathsf{om}^{\alpha'}$, $\mathsf{c}^{\alpha'}$ and $\mathsf{acc}^{\alpha'}$ changes w.r.t. om^{α}, c^{α} and acc^{α}, respectively, if and only if the last transition creates one future;

(f3) all the functions $s_3^{\alpha'}$, $\mathcal{C}^{\alpha'}$, $\mathsf{om}^{\alpha'}$, $\mathsf{c}^{\alpha'}$ and $\mathsf{acc}^{\alpha'}$ are left unchanged if the last transition does not consume or create a future.

We are now ready to report our case analysis: each case is discussed in a separate paragraph starting with the name of the possible rule used to infer the last transition $C_k \xrightarrow{\lambda} C_{k+1}$.

RED Rule Case. In this case $C_k = C'_k \parallel n\langle \mathsf{let}\, x{:}T = v\, \mathsf{in}\, t\rangle$ and $C_{k+1} = C'_k \parallel n\langle t[v/x]\rangle$. Notice that, by definition of s_1, for all t', T', v' and x' we have that $s_1(t'[v'/x']) = s_1(\mathsf{let}\, x'{:}T' = v'\, \mathsf{in}\, t')$. Recall condition (f3), then:

$$\begin{aligned} s_{1,4}^{\alpha'}(t[v/x]) &= s_{1,4}^{\alpha'}(\mathsf{let}\, x{:}T = v\, \mathsf{in}\, t) \\ &= s_{1,4}^{\alpha}(\mathsf{let}\, x{:}T = v\, \mathsf{in}\, t) \end{aligned}$$

Notice that C'_k is shared between C_k and C_{k+1}. In addition, by (f3), $\mathcal{C}^{\alpha'}$, $\mathsf{om}^{\alpha'}$, $\mathsf{acc}^{\alpha'}$ and $\mathsf{c}^{\alpha'}$ are the same as $\mathcal{C}^{\alpha}$, om^{α}, acc^{α}, and c^{α}, respectively. Hence we can conclude $M(\alpha') = M(\alpha)$.

COND$_1$ Rule Case. In this case $C_k = C'_k \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{if}\, v = v\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \mathsf{in}\, t\rangle$ and $C_{k+1} = C'_k \parallel n\langle \mathsf{let}\, x{:}T = e_1\, \mathsf{in}\, t\rangle$. Once again, taking into account (f3), we have:[4]

$$\begin{aligned} & s_{1,4}^{\alpha'}(\mathsf{let}\, x{:}T = e_1\, \mathsf{in}\, t) \\ & \subseteq s_4\, s_3^{\alpha'}\, (s_2\, s_1(\mathsf{let}\, x{:}T = e_1\, \mathsf{in}\, t) \cup s_2\, s_1(\mathsf{let}\, x{:}T = e_2\, \mathsf{in}\, t)) \\ & = s_{1,4}^{\alpha}(\mathsf{let}\, x{:}T = \mathsf{if}\, v = v\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \mathsf{in}\, t) \end{aligned}$$

Again, C'_k is shared between C_k and C_{k+1}. In addition, by (f3), $\mathcal{C}^{\alpha'}$, $\mathsf{om}^{\alpha'}$, $\mathsf{acc}^{\alpha'}$ and $\mathsf{c}^{\alpha'}$ are the same as $\mathcal{C}^{\alpha}$, om^{α}, acc^{α}, and c^{α}, respectively. Hence we can conclude $M(\alpha') \subseteq M(\alpha)$.

COND$_2$ Rule Case. Analogous to the above case.

FUT Rule Case. Let $C_k = C'_k \parallel n_1\langle \mathsf{let}\, x{:}T = o.l(\vec{v})\, \mathsf{in}\, t\rangle$ and $C_{k+1} = C'_k \parallel n_1\langle \mathsf{let}\, x{:}T = n_2\, \mathsf{in}\, t\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = \mathsf{grab}(o); t'\, \mathsf{in}\, \mathsf{release}(o); x\rangle$ with thread $t' = t''[o/self][\vec{v}/\vec{x}]$ assuming that $n'[\![F, M]\!]$ occurs in C'_k with n' class of the object o and $M.l = \varsigma(self{:}T).\lambda(\vec{x}{:}\vec{T}).t''$, then:

$$\begin{aligned} s_{1,4}^{\alpha'}(\mathsf{let}\, x{:}T = n_2\, \mathsf{in}\, t) &= s_4\, s_3^{\alpha'}\, s_2\, s_1(\mathsf{let}\, x{:}T = n_2\, \mathsf{in}\, t) \\ &= s_4\, s_3^{\alpha'}\, s_2\, s_1(t[n_2/x]) \end{aligned}$$

[4]Here and in the following, for simplicity, we denote function composition "$s \circ s'$" simply by juxtaposition "$s\ s'$".

We proceed with thread $n_2\langle t_2\rangle$, i.e. t_2 is the code of n_2. Notice that t_2 does not include thread names because t_2 has not started to run. This implies that $s^{\alpha'}_{1,4}(t_2) = s^{\emptyset}_{1,4}(t_2)$. Moreover t'' cannot include $\mathsf{get}@x$ or $\mathsf{claim}@(x, n')$ with x being free and in the formal parameters $\vec{x}$ (the restrictions we considered in the language imply that passed parameters cannot be used as futures). Hence $s_1(t''[o/self][\vec{v}/\vec{x}]) = s_1(t''[o/self])$. Hence

$$\begin{aligned}
&s^{\alpha'}_{1,4}(\mathsf{let}\, x{:}T = \mathsf{grab}(o); t' \,\mathsf{in}\, \mathsf{release}(o); x)\\
&= s_4\, s^{\alpha'}_3\, s_2\, s_1(\mathsf{let}\, x{:}T = \mathsf{grab}(o); t' \,\mathsf{in}\, \mathsf{release}(o); x)\\
&= s_4\, s^{\emptyset}_3\, s_2\, s_1(\mathsf{let}\, x{:}T = \mathsf{grab}(o); t''[o/self][\vec{v}/\vec{x}] \,\mathsf{in}\, \mathsf{release}(o); x)\\
&= s_4\, s_3\, s_2(\mathsf{grab}(o); s_1(t''[o/self]); \mathsf{release}(o))\\
&= \mathsf{grab}(o); s_4\, s_3 s_2\, s_1(t''[o/self]); \mathsf{release}(o)
\end{aligned}$$

On the other hand:

$$\begin{aligned}
s^{\alpha}_{1,4}(\mathsf{let}\, x{:}T = o.l(\vec{v}) \,\mathsf{in}\, t) &= s_4\, s^{\alpha}_3\, s_2\, s_1(\mathsf{let}\, x{:}T = o.l(\vec{v}) \,\mathsf{in}\, t)\\
&= s_4\, (\mathsf{let}\, x{=}o.l; s^{\alpha}_3\, s_2\, s_1(t))\\
&= \mathsf{let}\, x{=}o.l?; \mathcal{T} \cup \mathsf{let}\, x{=}o.l; \mathcal{T}'
\end{aligned}$$

where $\mathcal{T}$ contains the traces of $s_4 s^{\alpha}_3\, s_2\, s_1(t)$ that include at least a get or claim on x which is tagged with ? and $\mathcal{T}' = s_4 s^{\alpha}_3\, s_2\, s_1(t) \setminus \mathcal{T}$.

Now we proceed with the thread name abstractions. Notice that, since all threads in C_{k+1} already occurred C_k apart from n_2, we have that, excluding n_2 (for which acc^{α} is not defined), acc^{α} and $\mathsf{acc}^{\alpha'}$ coincide.

Given a marking $m' \in M(\alpha')$, we have that m' is composed of the tokens generated by C'_k plus two more tokens: one, generated by thread n_1, in a place $c@c'\langle \mathsf{om}^{\alpha'}\, s_5(b')\rangle$, with $b' \in s^{\alpha'}_{1,4}(\mathsf{let}\, x{:}T = n_2 \,\mathsf{in}\, t)$, and one, generated by thread n_2, in a place $c'@o.l^{+}\langle \mathsf{om}^{\alpha'}\, s_5(b''')\rangle$, with $+$ being ? if and only if in b' there is a get or claim on n_2 tagged with "?" and $b''' \in s^{\alpha'}_{1,4}(\mathsf{let}\, x{:}T = \mathsf{grab}(o);\, t' \,\mathsf{in}\, \mathsf{release}(o); x) = \mathsf{grab}(o); s_4\, s_3 s_2\, s_1(t''[o/self]); \mathsf{release}(o)$. That is, since b''' does not include future names, a place $c'@o.l^{+}\langle w\rangle$ with $w \in \mathsf{grab}(o); \mathsf{ST}(t''[o/self]); \mathsf{release}(o)$. We now consider a marking m of $M(\alpha)$ such that m is composed of the tokens generated by C'_k, by selecting the same traces (hence getting tokens in the same places), plus one more token generated by the thread n_1, by selecting the trace $\mathsf{let}\, x{=}o.l^{+}; b''$ of $s^{\alpha}_{1,4}(\mathsf{let}\, x{:}T = o.l(\vec{v}) \,\mathsf{in}\, t)$ with $b'' = b'[x/n_2]$. Hence, such token is in the place $c@c'\langle \mathsf{om}^{\alpha}\, s_5(\mathsf{let}\, x = o.l^{+}; b'')\rangle$ because we considered the same traces for (calling) threads in C'_k (and acc^{α} and $\mathsf{acc}^{\alpha'}$ coincide on n_1).

It is easy to see that m' is reachable from m through a call transition, i.e. a transition in P_{C_0} of kind *T-2* (see Definition 4.9). This because, since $n_2 \to o.l \in \mathsf{om}^{\alpha'}$ we have $\mathsf{om}^{\alpha'}\, s_5(b') = \mathsf{om}^{\alpha}\, s_5(b'[o.l/n_2]) = \mathsf{om}^{\alpha}\, s_5(b)$ with $b = b'[o.l/n_2]$. On the other hand, $\mathsf{om}^{\alpha}\, s_5(\mathsf{let}\, x{=}o.l^{+}; b'') = \mathsf{let}\, o.l^{+}; \mathsf{om}^{\alpha}\, s_5(b''[o.l/x]) = \mathsf{let}\, o.l^{+}; \mathsf{om}^{\alpha}\, s_5(b)$.

GET$_2$ Rule Case. In this case $C_k = C'_k \parallel n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = \mathsf{get}@n_1 \,\mathsf{in}\, t\rangle$ and $C_{k+1} = C'_k \parallel n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = v \,\mathsf{in}\, t\rangle$. We observe that by rule $Claim_2$ of the operational semantics the $\mathsf{get}@n_1$

primitive has been previously generated in a configuration in which the n_1 future was not available to be consumed. Hence, (since n_2 is the only thread where future name n_1 may occur) before this transition, the future was not yet consumed, i.e. $n_1 \notin \mathcal{C}^\alpha$. Moreover, $\mathcal{C}^{\alpha'} = \mathcal{C}^\alpha \cup \{n_1\}$. Notice that t cannot include $\mathsf{get@}(x, n')$, $\mathsf{claim@}(x, n')$ or runtime $\mathsf{get@}x$ because x is instantiated with a returned value, and we cannot use it as a future; hence $s_1(t[v/x]) = s_1(t)$. On the one hand,

$$\begin{aligned} s_{1,4}^{\alpha'}(\mathsf{let}\, x{:}T = v\, \mathsf{in}\, t) &= s_4\, s_3^{\alpha'}\, s_2\, s_1(\mathsf{let}\, x{:}T = v\, \mathsf{in}\, t) \\ &= s_4\, s_3^{\alpha'}\, s_2\, s_1(t[v/x]) \\ &= s_4\, s_3^{\alpha'}\, s_2\, s_1(t) \\ &= s_4\, s_3^{\mathcal{C}^\alpha \cup \{n_1\}}\, s_2\, s_1(t) \end{aligned}$$

On the other hand,

$$\begin{aligned} s_{1,4}^{\alpha}(\mathsf{let}\, x{:}T = \mathsf{get@}n_1\, \mathsf{in}\, t) &= s_4\, s_3^{\alpha}(\mathsf{get@}n_1; s_2\, s_1(t)) \\ &= s_4(\mathsf{get@}n_1; s_3^{\mathcal{C}^\alpha \cup \{n_1\}}\, s_2\, s_1(t)) \\ &= (\mathsf{get@}n_1?; s_3^{\mathcal{C}^\alpha \cup \{n_1\}}\, s_2\, s_1(t) \\ &\quad \cup\ \mathsf{get@}n_1; s_4\, s_3^{\mathcal{C}^\alpha \cup \{n_1\}}\, s_2\, s_1(t)) \end{aligned}$$

We are now ready to prove that each marking in $M(\alpha')$ is reachable from a marking taken from $M(\alpha)$. Let m' be a marking in $M(\alpha')$: m' is composed of a token in $c_1'@c_2'\langle \mathsf{om}^{\alpha'}(s_5(t'))\rangle$, for some trace $t' \in s_4\, s_3^{\mathcal{C}^\alpha \cup \{n_1\}}\, s_2\, s_1(t)$ extracted from the thread $n_2\langle \mathsf{let}\, x{:}T = v\, \mathsf{in}\, t\rangle$ with c_1', c_2' depending on $\mathsf{acc}^{\alpha'}$, plus the tokens deriving from the rest of the configuration C_k'. We now find a marking m in $M(\alpha)$ that reaches m'. Such a marking is obtained by considering a token in the place $c_1@c_2\langle \mathsf{om}^{\alpha}(s_5(\mathsf{get@}n_1; t'))\rangle$, for some c_1, c_2 depending on acc^{α} (such token is added by considering the trace $\mathsf{get@}n_1; t'$ belonging to $\mathsf{get@}n_1; s_4\, s_3^{\mathcal{C}^\alpha \cup \{n_1\}}\, s_2\, s_1(t) \subseteq s_{1,4}^{\alpha}(\mathsf{let}\, x{:}T = \mathsf{get@}n_1\, \mathsf{in}\, t)$), a token in $c_2@c_3\langle\rangle$ (added by $n_1\langle v\rangle$ because $n_1 \notin \mathcal{C}^\alpha$ as observed above) plus the tokens deriving from the rest of the configuration C_k' for which we assume to tag the same traces of the threads in C_k' as those considered for generating the above marking m'. By (f3) we have that $\mathsf{acc}^{\alpha} = \mathsf{acc}^{\alpha'}$, thus implying $c_1' = c_1$ and $c_2' = c_2$ (thanks also to the selection of the same traces for the threads in C_k'), as well as $\mathsf{om}^{\alpha} = \mathsf{om}^{\alpha'}$. This guarantees that the token of m' in the place $c_1'@c_2'\langle \mathsf{om}^{\alpha'}(s_5(t'))\rangle$ can be generated by the Petri net transition that consumes the tokens in the places $c_1@c_2\langle \mathsf{om}^{\alpha}(s_5(\mathsf{get@}n_1; t'))\rangle = c_1@c_2\langle \mathsf{om}^{\alpha}(\mathsf{get@}n_1; s_5(t'))\rangle = c_1@c_2\langle \mathsf{get@}o_{n_1}.l_{n_1}; \mathsf{om}^{\alpha}(s_5(t'))\rangle$ (with $n_1 \to o_{n_1}.l_{n_1} \in \mathsf{om}^{\alpha}$, i.e. o_{n_1} and l_{n_1} are the object and method of the thread n_1) and $c_2@c_3\langle\rangle$ in the marking m.

GET$_1$ Rule Case. Let $C_k = C_k' \parallel n_1\langle v\rangle \parallel n_2\langle x{:}T = \mathsf{get@}(n_1, o_{n_2})\, \mathsf{in}\, t\rangle$ and $C_{k+1} = C_k' \parallel n_1\langle v\rangle \parallel n_2\langle x{:}T = v\, \mathsf{in}\, t\rangle$. We have to analyze two cases, in the first one it is the first consumption of the future (i.e. $n_1 \notin \mathcal{C}^\alpha$), in the second one it is not (i.e. $n_1 \in \mathcal{C}^\alpha$). In the first case, the proof proceeds as in the case of the runtime get above. In the second one, we have that

$$s_{1,4}^{\alpha'}(\mathsf{let}\, x{:}T = v\, \mathsf{in}\, t) = s_4\, s^{\mathcal{C}_3^\alpha}\, s_2\, s_1(t)$$

because $\mathcal{C}^{\alpha'} = \mathcal{C}^{\alpha}$ and

$$\begin{aligned} s^{\alpha}_{1,4}(\mathsf{let}\, x{:}T = \mathsf{get@}(n_1, o_{n_2})\, \mathsf{in}\, t) &= s_4\, s^{\alpha}_3(\mathsf{get@}n_1; s_2\, s_1(t)) \\ &= s_4 s_3^{\mathcal{C}^{\alpha}}\, s_2\, s_1(t) \end{aligned}$$

because $n_1 \in \mathcal{C}^{\alpha}$. By (f3) we have that $\mathsf{acc}^{\alpha} = \mathsf{acc}^{\alpha'}$, $\mathcal{C}^{\alpha} = \mathcal{C}^{\alpha'}$ and $\mathsf{om}^{\alpha} = \mathsf{om}^{\alpha'}$, hence $M(\alpha) = M(\alpha')$.

CLAIM$_2$ Rule Case. In this case we have

$$\begin{aligned} C_k &= C'_k \parallel n_1\langle t_1\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = \mathsf{claim@}(n_1, o)\, \mathsf{in}\, t_2\rangle \\ C_{k+1} &= C'_k \parallel n_1\langle t_1\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = \mathsf{release}(o); \mathsf{get@}n_1\, \mathsf{in}\, \mathsf{grab}(o); t_2\rangle \end{aligned}$$

where t_1 is not a value. This implies that it is the first claim of the future, i.e., $n_1 \notin \mathcal{C}^{\alpha}$. Hence, on the one hand, we have:

$$\begin{aligned} & s^{\alpha}_{1,4}(\mathsf{let}\, x{:}T = \mathsf{claim@}(n_1, o)\, \mathsf{in}\, t_2) \\ &= s_4\, s^{\alpha}_3\, s_2\, s_1(\mathsf{let}\, x{:}T = \mathsf{claim@}(n_1, o)\, \mathsf{in}\, t_2) \\ &= s_4\, s^{\alpha}_3(\mathsf{claim@}(n_1, o); s_2\, s_1(t_2)) \\ &= s_4\, (\mathsf{release}(o); \mathsf{get@}n_1; \mathsf{grab}(o); s_3^{\mathcal{C}^{\alpha} \cup \{n_1\}}\, s_2\, s_1(t_2)) \end{aligned}$$

On the other hand since, by $(f1)$ we have $s^{\alpha'}_3 = s^{\alpha}_3$, we have:

$$\begin{aligned} & s^{\alpha'}_{1,4}(\mathsf{let}\, x{:}T = \mathsf{release}(o); \mathsf{get@}n_1\, \mathsf{in}\, \mathsf{grab}(o); t_2) \\ &= s_4\, s^{\alpha}_3\, s_2\, s_1(\mathsf{let}\, x{:}T = \mathsf{release}(o); \mathsf{get@}n_1\, \mathsf{in}\, \mathsf{grab}(o); t_2) \\ &= s_4\, s^{\alpha}_3\, (\mathsf{release}(o); \mathsf{get@}n_1; \mathsf{grab}(o); s_2\, s_1(t_2)) \\ &= s_4\, (\mathsf{release}(o); \mathsf{get@}n_1; \mathsf{grab}(o); s_3^{\mathcal{C}^{\alpha} \cup \{n_1\}}\, s_2\, s_1(t_2)) \end{aligned}$$

By (f3) we have that $\mathsf{acc}^{\alpha} = \mathsf{acc}^{\alpha'}$, $\mathcal{C}^{\alpha} = \mathcal{C}^{\alpha'}$ and $\mathsf{om}^{\alpha} = \mathsf{om}^{\alpha'}$, hence $M(\alpha) = M(\alpha')$.

CLAIM$_1$ Rule Case. We have

$$\begin{aligned} C_k &= C'_k \parallel n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = \mathsf{claim@}(n_1, o)\, \mathsf{in}\, t\rangle \\ C_{k+1} &= C'_k \parallel n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\, x{:}T = v\, \mathsf{in}\, t\rangle \end{aligned}$$

As for the GET$_1$ rule, we have to analyze two cases, in the first one it is the first consumption of the future (i.e. $n_1 \notin \mathcal{C}^{\alpha}$), in the second one it is not (i.e. $n_1 \in \mathcal{C}^{\alpha}$). In the latter case, we proceed exactly as in the GET$_1$ rule.

So we now analyze the case where it is the first consumption of the future, i.e. $n_1 \notin \mathcal{C}^{\alpha}$ and $\mathcal{C}^{\alpha'} = \mathcal{C}^{\alpha} \cup \{n_1\}$. Notice that t cannot include $\mathsf{get@}(x, n')$, $\mathsf{claim@}(x, n')$ or runtime $\mathsf{get@}x$ because

x is instantiated with a returned value, and it cannot be used as a future; hence $s_1(t[v/x]) = s_1(t)$. On the one hand,

$$\begin{aligned} s_{1,4}^{\alpha'}(\mathsf{let}\, x{:}T = v\, \mathsf{in}\, t) &= s_4\, s_3^{\alpha'}\, s_2\, s_1(\mathsf{let}\, x{:}T = v\, \mathsf{in}\, t) \\ &= s_4\, s_3^{\alpha'}\, s_2\, s_1(t[v/x]) \\ &= s_4\, s_3^{\alpha'}\, s_2\, s_1(t) \\ &= s_4\, s_3^{\mathcal{C}^\alpha \cup \{n_1\}}\, s_2\, s_1(t) \end{aligned}$$

On the other hand,

$$\begin{aligned} & s_{1,4}^{\alpha}(\mathsf{let}\, x{:}T = \mathsf{claim}@(n_1, o)\, \mathsf{in}\, t) \\ & = s_4\, s_3^{\alpha}(\mathsf{claim}@(n_1, o); s_2\, s_1(t)) \\ & = s_4(\mathsf{release}(o); \mathsf{get}@n_1; \mathsf{grab}(o); s_3^{\mathcal{C}^\alpha \cup \{n_1\}}\, s_2\, s_1(t)) \\ & = (\mathsf{release}(o); \mathsf{get}@n_1?; \mathsf{grab}(o); s_3^{\mathcal{C}^\alpha \cup \{n_1\}}\, s_2\, s_1(t) \\ & \quad \cup\ \mathsf{release}(o); \mathsf{get}@n_1; \mathsf{grab}(o); s_4\, s_3^{\mathcal{C}^\alpha \cup \{n_1\}}\, s_2\, s_1(t)) \end{aligned}$$

We are now ready to prove that each marking in $M(\alpha')$ is reachable from a marking taken from $M(\alpha)$. Let m' be a marking in $M(\alpha')$: m' is composed of a token in $c_1'@c_2'\langle \mathsf{om}^{\alpha'}(s_5(t'))\rangle$, for some trace $t' \in s_4\, s_3^{\mathcal{C}^\alpha \cup \{n_1\}}\, s_2\, s_1(t)$ extracted from the thread $n_2\langle \mathsf{let}\, x{:}T = v\, \mathsf{in}\, t\rangle$ with c_1', c_2' depending on $\mathsf{acc}^{\alpha'}$, plus the tokens deriving from the rest of the configuration C_k'. We now find a marking m in $M(\alpha)$ that reaches m'. Such a marking is obtained by considering a token in the place $c_1@c_2\langle \mathsf{om}^{\alpha}(s_5(\mathsf{release}(o); \mathsf{get}@n_1; \mathsf{grab}(o); t'))\rangle$, for some c_1, c_2 depending on acc^{α} (such token is added by considering the trace $\mathsf{release}(o); \mathsf{get}@n_1; \mathsf{grab}(o); t'$ belonging to $\mathsf{release}(o); \mathsf{get}@n_1; \mathsf{grab}(o); s_4\, s_3^{\mathcal{C}^\alpha \cup \{n_1\}}\, s_2\, s_1(t) \subseteq s_{1,4}^{\alpha}(\mathsf{let}\, x{:}T = \mathsf{claim}@(n_1, o)\, \mathsf{in}\, t)$), a token in $c_2@c_3\langle\rangle$ (added by $n_1\langle v\rangle$ because $n_1 \notin \mathcal{C}^\alpha$ as observed above) plus the tokens deriving from the rest of the configuration C_k' for which we assume to tag the same traces of the threads in C_k' as those considered for generating the above marking m'. By (f3) we have that $\mathsf{acc}^{\alpha} = \mathsf{acc}^{\alpha'}$, thus implying $c_1' = c_1$ and $c_2' = c_2$ (thanks also to the selection of the same traces for the threads in C_k'), as well as $\mathsf{om}^{\alpha} = \mathsf{om}^{\alpha'}$. This guarantees that the token of m' in the place $c_1'@c_2'\langle \mathsf{om}^{\alpha'}(s_5(t'))\rangle$ can be generated by the Petri net transitions *T-4*, *T-5*, and *T-3* that consume the tokens in the places

$$\begin{aligned} & c_1@c_2\langle \mathsf{om}^{\alpha}(s_5(\mathsf{release}(o); \mathsf{get}@n_1; \mathsf{grab}(o); t'))\rangle \\ & = c_1@c_2\langle \mathsf{om}^{\alpha}(\mathsf{release}(o); \mathsf{get}@n_1; \mathsf{grab}(o); s_5(t'))\rangle \\ & = c_1@c_2\langle \mathsf{release}(o); \mathsf{get}@o_{n_1}.l_{n_1}; \mathsf{grab}(o); \mathsf{om}^{\alpha}(s_5(t'))\rangle \end{aligned}$$

(with $n_1 \to o_{n_1}.l_{n_1} \in \mathsf{om}^{\alpha}$, i.e. o_{n_1} and l_{n_1} are the object and method of the thread n_1) and $c_2@c_3\langle\rangle$ in the marking m. Notice that during the execution of the *T-4*, *T-5*, and *T-3* transitions, *T-4* generates a token in the place "o" which is subsequently removed by *T-3*.

GRAB Rule Case. In this case $C_k = C'_k \parallel o[c, F, \bot] \parallel n\langle \mathsf{grab}(o); t\rangle$ and $C_{k+1} = C'_k \parallel o[c, F, \top] \parallel n\langle t\rangle$. Recall condition (f3), then:

$$\begin{aligned} s^{\alpha}_{1,4}(\mathsf{grab}(o); t) &= \mathsf{grab}(o); s^{\alpha}_{1,4}(t) \\ &= \mathsf{grab}(o); s^{\alpha'}_{1,4}(t) \end{aligned}$$

Notice that C'_k is shared between C_k and C_{k+1}; and C_k has the additional . In addition, by (f3), $\mathcal{C}^{\alpha'}$, $\mathsf{om}^{\alpha'}$, $\mathsf{acc}^{\alpha'}$ and $\mathsf{c}^{\alpha'}$ are the same as $\mathcal{C}^{\alpha}$, om^{α}, acc^{α}, and c^{α}, respectively. Therefore, to show that each marking $m' \in M(\alpha')$ is reachable from a marking $m \in M(\alpha)$ it is sufficient to consider the marking m obtained by selecting the same traces in C'_k, and for thread n the trace $\mathsf{grab}(o); t'$ with t' is the trace considered in m', plus the token in place "*o*" generated by $o[c, F, \bot]$. The marking m' (which, instead does not include the token in place "*o*") is reached from m by firing the transition ***T-3***.

RELEASE Rule Case. It is symmetric w.r.t. the above GRAB rule case.

The Other Cases. The LET, SUSPEND, FLOOKUP and FUPDATE rules are analogous to RED rule in that $M(\alpha') = M(\alpha)$. □

Theorem 4.11 (Marking Reachability). *Let $C_0 \to \cdots \to C_k$, with $k \geq 0$, be an execution in the operational semantics of the initial configuration C_0. Each marking in $M(C_0 \to \ldots \to C_k)$ is reachable from the initial marking of P_{C_0}.*

Proof: The proof is by induction on k. Concerning the base case, $k = 0$, we have that each marking in $M(C_0)$ is reachable from the initial marking m_0 (one token in the place *start* and one token in each of the object places) by one of the ***T-1*** transitions (see Definition 4.9). This because, considered t such that $\mathit{run}\langle t\rangle$ in C_0, $M(C_0)$ includes a marking composed by one token in the place $\mathit{run}@\mathit{run}\langle \mathsf{om}^{C_0}(s_5(t'))\rangle$ for each $t' \in s^{C_0}_{1,4}(t)$ and we have $\{\mathsf{om}^{C_0}(s_5(t')) \mid t' \in s^{C_0}_{1,4}(t)\} = \mathsf{ST}(t)$. For the induction case, we directly resort to Lemma A.6. □

B. Proof of Lemma 5.3

Lemma 5.3. *Let C_0 be an initial configuration and let $\alpha = C_0 \to \ldots \to C_k$ be an execution in the operational semantics. Let P_{C_0} be the Petri net associated to C_0. If C_k is an extended deadlock then $M(\alpha)$ includes an extended deadlock marking of P_{C_0}.*

Proof: Being an extended deadlock, configuration C_k has a set of threads N such that each $n \in N$ satisfies one of the following conditions:

(b1) $n\langle \mathsf{let}\, x{:}T = \mathsf{get}@(n', o)\, \mathsf{in}\, t\rangle$ or $n\langle \mathsf{let}\, x{:}T = \mathsf{get}@n'\, \mathsf{in}\, t\rangle$ and $n' \in N$;

(b2) $n\langle \mathsf{let}\, x{:}T = \mathsf{grab}(o)\, \mathsf{in}\, t\rangle$ and there is $n' \in N$ s.t. $n'\langle \mathsf{let}\, y{:}T = \mathsf{get}@(n'', o)\, \mathsf{in}\, t'\rangle$.

Condition (b1) is the rewriting of "n is waiting for n'" and condition (b2) formalizes "n is waiting for object o that is blocked by another thread in the deadlock".

Let us assume t_n, for each $n \in \mathsf{n}(C_k)$, to be such that $n\langle t_n\rangle$ is in C_k. We now fix a trace for each thread n of C_k, i.e. define $\{t'_n \mid n \in \mathsf{n}(C_k)\}$ such that $\forall n \in \mathsf{n}(C_k).\, t'_n \in s^{\alpha}_{1,4}(t_n)$, as follows. For the threads $n \in N$, we select a trace t'_n in the following way:

(c1) if $n\langle \mathsf{let}\, x{:}T = \mathsf{get@}(n', o)\, \mathsf{in}\, t\rangle$ select a trace $t'_n = \mathsf{get@}(n', o)?; \hat{t} \in s^{\alpha}_{1,4}(\mathsf{let}\, x{:}T = \mathsf{get@}(n', o)\ \mathsf{in}\, t)$;

(c2) if $n\langle \mathsf{let}\, x{:}T = \mathsf{get@}n'\, \mathsf{in}\, t\rangle$ select a trace $t'_n = \mathsf{get@}n'?; \hat{t} \in s^{\alpha}_{1,4}(\mathsf{let}\, x{:}T = \mathsf{get@}n'\, \mathsf{in}\, t)$;

(c3) if $n\langle \mathsf{let}\, x : T = \mathsf{grab}(o)\, \mathsf{in}\, t\rangle$ select a trace $t'_n = \hat{t} \in s^{\alpha}_{1,4}(\mathsf{let}\, x : T = \mathsf{grab}(o)\, \mathsf{in}\, t)$ such that $\hat{t}$ is ?-free.

For all threads $n \in \mathsf{n}(C_k) - N$ we take a trace $t'_n \in s^{\alpha}_{1,4}(t_n)$ such that t'_n is ?-free (this trace exists by s_4 definition).

Let P_{C_0} be the Petri net associated to C_0. Once fixed the above set of traces for the threads in C_k, we have that $M(\alpha)$ includes the marking built as follows:

1. for each $n \in \mathsf{n}(C_k)$ such that $n \notin \mathcal{C}^{\alpha}$, add a token in
   (a) $\mathit{run}@\mathit{run}\langle \mathsf{om}^{\alpha}(s_5(t'_{\mathit{run}}))\rangle$ if $n = \mathit{run}$
   (b) $\mathsf{acc}^{\alpha}_{\cdot, t'_{\mathit{run}}}(n)\langle \mathsf{om}^{\alpha}(s_5(t'_n))\rangle$ if $\mathsf{c}^{\alpha}(n) = \mathit{run}$
   (c) $\mathsf{acc}^{\alpha}_{t'_{\mathsf{c}^{\alpha}(n')}, t'_{n'}}(n)\langle \mathsf{om}^{\alpha}(s_5(t'_n))\rangle$ if $\mathsf{c}^{\alpha}(n) = n' \wedge n' \neq \mathit{run}$
2. for each $o[c, F, L]$ in C_k, add a token in place o if $L = \bot$.

We now show that this is an extended deadlock marking. Let D be the set of places where we added tokens according to item 1. above, whenever applied to the threads $n \in N$. Trivially each of these places has at least one token. Moreover, conditions (c1–3) guarantee that each place in is in the form $c@c'\langle t\rangle$, with t being one of "$\mathsf{get@}(o''.l''?, o'); t'$", "$\mathsf{get@}o''.l''?; t'$" or "$\mathsf{grab}(o'); t'$". If $c = o.l?$ then 1(c) and the fact that traces with tagged get are selected, by (c1–2), only for threads in N guarantee that there is $p' \in D$ in the form $c''@c'''\langle \mathsf{get@}(o.l?, o'''); t''\rangle$ or $c''@c'''\langle \mathsf{get@}o.l?; t''\rangle$. If $c' = o'.l'?$ then, similarly, 1(b–c) and the fact that traces with tagged get are selected, by (c1–2), only for threads in N guarantee that there is $p' \in D$ in the form $c''@c\langle \mathsf{get@}(o'.l'?, o'''); t''\rangle$ or $c''@c\langle \mathsf{get@}o'.l'?; t''\rangle$. Finally, if $t = \mathsf{grab}(o'); t'$ then (c3) and (b2) guarantee that t' is ?-free and that there is $p' \in D$ in the form $c@c'\langle \mathsf{get@}(o'''.l'''?, o'); t''\rangle$. We now conclude by showing that any place $p \notin D$ in which item 1 or 2 added a token are such that

- p is *?-free*,
- p is in the form $c@c'\langle t\rangle$, with c', t being *?-free* and $c = o.l?$ for some o,l such that there is $p' \in D$ in the form $c''@o.l?\langle t'\rangle$.

For item 2 this is obvious (p is *?-free*). We now analyse item 1: such tokens must be generated by threads $n\langle t_n\rangle$ with $n \notin N$. This guarantee that the selected t'_n is *?-free*. We now show by contradiction that also c' is *?-free*. Assume that $c' = o.l?$; this implies, by 1(b-c), that $\mathsf{c}^\alpha(n) \in N$ because the corresponding trace has been tagged. By (b1) we would have that also $n \in N$ that contradicts the hypothesis. Finally, we observe that if $c = o.l?$ then by 1(c) $n' = \mathsf{c}^\alpha(\mathsf{c}^\alpha(n)) \in N$ and the corresponding trace has been tagged and starts with a get on an n'' which is abstractly identified with $o.l?$. By (b1) we have that $n'' \in N$ hence by 1(b-c) such thread adds a token in a place $p' \in D$ in the form $c''@o.l?\langle t'\rangle$. □