Translating Active Objects into Colored Petri Nets for Communication Analysis*

Anastasia Gkolfi, Crystal Chang Din, Einar Broch Johnsen, Martin Steffen, and Ingrid Chieh Yu

Department of Informatics, University of Oslo, Oslo, Norway
{natasa,crystald,einarj,msteffen,ingridcy}@ifi.uio.no

Abstract. Actor-based languages attract attention for their ability to scale to highly parallel architectures. Active objects combine the asynchronous communication of actors with object-oriented programming by means of asynchronous method calls and synchronization on futures. However, the combination of asynchronous calls and synchronization introduces communication cycles which lead to a form of communication deadlock. This paper addresses such communication deadlocks for ABS, a formally defined active object language which additionally supports cooperative scheduling to express complex distributed control flow, using first-class futures and explicit process release points. Our approach is based on a translation of the semantics of ABS into colored Petri nets, such that a particular program corresponds to a marking of this net. We prove the soundness of this translation and demonstrate by example how the implementation of this net can be used to analyze ABS programs with respect to communication deadlock.

1 Introduction

The Actor model [1,2] of concurrency is attracting increasing attention for their *decoupling* of control flow and communication. This decoupling enables both scalability (as argued with the Erlang programming language [3] and Scala's actor model [14]) and compositional reasoning [11]. Actors are independent units of computation which exchange messages and execute local code sequentially. Instead of pushing the current procedure (or method activation) on the control stack when sending a message as in thread-based concurrency models, messages are sent asynchronously, without any transfer of control between the actors. In the actor model, a message triggers the execution of a method body in the target actor, but a reply to the message is not directly supported. Extending the basic actor model, active object languages (e.g., [8, 18]), which combine actor-like communication with object orientation, use so-called futures to reintroduce synchronization by combining asynchronous message sending with the call and reply structure of method calls. A future can be seen as a mailbox from which a reply may be retrieved, such that the synchronization is decoupled from message

* The work was partially supported by the Norwegian Research Council under the CUMULUS project.

sending and associated with fetching the reply from a method call. The caller synchronizes with the existence of a reply from a method call by performing a *blocking* get-operation on the future associated with the call. However, this synchronization may lead to complex dependency cycles in the communication chain of a program, and gives rise to a form of deadlock with a set of mutually blocked objects. This situation is often called a *communication deadlock* [9].

This paper addresses the problem of communication deadlock for the active object language ABS [18,19]. ABS is characteristic in that it supports *cooperative concurrency* in the active objects. Cooperative concurrency allows the execution of a method body to be suspended at explicit points in the code, for example by testing whether a future has received a value. Cooperative concurrency leads to a form of local race-free interleaving for concurrently executing active objects, which allows more execution traces than in standard active objects. Our approach to tackle the callback problem for ABS is based on a translation of the formal semantics of ABS into colored Petri nets (CPN) [17]. Petri nets provide a basic model of concurrency, causality, and synchronization [22, 25], which has previously been used to analyze communication patterns and deadlock, e.g., [10, 15]. CPNs extend the basic Petri net model with support for modeling data. In contrast to previous work, we do not produce a particular Petri net for each program to be analyzed. Instead, we provide an encoding and implementation of the formal semantics of ABS itself as a net, and use colored tokens in this net to encode the program. Consequently, the number of places in the net is independent of the size of a program, and different programs are captured by different markings of the net. For example, this approach allows us to capture dynamic object creation by firing transitions in the net.

The main contributions of this paper are:

- a deep encoding of the formal semantics of ABS in CPNs;
- a translation of concrete ABS programs into markings of this net;
- a soundness proof for the translation from ABS to CPN; and
- an example demonstrating how to analyze communication deadlocks for active objects in ABS using the implementation of this net in CPN Tools [24].

The paper is organized as follows: Section 2 introduces the syntax and semantics of the ABS language, focusing on the language features for communication and synchronization. Section 3 briefly introduces colored Petri nets. Section 4 explains the translation from ABS semantics to colored Petri nets and the soundness proof for this translation. Section 5 presents a concrete ABS example and shows how the CPN Tools detects communication deadlock. Section 6 discusses related work and Section 7 concludes the paper.

2 The ABS Concurrency Model

The Abstract Behavioral Specification language (ABS) [18, 19] is an object-oriented language for modeling concurrent and distributed systems. ABS combines asynchronous communication from the Actor model [1, 2] with object orientation, and supports cooperative scheduling such that process release points

Syntactic categories.	*Definitions.*
s in Stmt	$P ::= \overline{CL}\ \{\overline{T}\ \overline{x};\ s\}$
e in Expr	$CL ::= \texttt{class}\ C\ (\overline{T}\ \overline{x})\ \{\overline{T}\ \overline{x};\ \overline{M}\}$
g in Guard	$Sg ::= T\ m\ (\overline{T}\ \overline{x})$
	$M ::= Sg\ \{\overline{T}\ \overline{x};\ s\}$
	$s ::= s; s \mid \texttt{skip} \mid x = rhs \mid \texttt{if}\ e\ \{s\}\ \texttt{else}\ \{s\}$
	$\mid \texttt{while}\ e\ \texttt{do}\ s \mid \texttt{suspend} \mid \texttt{await}\ g \mid \texttt{return}\ e$
	$rhs ::= e \mid cm \mid \texttt{new}\ C(\overline{e})$
	$cm ::= e!m(\overline{e}) \mid x.\ \texttt{get}$
	$g ::= x? \mid g \wedge g$

Fig. 1. Abstract syntax of ABS, where overline notation such as $\overline{e}$ and $\overline{x}$ denotes (possibly empty) lists over the corresponding syntactic categories.

are explicit in the program code. For the purposes of this paper, we focus on the communication and synchronization aspects of ABS. Also we ignore other aspects such as concurrent object groups, i.e., we consider one object per group, the functional sublanguage, and deployment aspects such as deployment components and resource annotations [19]. ABS is statically typed, based on interfaces as object types [18]. Ignoring the details of the type system, we let primitive types such as `Int` and `Bool` and class names constitute the types of a program, and ignore subtyping issues.

2.1 The Syntax

Fig. 1 presents the syntax of ABS [18], focusing on communication and synchronization. Programs P consist of class definitions CL and a main block representing the program's initial activity. Statements s include standard control-flow constructs such as sequential composition, assignment statement, conditionals, and while-loops. ABS supports *asynchronous* method calls $f = e!m(\overline{e})$ where the caller and callee proceed concurrently and f is a so-called *future*. A future is a "mailbox" where the return value from the method call may eventually be returned to by the callee. A future that contains return value is resolved. The result of the asynchronous call can then be obtained by $f.\ \texttt{get}$. Note that we may alternatively write asynchronous method call statement as $e!m(\overline{e})$, if the return value is not required. ABS also supports local synchronous calls which are more standard. For brevity, we elide discussion of synchronous method calls here (the CPN realization in Sec. 4 also covers synchronous, reentrant self-calls).

The (active) objects of ABS act like monitors, allowing at most one method activation, or process, to be executed at a time. The local execution in an object is based on *cooperative* scheduling by introducing a guard statement $\texttt{await}\ g$: If g evaluates to true, execution may proceed; if the guard g evaluates to false, execution is suspended and another process may execute. For a future f, the guard $f?$ evaluates true if f contains the return value from the associated method call and otherwise it evaluates to false. The **suspend**-statement always suspends the executing process. The typical usage of asynchronous calls follow the pattern $f = e!m(\overline{e}); \ldots; \texttt{await}\ \ f?; \ldots; x = f.\ \texttt{get}$.

2.2 The Operational Semantics

The operational semantics of ABS specifies transitions between *configurations.* A run-time configuration contains objects $o(\!|a, p, q|\!)$, messages $\langle o'.m(v)\rangle_f$, resolved futures $\langle v\rangle_f$, and unresolved futures $\langle \bot\rangle_f$. We use $\|$ to denote the (associative and commutative) parallel composition of such entities in a run-time configuration. Class definitions, which do not change during execution, are assumed to be implicitly available in the operational rules. The semantics maintains as invariant that object identities o and future identities f are unique. Objects $o(\!|a, p, q|\!)$ are instances of classes with an identifier o, an object state a which maps instance variables to values, an active process p, and an unordered queue q of suspended processes. A *process* p is a triple $\langle l \mid s\rangle_f$ with a local state l (mapping method-local variables to values), a statement s, and a future reference f. We omit the future reference in the rules if it is unnecessary. The special process *idle* is used to represent that there is no active process. A message $\langle o'.m(v)\rangle_f$ represents a method call *before* it starts to execute and the resolved future $\langle v\rangle_f$ the corresponding return value after method execution.

Fig. 2 gives the rules of the operational semantics, concentrating on the behavior of a single active object. A skip-statement has no effect (cf. rule Skip). In an idle object, the scheduler selects (and removes) a process p from the queue, and starts executing it (cf. rule Activate). Executing **`suspend`** moves the active process to the queue, resulting in an idle object (cf. rule Suspend). Assignments are either to instance variables or local variables (cf. rules Assign_1 and Assign_2, where σ is used to abbreviate the pair of local states l and object states a. We assume that these are disjoint, so the two cases are mutually exclusive.) We omit the standard rules for conditionals and while-loops. Object creation is captured by rule New-Object, where a' is the initial state of the new object (determined by an auxiliary function *atts*) and p' is the object's initial activity. An asynchronous method call creates a fresh future reference f and adds a message and unresolved future corresponding to the call to the configuration (cf. rule Async-Call). Binding a method name to the corresponding method body is done in rule Bind-Mtd. The binding operation, locating the code of the method body and instantiating the formal parameters, works in the standard way via late-binding, consulting the class hierarchy.

The return statement stores the return value in the corresponding future, resolving the future (cf. rule Return). The get-command allows the result value to be obtained from the corresponding future reference if the future's value has been produced, in which case the future has been *resolved* (cf. rule Get). Otherwise, the get-command blocks. An attempt to fetch a future value via a get statement does not introduce a scheduling point. Should the value never be produced, e.g., because the corresponding method activation does not return, the client object of the future, executing the get-command, will be blocked. A common pattern for obtaining a future value therefore makes use of await: executing **`await`** $x?; x.$ **`get`** checks whether or not the future reference for variable x has been produced. If not, the semantics of the await statement introduces a

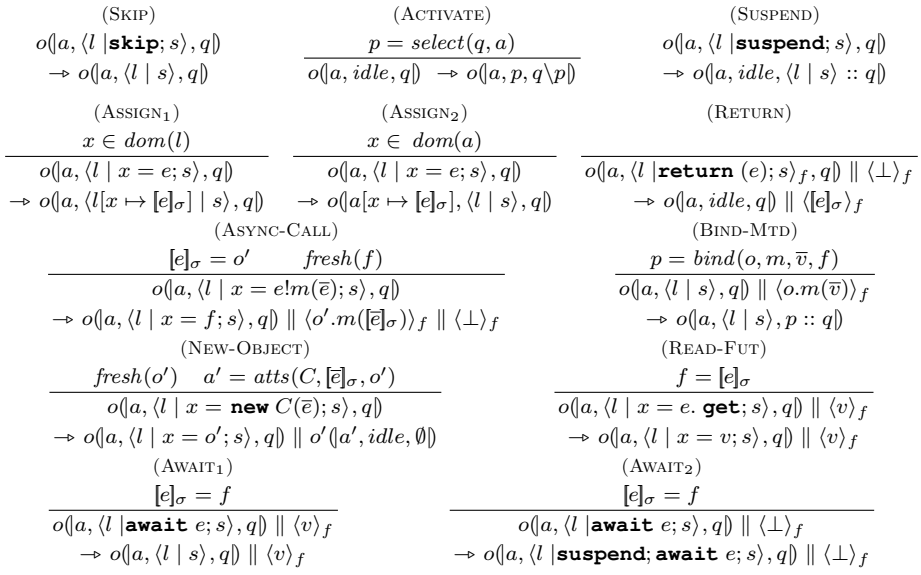

Fig. 2. Operational semantics

scheduling point. Once $x?$ evaluates to true, the future's value remains available so x. **get** will not block. (see again rule Read-Fut).

Executing an await with a guard expression which evaluates to the identifier of a resolved future, behaves like a skip (cf. rule Await$_1$). An await on a list of futures are equivalent to a list of awaits for individual futures. If the future corresponding to the guard expression has not been resolved, a **suspend**-statement is introduced to enable scheduling another process (cf. rule Await$_2$).

3 Colored Petri Nets

Places and transitions in Petri nets capture true concurrency in terms of causality and synchronization [22, 25]. Colored Petri nets (CPNs) extend the basic Petri net formalism to additionally model, e.g., data [16, 17]. A CPN has color sets (= types). The set of types determines the data values and the operations that can be used in the net expressions. A type can be arbitrarily complex, defined by many sorted algebra in the same way as abstract data types. Each place in a CPN has an associated color set, restricting the kind of data a place can contain. Tokens in a typed place represent individual values of that type. CPNs in their basic form (ignoring hierarchical definitions) are defined as follows:

Definition 1 (Colored Petri net). *A* colored Petri net *(CPN) is a tuple* $(P, T, A, \Sigma, V, C, G, E, I)$ *where*

- places P *and* transitions T *are disjoint finite sets;*

– arcs A *form a bipartite, directed graph over* P *and* T*, i.e.,* $A \subseteq P \times T \dot{\cup} T \times P$*;*
– types Σ *form a finite set (each type seen as a non-empty "color set");*
– typed variables V *form a finite set, i.e.,* $type(v) \in \Sigma$ *for all* $v \in V$*;*
– *a* coloring $C : P \to \Sigma$ *associates a type to each place.*
– labeling functions $G : T \to Expr_V$ *(guards) and* $E : A \to Expr_V$ *associate expressions to transitions and arcs; and the*
– initialization function $I : P \to Expr_\emptyset$ *associates expressions to places.*

in which expressions are appropriately typed; i.e., $type(G(t)) = Bool$*,* $type(E(a)) = C(p) \to \mathbb{N}$*, where* p *is the place connected to* a*, and* $type(I(p)) = C(p) \to \mathbb{N}$ *for all places.*

Transitions and their guards express synchronization conditions which, together with the labels on the arcs, express the transition semantics of Petri nets. Since tokens are individual typed values and expressions contain variables, the enabledness of transitions depends on the choice of values for the free variables.

Bindings (or variable assignments) b are mappings from variables to values; we assume bindings to respect the types of the variables. The *variables of a transition* t, written $\mathrm{Var}(t) \subseteq V$, consist of the free variables in the guard of t and in the arc expressions of the arcs connected to t. The binding of a transition covers (at least) all variables from $Var(t)$. Let $[\![E]\!]_b$ denote the value of expression E under variable binding b. Given a CPN, a *marking* M is a function $P \to (\Sigma \to \mathbb{N})$ (the *initial marking* $M_0(p)$ is defined by $I(p)$) and a *step* is a selection of the net's transitions together with appropriate bindings for the variables of each transition such that the selected transitions are enabled, defined as follows:

Definition 2 (Enabledness). *A transition* t *is* enabled *in a marking* M *for binding* b*, if,*

1. $[\![G(t)]\!]_b = true$*, and*
2. $M(p) \geq_m [\![E(p,t)]\!]_b$*, for all places* $p \in P$*,*

where $\geq_m$ *is the usual ordering between multisets.*
A step Y *is* enabled *in a marking* M*, if for all places* p*,* (t, b) *from* Y*,* t *is enabled for* b *in* M*,* $M(p) \geq_m [\![E(p,t)]\!]_Y$*. The semantics* $[\![E(p,t)]\!]_Y$ *represents the multi-set* $\sum_{(t,b)\in Y} [\![E(p,t)]\!]_b$*.*

When t is enabled for b, in M, it may *occur* or "fire", leading to the marking M' where $M'(p) = (M(p) - [\![E(p,t)]\!]_b) + [\![E(t,p)]\!]_b$, for all places p. Similarly for enabled steps Y, $M_1 \overset{Y}{\twoheadrightarrow} M_2$ denotes that a marking M_1 evolves into M_2 by "firing" step Y. A (finite) *occurrence sequence* is a sequence of markings and steps of the form

$$M_1 \overset{Y_1}{\twoheadrightarrow} M_2 \overset{Y_2}{\twoheadrightarrow} M_3 \ldots M_n \overset{Y_n}{\twoheadrightarrow} M_{n+1} \ . \tag{1}$$

Note that "true concurrency" semantics, typical for Petri nets, allows the simultaneous, firing of transitions in a step. Whereas steps are required to be non-empty, a step which only fires one transition t and binding b, is denoted $\overset{t,b}{\to}$. A reduction semantics restricted to such single transition steps is equivalent to the unrestricted semantics, but corresponds to "interleaving concurrency".

4 Translating ABS Semantics to Colored Petri Nets

In this section, we define the translation from ABS to CPNs. After a short introduction covering the core ideas of the translation, in Section 4.2 we highlight crucial parts of how the ABS semantics are represented on the Petri net level, focusing on parts of the communication mechanism, in particular dealing with asynchronous method calls and the resolution of futures via **`get`**. In Section 4.3, we define an abstraction function relating program configurations and the corresponding Petri net markings. Afterwards, Section 4.4 establishes the soundness of the Petri net semantics, defining a simulation relation between the steps of the operational semantics and the transitions of the resulting Petri net.

4.1 Overview over the Petri Net Semantics for ABS

The starting point of the translation are abstract ABS programs, i.e. programs where *data* values have been abstracted already. Still, there are two remaining sources of *infinity* in the state space: creation of (active) objects and creation of processes and accompanying future references via asynchronous method calls. Note in passing that in absence of synchronous, reentrant method calls, unboundedly growing stacks do not contribute to the potential unboundedness of the state space. In the translation, one can conceptually distinguish between *language-specific* aspects and *program-specific* aspects: the ABS language and its semantics is represented by one CPN, common for all programs. This CPN therefore can be seen as a translation of the ABS-language as such. Roughly, each semantic rule from the operational semantics of Fig. 2 is represented by transitions and places, with appropriate types and guards. Fig. 3 shows a birds eye view of the overall Petri net as represented in the CPN Tools.

In contrast, one particular program, respectively, one particular run-time configuration of a program, is represented by a *marking* of the Petri net. The expressive power of *colored* Petri nets is crucial to achieve such a conceptually clear and structural translation: since tokens are distinguishable, the transitions and places operated on type values allow to represent the components of a configuration in a clean manner. For instance, object, process, and future identities are all naturally represented in the tool by integers.

4.2 CPN-ABS Communication Mechanism

Fig. 3 shows the implementation of this translation with the CPN Tools. From now on, we will refer to it as CPN-ABS. In CPN-ABS, communication takes place between objects represented as tokens which carry information about their identity, their class, and their process pool, therefore triples of the form $(id,class,q)$. CPN-ABS supports not only object communication, but also the construction of the information each object carries. This allows dynamic creation of objects. CPN-ABS can be structurally divided into two parts: the first part, where all this information can be dynamically created through transition firing, and the second part which can simulate the possible communications of the objects. As

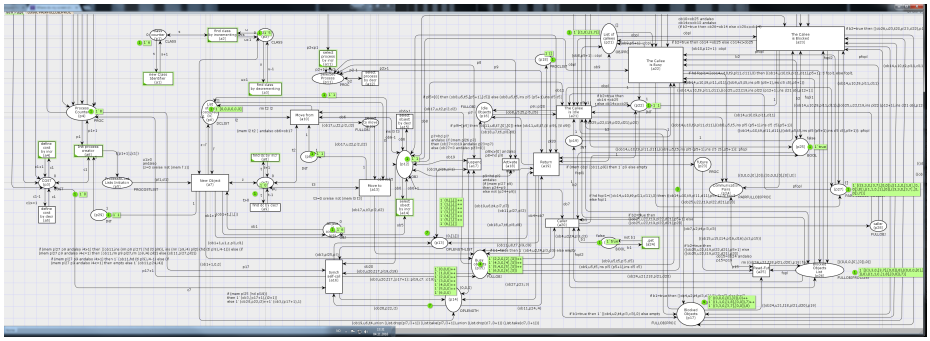

Fig. 3. ABS semantics implemented with the CPN Tools

shown in Fig. 3, CPN-ABS contains a lot of details in order to faithfully simulate ABS. In the following, we concentrate on an extract of the implementation (cf. Fig. 4), which focuses on the asynchronous communication mechanism. The implementation covers all ABS rules from Section 2, as well as synchronous reentrant self calls. For simplicity, in Fig. 4, we omit details like places which have an indirect relation with the semantics, and furthermore arcs and inscriptions where obvious.

As we can see in Fig. 4, there are three disjoint places where the object tokens can be located: *"Active Objects"*, *"Idle Objects"*, and *"Blocked Objects"*. When a method of an (active) object returns (here by firing the transition *"Return"*), it resolves a future and it moves the object to the *"Idle Objects"* place, as one can observe from the RETURN rule of the semantics. The inverse can be achieved through the ACTIVATE rule, where a process from the pool is activated. This is simulated by the *"Activate"* transition with the corresponding token moving.

Transition *"Caller"* selects the calling object from the *"Active Objects"* place (here we omit how the object selection is being done). We have two cases of communication through asynchronous method calls: immediately followed by a **`get`** statement or not. Both are simulated in the yellow region of the picture: It contains one place, *"Is_synchronous"*, which has a token of type *Bool*. Its value corresponds to the presence of a **`get`** statement in the obvious way. By firing the transition *"Get"*, we alternate the value of the token. So, from this yellow region, transition *"Caller"* takes the information on whether the asynchronous call is followed by a **`get`** statement or not. In the latter case, the value of b is false and transition *"Caller"* maintains the object in the *"Active Objects"* place (which has the corresponding meaning for the status of the object – see rule ASYNC-CALL of the semantics), otherwise it sends the caller object to the *"Blocked Objects"* place until the waiting future can be retrieved from the *"Future"* place (see rule READ-FUT in the ABS semantics).

As the places related to the status of an object are disjoint, the callee object can reside only in one among the three corresponding places. Therefore, one among the transitions *"Active callee"*, *"Blocked callee"*, and *"Idle callee"* can fire each time for the selected object (here, again, we omit details about how

the object selection is done). In CPN-ABS, the process pool is implemented as a FIFO queue. So, the transitions that refer to the callee update its process queue by adding at the end a new process related to this particular method call. They also create a communication pair token at the *"Communication pairs"* place by matching the token of the *"Caller"* place (created by the *"Caller"* transition) with the callee object and the process created for this method execution.

4.3 The Abstraction Function

In this section, we define a translation from ABS configurations to Petri net markings. The translation is given in the form of an abstraction function α. In it's core, it's a structural translation of ABS-configurations, ignoring the *data* parts of the program, i.e., the value of variables in the instance states and local states. Hence the translation yields an *abstraction* at the same time, and the resulting Petri nets marking over-approximate the original behavior, due to this form of data abstraction. Let *Obj* be the set of objects in an ABS program, *Class* the set of its classes and *Proc* the set of the processes. We define the following injections from those sets to the set of positive integers: $h : Obj \to \mathbb{Z}^+$, $d : Class \to \mathbb{Z}^+$, and $g : Proc \to \mathbb{Z}^+$. Let $\mathcal{C}$ be the set of the configurations of an ABS program and *Msg* the set of the invocation messages. We define the projection functions from the ABS configurations as follows:

- $cl : \mathcal{C} \to Class$ which projects the object class in an ABS configuration,
- $ob : \mathcal{C} \to Obj$ which projects the objects in ABS configurations,
- $pr : \mathcal{C} \to \mathcal{P}(Proc)$ which projects the process pools of the objects of ABS configurations,
- $msg : \mathcal{C} \to Msg$ which projects the messages *Msg* of ABS configurations and
- $fut : \mathcal{C} \to F$ which projects the set of resolved futures that are related to **`get`** statements for each configuration.

Then, let $m : Msg \to Proc$ be the injection which maps each invocation message to the process that will be created for the execution of the called method. Let furthermore $fr : F \to Proc$ be the injection from the set of resolved futures F related to **`get`** statements to the set of processes *Proc*, since they are related to the change of the blocked status of the objects which wait to read those futures. Finally, let $pq : \mathcal{P}(Proc) \to \mathcal{P}(\mathbb{Z}^+)$ be the mapping from the process pools to sets of (unique) positive integers such that for every process pool S, $pq(S) = \{g(s) \in \mathbb{Z}^+ \mid s \in S\}$. In CPN-ABS, we model objects as tokens which carry information about their identity, their class and their process pool. As a consequence, each object is represented as a triple (id,$class$,q), where id is the object identifier of type *Int*, $class$ is the corresponding class of the object i.e. the class identifier of type *Int* and q is the process pool of the object of type list of integers.

Now, we can define the abstraction function α. In the following, P is the set of the places and $M(p)$ the marking of a place p in CPN-ABS. Then, for all

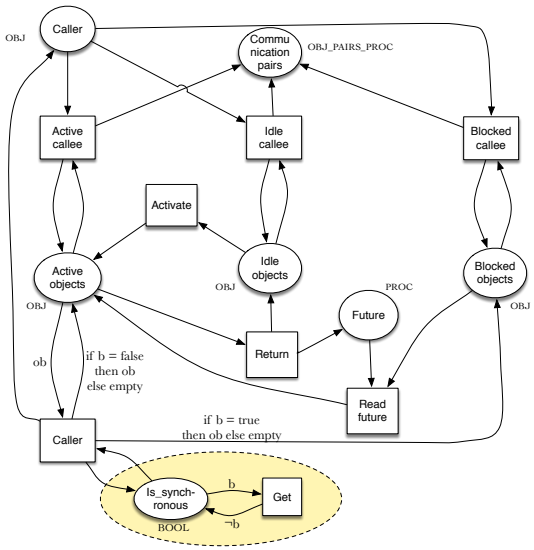

Fig. 4. Extract of the communication mechanism of CPN-ABS

configurations $c \in \mathcal{C}$:

$$\begin{aligned}\alpha(c) = \bigcap \{M \mid \exists p, p', p'' \in P \text{ s.t. } & p \neq p' \neq p'' \\ & \wedge((h \circ ob)(c), (d \circ cl)(c), (pq \circ pr)(c)) \in M(p) \\ & \wedge\ (m \circ msg)(c) \in M(p') \\ & \wedge\ (g \circ fr \circ fut)(c) \in M(p'')\}\ ,\end{aligned} \tag{2}$$

where, $\bigcap$ denotes intersection over sets of multisets. Observe that, for every ABS configuration, the above intersection is nonempty, i.e. there is a marking such that all the objects of the configuration are represented as tokens in specific places of the model.

4.4 Soundness Proof of the Translation

In this section we sketch the soundness proof of the translation, establishing a *simulation* relation between the small step operational semantics of ABS and the transitions of CPN-ABS. In particular, we need to prove that, for any ABS configuration c, if $c \rightarrowtriangle_r c'$ for some semantic rule r, then there exists a marking M' and a *sequence* of CPN-ABS transitions u, such that $\alpha(c) \xrightarrow{u} M'$ and $\alpha(c') \subseteq_m M'$ (where, with $\subseteq_m$ we denote the subset relation between sets of multisets as an extension of $\leq_m$). To establish the above relation, we need to prove that u has a corresponding CPN-ABS occurrence sequence, i.e. that all the transitions of u can fire in the same order as they appear in u.

Consequently, we try to construct each transitions sequence u in such a way that there exists the corresponding occurrence sequence. We use a finite alphabet $\mathcal{B}$ which consists of the names of the transitions that appear in CPN-ABS and

construct words over this alphabet that correspond to occurrence sequences, with ϵ to be the empty word. We should mention here, that, for all $b \in \mathcal{B}$, $b^0 = \epsilon$. We call these words *occurrence words*. The set that contains those occurrence words can be given from the image of a translation function $Tr : Sem \to \mathcal{B}^*$, where *Sem* is the set of the ABS semantic rules of Section 2. In the following, we provide some definitions and lemmas in order to achieve modularity for the construction of occurrence words. We will denote as $En(M)$ *the set of enabled transitions for a marking* M and $\mathcal{M}_{reach}$ the set of reachable markings of the Petri net.

Definition 3 (Independent transition). *A transition $t \in T$ is called* independent *if, for any marking $M \in \mathcal{M}_{reach}$, $t \in En(M)$ and $M \xrightarrow{t} M'$ implies $En(M) \subseteq En(M')$.*

Definition 4 (Post-transition). *The* post-transitions *of a transition $t \in T$ are given by the function $PostTrans : T \times \mathcal{M}_{reach} \to \mathcal{P}(T)$ where $PostTrans(t, M) = \{t' \in En(M') \mid M \xrightarrow{t} M'\}$.*

Lemma 1 (Composition). *The* composition *of an occurrence sequence $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_{n+1}$ with another occurrence sequence $M'_1 \xrightarrow{t'_1} M'_2 \xrightarrow{t'_2} \ldots \xrightarrow{t'_m} M'_{m+1}$ is the occurrence sequence $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_{n+1} \xrightarrow{t'_1} M''_2 \xrightarrow{t'_2} \ldots \xrightarrow{t'_m} M''_{m+1}$, whenever $M'_1 \subseteq_m M_{n+1}$ and $[\![G(t'_1)]\!]_{b_{n+1}} = true$ and furthermore $\bigwedge_{2 \leq i \leq m} [\![G(t'_i)]\!]_{b_i} = true$ and $M'_j \subseteq_m M''_j$, for all $1 \leq j \leq m+1$.*

Proof. For the prefix of the sequence which is identical to the first composed sequence, the result is trivial . Then, since $M'_1 \subseteq_m M_{n+1}$, after t'_1, obviously, if $[\![G(t'_2)]\!]_{b''_2} = true$, then $M'_2 \subseteq_m M''_2$, and so on. □

In the sequel, we accordingly use the term *composition of occurrence words.*

Lemma 2. *For all ABS semantic rules r, $Tr(r)$ is an occurrence word.*

Proof. The idea is to assign to *Tr* a concrete value for each possible argument (i.e. for each rule of the operational semantics, and then, for each value, to prove that it is an occurrence word). As in Section 4.2 we presented just an extract of the real implementation, we will present one representative case, namely rule Read-Fut rule based on the Petri net extract of Fig. 4.

In this case, $Tr(\text{Read-Fut}) =$ *"Get"*$^{i-1}$ *"Caller" "Read Future"*, where $i = 1$ if the marking of the place *"Is_synchronous"* is *true* before firing *"Get"*, 0 otherwise. We need to prove that it is an occurrence word. Indeed, let c be the configuration before the application of the rule Read-Fut and c' the one after it. Let, ob_1 be the object abstracted from c. Then $ob_1 \in M($*"Active Objects"*$)$. *"Get"* is an independent transition. If $i = 0$, then $M \xrightarrow{\text{Get}} M^{(1)}$ is an occurrence sequence. Otherwise, $M^{(1)} = M$. Obviously, $M^{(1)}($*"Is_synchronous"*$) = \{true\}$. Transition *"Caller"* $\in En(M^{(1)})$, since *"Caller"* $\in En(M)$ and also *"Caller"* is a post-transition of *"Get"*, so we can take $M^{(1)} \xrightarrow{\text{Caller}} M^{(2)}$ where

```
1  class Service(Int limit) {
2    Producer prod = new Producer(); Proxy proxy = new Proxy(limit,this,prod);
3    Proxy lastProxy = proxy;
4
5    Void run() { this!produce(); }
6    Void subscribe(Client cl){Fut<ProxyI> f; f = lastProxy!add(cl); lastProxy = f.get;}
7    Void produce(){proxy!start_publish(); }
8  }
9  class Proxy(Int limit, Service server, Producer prod) {
10   List<Client> myClients = Nil; Proxy nextProxy;
11
12   Proxy add(Client cl){ Proxy lastProxy = this; Fut<ProxyI> f';
13    if length(myClients) < limit {myClients = append(myClients, cl);}
14    else {if nextProxy == null {nextProxy = new Proxy(limit,server,prod);}
15       f' = nextProxy!add(cl); lastProxy = f'.get;} return lastProxy; }
16
17   Void start_publish(){ Fut<ProxyI> f''; f'' = prod!detectNews(); await f''?;
18                                          News ns = f''.get; this!publish(ns); }
19
20   Void publish(News ns){ myClients!signal(ns);
21    if nextProxy == null {server!produce();} else {nextProxy!publish(ns);} }
22 }
```

Fig. 5. Implementation of the publisher-subscriber example.

$ob_1 \in M^{(2)}$(*"Blocked Objects"*). So, *"Get" "Caller"* is an occurrence word. From the hypothesis of READ-FUT we know that there exists a marking $M^{(3)}$ s.t. $f \in M^{(3)}$(*"Future"*) and from Lemma 1, we obtain that *"Get"* $^{i-1}$ *"Caller" "Read Future"* is an occurrence word. □

Theorem 1 (Simulation). *CPN-ABS is a (weak) simulation of ABS.*

Proof. We need to prove that, for any ABS configuration c, if $c \rightarrowtriangle_r c'$ for some semantic rule $r \in Sem$, then there exists a marking M' and an occurrence word given by $Tr(r)$, such that $\alpha(c) \overset{Tr(r)}{\rightarrow} M'$ and $\alpha(c') \subseteq_m M'$. This follows straightforwardly from the definition of the abstraction function α, the image of Tr, and from Lemma 2. □

5 Deadlock Detection

The translation CPN-ABS and the underlying Petri net tool can be used for the detection of possible communication deadlocks of ABS programs. CPN-ABS contains three disjoint places, where, depending on the status of objects (i.e. active, idle or blocked), objects can be located. The place *"Blocked Objects"* which hosts the blocked objects has a color set of pair (*ob*,*p*), where *ob* is object invoking an asynchronous call with a get-statement, i.e. an asynchronous blocking call, and p is the process that has been added to the process queue of the callee

for the execution of the called method. Recall that *ob* is of color (*id*,*class*,*q*), where *id* is object identity, *class* is the class that the object belongs to, and *q* is the process queue of the object. Given this particular structure of CPN-ABS, *there is a deadlock cycle [21] if and only if there exists a marking of the place* "Blocked Objects"*, in which there exists n tokens* (ob_1, p_1) *to* (ob_n, p_n) *that form a cycle, i.e. for* $1 \leq i < n$*,* $p_i \in q_{i+1}$ *and* $p_n \in q_1$ (where q_i is the process queue of the i^{th} object). This deadlock situation can be detected by the state space report of the model checker of the CPN Tool used to implement CPN-ABS.

5.1 Example

We now use the publisher-subscriber example of Fig. 5 to illustrate how CPN-ABS detects communication deadlocks. `Service` objects publish news updates to subscribing clients through a chain of `Proxy` objects. Each proxy object handles a bounded number of clients. `Service` objects handle a subscribe request efficiently by delegating its time-consuming parts to `Proxy` objects, and the proxies publish news to clients using asynchronous calls (without futures) to make the cooperation efficient. As asynchronous method calls without get-statements do not cause deadlocks, we omit them from our analysis and only consider asynchronous blocking calls of the form $f = e!m(\overline{e}); \ldots; x = f.\textbf{get}$, where there are no suspension points in between. There are two asynchronous blocking calls in lines 6 and 15 in the example, namely $f = \mathit{lastProxy}!\mathit{add}(\mathit{cl}); \mathit{lastProxy} = f.\textbf{get}$ and $f' = \mathit{nextProxy}!\mathit{add}(\mathit{cl}); \mathit{lastProxy} = f'.\textbf{get}$. The former one expresses that a `Service` object invokes method `add` on a `Proxy` object through method `subscribe`. Similarly, the later one expresses that a `Proxy` object invokes method `add` on the next `Proxy` object through method `add`. By applying the model checker on an Intel i7 3.4 GHz, in less than 1 second we get the full state space report in which tokens of color $((o_1, \mathit{Service}, q), p)$ and $((o_3, \mathit{Proxy}, q'), p')$ can be found in the place *"Blocked Objects"*, and for all p, p', q, q' we have $p \notin q'$ and $p' \notin q$. This shows that the implementation of the publisher-subscriber protocol is deadlock free.

Now, we slightly modify the protocol, where **`get`**-statements are added to the method calls in lines 7 and 21 and the **`await`** statement in line 17 is removed. In this case, CPN-ABS detects a communication deadlock cycle shown in Fig. 6, where $p \in q'$ and $p' \in q$ and both objects are trapped in the place *"Blocked Objects"* and cannot exit from there; in Fig. 6, the third and the fifth argument in the color tuples are outside of the scope of this work, so we ignore them, while, the existence of the two zero value tokens is for initialization reasons and they do not affect the deadlock analysis. Based on the information we obtained from this reachable marking, we can trace back to the program code and determine the deadlock represented by the call chain.

Remark that the translation supports scalability: the size of the net is independent from the program and represents the ABS semantics as such. I.e., by increasing the number of `Proxy` objects or clients, only the number of tokens is affected and the analysis is highly automated.

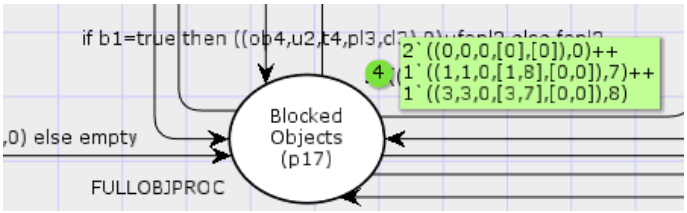

Fig. 6. Deadlock detection by CPN-ABS.

6 Related Work

Deadlock detection is traditionally concerned with the usage of locks for thread-based concurrency. This line of work is surveyed in [23], which develops a type and effect system to capture lock manipulation for such a language. However, in active objects communication deadlocks are caused by call-cycles with synchronization, and the cooperative scheduling of ABS makes the analysis more complex. The problem has been studied using different approaches, including behavioral types [13], cost analysis [12], protocol specifications [21], and Petri nets [10] (discussed below). As the problem is undecidable and the approaches differ substantially, it is difficult to say exactly how they relate to each other in terms of strength of the proposed analyses. Petri nets and its extensions are popular formalisms to model and analyze systems with concurrency, communication and synchronization [22, 25]. Petri nets have in particular been applied to protocol and work flow analysis, but have also been used to study process algebra (e.g., [5, 7]), more recently including asynchronous communication [4]. Approaches which encode programming language features into Petri nets have been developed for Ada [15] and more recently for, e.g., Java [20], and for choreography languages like Orc [6]. In general, these approaches translate programs into nets such that the size of the program determines the size of the net and dynamic invocations or object creation cause difficulties. Previous work on deadlock analysis for active objects using Petri nets [10] follows a similar approach such that places represent locks on objects, futures, and processes. Transitions are introduced for each possible caller and callee to a method. To obtain a finite net, the approach abstracts from the actual number of futures such that the wrong future may be accessed in the Petri net. But if the net is deadlock free, so is the original active object program. In contrast to these approaches encoding a specific program as a net, our approach directly encodes the language semantics as a CPN and uses markings to define the concrete program; the colors of CPN are used to distinguish different method invocations and to create new objects and the size of the net itself is independent of the specific program.

7 Conclusion

This paper proposes an encoding of the formal semantics of ABS as a net, such that a program is given as a marking for this net. Exploiting the colored tokens,

our net can support dynamic program behavior. We provide a soundness proof for our encoding and show how a model checker for colored Petri nets can be used to analyze communication deadlock for active objects in ABS. Whereas this paper has focused on communication and synchronization for ABS programs, ABS supports the specification of real-time behavior, deployment architectures, and resource-aware systems [19]. Our next step is to extend the model to support these features, and explore the usage of colored Petri nets for resource analysis and to compare resource-management strategies for distributed ABS programs.

References

1. G. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1986.
2. G. Agha and C. Hewitt. Concurrent programming using actors. In *Object-Oriented Concurrent Programming*. The MIT Press, 1987.
3. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
4. P. Baldan, F. Bonchi, F. Gadducci, and G. V. Monreale. Modular encoding of synchronous and asynchronous interactions using open Petri nets. *Science of Computer Programming*, 109, 2015.
5. E. Best, R. R. Devillers, and M. Koutny. *Petri net algebra*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2001.
6. R. Bruni, H. C. Melgratti, and E. Tuosto. Translating Orc features into Petri nets and the join calculus. In *WS-FM'06*, volume 4184 of *Lecture Notes in Computer Science*. Springer Verlag, 2006.
7. N. Busi and R. Gorrieri. A Petri net semantics for pi-calculus. In *CONCUR '95*, volume 962 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.
8. D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer Verlag, 2005.
9. K. M. Chandy, J. Misra, and L. M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2), 1983.
10. F. S. de Boer, M. Bravetti, I. Grabe, M. Lee, M. Steffen, and G. Zavattaro. A Petri net based analysis of deadlock for active objects and futures. In *FACS 2012*, Lecture Notes in Computer Science. Springer Verlag, 2013.
11. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*. Springer Verlag, Mar. 2007.
12. A. Flores-Montoya, E. Albert, and S. Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *FMOODS/FORTE 2013*, volume 7892 of *Lecture Notes in Computer Science*. Springer Verlag, 2013.
13. E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in core ABS. *Software and System Modeling*, 15(4), 2016.
14. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3), 2009.
15. J. Ichbiah, J. G. P. Barnes, J. C. Heliard, B. Krieg-Brückner, O. Roubine, and B. A. Wichmann. Modules and visibility in the Ada programming language. In *On the Construction of Programs*. Cambrige University Press, 1980.
16. K. Jensen. Coloured Petri Nets. In *Petri Nets: Central Models and their Properties, (Advances in Petri Nets 1986) Part I*, volume 254 of *Lecture Notes in Computer Science*. Springer Verlag, 1987.

17. K. Jensen and L. M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer Verlag, 2009.
18. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO 2010*, volume 6957 of *Lecture Notes in Computer Science*. Springer Verlag, 2011.
19. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 84(1), 2015.
20. B. Long, P. A. Strooper, and L. Wildman. A method for verifying concurrent Java components based on an analysis of concurrency failures. *Concurrency and Computation: Practice and Experience*, 19(3), 2007.
21. O. Owe and I. C. Yu. Deadlock detection of active objects with synchronous and asynchronous method calls. In *Proceedings of NIK 2014*, 2014.
22. C. Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Bonn, 1962.
23. K. I. Pun. *Behavioural static analysis for deadlock detection*. PhD thesis, Department of informatics, University of Oslo, Norway, 2014.
24. A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN tools for editing, simulating, and analysing coloured Petri nets. In *ICATPN 2003*, volume 2679 of *Lecture Notes in Computer Science*. Springer Verlag, 2003.
25. W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs in Computer Science*. Springer Verlag, 1985.