Science of Computer Programming 181 (2019) 1–26

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Translating active objects into colored Petri nets for communication analysis ☆

Anastasia Gkolfi [a,*], Crystal Chang Din [a], Einar Broch Johnsen [a], Lars Michael Kristensen [b], Martin Steffen [a], Ingrid Chieh Yu [a]

[a] *Department of Informatics, University of Oslo, Norway*
[b] *Department of Computing, Mathematics and Physics, Høgskulen på Vestlandet, Norway*

ARTICLE INFO

Article history:
Received 15 February 2018
Received in revised form 15 January 2019
Accepted 13 April 2019
Available online 6 May 2019

Keywords:
Colored Petri nets
Model checking
Semantics
Static analysis
Concurrency

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 may introduce communication cycles which lead to a form of communication deadlock and livelock. This paper addresses such communication deadlocks and livelocks 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 program and its state correspond 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 deadlocks and livelocks.

1. Introduction

The Actor model [1,2] of concurrency is attracting increasing attention for its *decoupling* of control flow and communication. This decoupling enables both scalability (as argued with the Erlang programming language [3] and Scala's actor model [4]) and compositional reasoning [5–7]. 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., [8–10]) 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 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*

☆ The work was partially supported by the Research Council of Norway under the CUMULUS project.

* Corresponding author.
E-mail addresses: natasa@ifi.uio.no (A. Gkolfi), crystald@ifi.uio.no (C.C. Din), einarj@ifi.uio.no (E.B. Johnsen), Lars.Michael.Kristensen@hvl.no (L.M. Kristensen), msteffen@ifi.uio.no (M. Steffen), ingridcy@ifi.uio.no (I.C. Yu).

https://doi.org/10.1016/j.scico.2019.04.002
0167-6423/

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* [11].

In this paper we work with the active object language ABS [9,12]. 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. However, while no progress is made at the suspension points, the scheduler is continuously activating and releasing processes. We call this situation *communication livelock*, in the sense that the object is not blocked, but the generated processes can not progress due to that each of them is busy-waiting for a process on another object.

This paper addresses the problem of communication deadlock and livelock for the active object language ABS. Our approach to tackle these problems is based on a translation of the formal semantics of ABS into colored Petri nets (CPN) [13]. Petri nets provide a basic model of concurrency, causality, and synchronization [14,15], which has previously been used to analyze communication patterns and deadlock (e.g., [16,17]). CPNs extend the basic Petri net model with support for modeling data. In contrast to previous work, we do not produce a different Petri net for each ABS 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 and its state. 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. This also allows us to capture dynamic object creation by firing transitions in the net. The main contributions of this paper are:

- an encoding of the formal semantics of ABS in CPNs (Section 4);
- a translation of concrete ABS programs into markings of this net (Section 5);
- a soundness proof for the translation from ABS to CPN (Section 6); and
- a case study demonstrating how to analyze communication deadlocks and livelocks for active objects in ABS using the implementation of this net in CPN Tools [18] (Section 7).

The CPN model itself is available under https://github.com/natasa-gkolfi/cpnabs. This paper extends a paper from FSEN 2017 [19]; compared to that paper we here present the translation of ABS into CPN and the associated correctness proof in full detail, and add support for livelock analysis. The communication analysis has also been improved to be directly supported in the CPN Tools.

The paper is organized as follows: Section 2 introduces the ABS language, focusing on language features for communication and synchronization, and Section 3 introduces colored Petri nets. Section 4 discusses the translation from ABS semantics to colored Petri nets and Section 5 the translation from ABS configurations to Petri net markings, before Section 6 considers the soundness of the translation. Section 7 presents ABS examples and show how the CPN Tool detects communication deadlock and livelock, respectively. Finally, Section 8 concludes the paper with a discussion of related and future work.

2. The ABS concurrency model

The Abstract Behavioral Specification language (ABS) [9,12] is an object-oriented language for modeling concurrent and distributed systems. ABS is an active object language [10] which combines asynchronous communication from the Actor model [1,2] with object orientation. Objects protect their own local state in a monitor-like fashion and support cooperative scheduling, where each process or method activation can relinquish access via explicit scheduling commands such as **`suspend`** or **`await`** (see below). In 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 [12]. ABS is statically typed, based on interfaces as object types [9]. 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. It is possible to add these details to the translation in a similar way, but in the current work we are interested in the communication topologies of ABS rather than in the full functional details as our focus is on communication analysis and the detection of deadlocks and livelocks.

2.1. The syntax

Fig. 1 presents the syntax of ABS, focusing on communication and synchronization. For a more detailed introduction to the syntax and semantics of the language we refer to [9]. Programs P consist of class definitions CL and a main block representing the program's initial activity. Each class is defined by its class name C, its typed constructor arguments, instance variable declarations, and a set of method definitions. A method definition M consists of a signature Sg, where x represent variables and T represents types. The syntactic category of types is left unspecified as the paper is not concerned with the language's static type system, but includes standard types like Booleans and integers, and in particular class names as types for objects. 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

Syntactic categories.		*Definitions.*
s in Stmt		$P ::= \overline{CL}\ \{\overline{T}\ \overline{x};\ s\}$
e in Expr		$CL ::= \mathbf{class}\ C\,(\overline{T}\ \overline{x})\,\{\overline{T}\ \overline{x};\ \overline{M}\}$
g in Guard		$Sg ::= T\ m\ (\overline{T}\ \overline{x})$
		$M ::= Sg\ \{\overline{T}\ \overline{x};\ s\}$
$x, y \ldots$	variables	$s ::= s; s \mid \mathbf{skip} \mid x = rhs \mid \mathbf{if}\ e\ \{s\}\ \mathbf{else}\ \{s\}$
$m, m' \ldots$	method names	$\quad \mid \mathbf{while}\ e\ \mathbf{do}\ s \mid \mathbf{suspend} \mid \mathbf{await}\ g \mid \mathbf{return}\ e$
		$rhs ::= e \mid cm \mid \mathbf{new}\ C(\overline{e})$
		$cm ::= e!m(\overline{e}) \mid e.m(\overline{e}) \mid x.\ \mathbf{get}$
		$g ::= x? \mid g \wedge g$

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.

be returned to by the callee. A future that contains a return value is resolved. The result of the asynchronous call can then be obtained by $f.$ **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 $e.m(\overline{e})$ which synchronize in the usual reentrant way, passing control directly to the called method and then back to the callee upon completion.

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 **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 to 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 $f = e!m(\overline{e}); \ldots;$ **await** $f?; \ldots; x = f.$ **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 that are candidates to be activated by the scheduler if the currently active process will suspend or return. A emphprocess 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). Assign_1 and Assign_2 are the assignment rules. Assignments are either to instance variables or local variables, 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. The semantic value of an expression e relative to state σ is denoted by $[\![e]\!]_\sigma$, where the values of variables occurring in e are given by σ.

Object creation is captured by the New-Object rule, where a' is the initial state of the new object (determined by an auxiliary function *atts* having the class C as a parameter) 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** statement 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 Read-Fut). Otherwise, the **get** statement 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** statement, 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 scheduling point. Once the guard $x?$ 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). 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).

$$\text{(SKIP)}\quad o(\!|a,\langle l\mid \mathbf{skip};s\rangle,q|\!) \rightharpoonup o(\!|a,\langle l\mid s\rangle,q|\!)$$

$$\text{(ACTIVATE)}\quad \frac{p = select(q,a)}{o(\!|a,idle,q|\!) \rightharpoonup o(\!|a,p,q\backslash p|\!)}$$

$$\text{(SUSPEND)}\quad o(\!|a,\langle l\mid \mathbf{suspend};s\rangle,q|\!) \rightharpoonup o(\!|a,idle,\langle l\mid s\rangle :: q|\!)$$

$$\text{(ASSIGN}_1\text{)}\quad \frac{x\in dom(l)}{o(\!|a,\langle l\mid x=e;s\rangle,q|\!) \rightharpoonup o(\!|a,\langle l[x\mapsto [\![e]\!]_\sigma]\mid s\rangle,q|\!)}$$

$$\text{(ASSIGN}_2\text{)}\quad \frac{x\in dom(a)}{o(\!|a,\langle l\mid x=e;s\rangle,q|\!) \rightharpoonup o(\!|a[x\mapsto [\![e]\!]_\sigma],\langle l\mid s\rangle,q|\!)}$$

$$\text{(BIND-MTD)}\quad \frac{p = bind(o,m,\overline{v},f)}{o(\!|a,\langle l\mid s\rangle,q|\!)\parallel \langle o.m(\overline{v})\rangle_f \rightharpoonup o(\!|a,\langle l\mid s\rangle,p::q|\!)}$$

$$\text{(NEW-OBJECT)}\quad \frac{fresh(o')\quad a' = atts(C,[\![\overline{e}]\!]_\sigma,o')}{o(\!|a,\langle l\mid x=\mathbf{new}\ C(\overline{e});s\rangle,q|\!) \rightharpoonup o(\!|a,\langle l\mid x=o';s\rangle,q|\!)\parallel o'(\!|a',idle,\emptyset|\!)}$$

$$\text{(ASYNC-CALL)}\quad \frac{[\![e]\!]_\sigma = o'\qquad fresh(f)}{o(\!|a,\langle l\mid x=e!m(\overline{e});s\rangle,q|\!) \rightharpoonup o(\!|a,\langle l\mid x=f;s\rangle,q|\!)\parallel \langle o'.m([\![\overline{e}]\!]_\sigma)\rangle_f \parallel \langle\perp\rangle_f}$$

$$\text{(RETURN)}\quad \frac{}{o(\!|a,\langle l\mid \mathbf{return}\ (e);s\rangle_f,q|\!)\parallel\langle\perp\rangle_f \rightharpoonup o(\!|a,idle,q|\!)\parallel\langle [\![e]\!]_\sigma\rangle_f}$$

$$\text{(READ-FUT)}\quad \frac{f=[\![e]\!]_\sigma}{o(\!|a,\langle l\mid x=e.\ \mathbf{get};s\rangle,q|\!)\parallel\langle v\rangle_f \rightharpoonup o(\!|a,\langle l\mid x=v;s\rangle,q|\!)\parallel\langle v\rangle_f}$$

$$\text{(AWAIT}_1\text{)}\quad \frac{[\![e]\!]_\sigma = f}{o(\!|a,\langle l\mid \mathbf{await}\ e;s\rangle,q|\!)\parallel\langle v\rangle_f \rightharpoonup o(\!|a,\langle l\mid s\rangle,q|\!)\parallel\langle v\rangle_f}$$

$$\text{(AWAIT}_2\text{)}\quad \frac{[\![e]\!]_\sigma = f}{o(\!|a,\langle l\mid \mathbf{await}\ e;s\rangle,q|\!)\parallel\langle\perp\rangle_f \rightharpoonup o(\!|a,\langle l\mid \mathbf{suspend};\mathbf{await}\ e;s\rangle,q|\!)\parallel\langle\perp\rangle_f}$$

$$\text{(SELF-SYNC-CALL)}\quad \frac{l(destiny)=f'\quad [\![e]\!]_\sigma = o\quad [\![\overline{e}]\!]_\sigma=\overline{v}\qquad fresh(f)\quad bind(o,m,\overline{v},f)=\langle l'\mid s'\rangle}{o(\!|a,\langle l\mid x=e.m(\overline{e});s\rangle,q|\!) \rightharpoonup o(\!|a,\langle l'\mid s';cont(f')\rangle,\langle l\mid x=f.\ \mathbf{get};s\rangle::q|\!)\parallel\langle\perp\rangle_f}$$

$$\text{(SELF-SYNC-RETURN-SCHED)}\quad \frac{l'(destiny)=f}{o(\!|a,\langle l\mid cont(f)\rangle,\langle l'\mid s\rangle::q|\!) \rightharpoonup o(\!|a,\langle l'\mid s\rangle,q|\!)}$$

Fig. 2. Operational semantics.

Synchronous self-calls directly transfer control from the execution of the current to the invoked method (and back when returning), bypassing the SUSPEND and ACTIVATE rules. Technically, a special *cont* instruction is here inserted at the end of the statement list of the new process in rule SELF-SYNC-CALL, which is then used to re-activate the caller process in rule SELF-SYNC-RETURN-SCHED. The expression *destiny* records the method's future; i.e., *destiny* stores the return address of the method activation.

2.3. ABS example

We provide a publisher-subscriber example in Fig. 3 to present the ABS language. `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. The code in line 6 expresses that a `Service` object invokes method `add` on a `Proxy` object through method `subscribe`. Similarly, the code in line 16 expresses that a `Proxy` object invokes method `add` on the next `Proxy` object through method `add`. Note that both cases contain asynchronous blocking calls of the form $f = e!m(\overline{e}); \ldots; x = f.\mathbf{get}$, where there are no suspension (scheduling) points in between. The caller of the `add` method is blocked until it receives the results. On the other hand, a `Proxy` publishes news only when it receives the news. When there is no arrival of news, the `Proxy` object meanwhile performs other actions. This interleaving behavior is achieved by the usage of **`await`** statement for explicit process synchronization.

3. Colored Petri nets

Petri nets capture true concurrency in terms of causality and synchronization [14,15], and consist of places, transitions, and arcs. Colored Petri nets (CPNs), a well-established form of high-level Petri nets [20], extend the basic Petri net formalism to enable the modeling of data [13,21]. In CPNs, the data values are multisorted or *typed*. Types, representing sets of values, are called *color sets* in CPN terminology, and individual values are seen as colors. A type can be arbitrarily complex, defined by many sorted algebra in the same way as abstract data types. Each place in a CPN is typed; i.e., a place has an associated color set which determines the kind of data ("colored tokens") the place can contain. Tokens in a typed place represent individual values of that type. In the following, we formally introduce CPNs in their basic form, without hierarchies. Hierarchies, which enrich the nets with modularity for practical reasons, require a more complicated though equivalent definition. The basic definition of CPNs suffices for our purposes as any hierarchical CPN can be unfolded to a semantically equivalent non-hierarchical CPN [13].

```
class Service(Int limit, Producer prod) {
  Proxy proxy = new Proxy(limit,this,prod);
  Proxy lastProxy = proxy;

  Void run() { this!produce() }
  Void subscribe(Client cl){Fut<Proxy> f; f = lastProxy!add(cl); lastProxy = f.get }
  Void produce(){proxy!start_publish() }
}

class Proxy(Int limit, Service server, Producer prod) {
  List<Client> myClients = Nil; Proxy nextProxy;

  Proxy add(Client cl){ Proxy lastProxy = this; Fut<Proxy> f';
    if length(myClients) < limit {myClients = append(myClients, cl)}
    else {if nextProxy == null {nextProxy = new Proxy(limit,server,prod)} else {skip};
            f' = nextProxy!add(cl); lastProxy = f'.get};
    return lastProxy }

  Void start_publish(){ Fut<Proxy> f''; f'' = prod!detectNews(); await f''?;
                        News ns = f''.get; this!publish(ns) }

  Void publish(News ns){ myClients!signal(ns);
    if nextProxy == null {server!produce()} else {nextProxy!publish(ns)} }
}

{ Producer prod = new Producer();
  Service s = new Service(3,prod);
  Client c1 = new Client(); s.subscribe(c1);
  Client c2 = new Client(); s.subscribe(c2);
  News n1 = new News(); prod.add(n1)
  Client c3 = new Client(); s.subscribe(c3);
  Client c4 = new Client(); s.subscribe(c4);
  News n2 = new News(); prod.add(n2) }
```

Fig. 3. Implementation of the publisher-subscriber example. We provide the implementation of Service class, Proxy class and the main block.

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

1. *places* P and *transitions* T are disjoint finite sets;
2. A is the set of *arcs*, such that $A \subseteq (P \times T) \dot{\cup} (T \times P)$;
3. Σ is a finite set of *types* (where each individual type is seen as a non-empty “color set”);
4. *typed variables* V form a finite set, i.e., $\mathit{type}(v) \in \Sigma$ for all $v \in V$;
5. a *coloring* $C : P \to \Sigma$ associates a type to each place.
6. *labeling functions* $G : T \to \mathit{Expr}_V$ and $E : A \to \mathit{Expr}_V$ associate expressions with free variables from V to transitions, resp. to arcs. Expressions associated with transitions are called guards, and we write e and g for expressions resp. guards;
7. an *initialization function* $I : P \to \mathit{Expr}_\varnothing$ associates an expression without free variables to every place.

The “static” structure of a Petri net forms a graph, with the places and transitions as nodes and the arcs as edges. From the conditions 1 and 2 of Definition 1, places, transitions, and arcs of a Petri net form a directed, bi-partite graph: An arc (p, t) is outgoing for a place p and incoming for a transition t, whereas an arc (t, p) is incoming for p and outgoing for t. The guards associated with transitions express synchronization conditions which, together with the expressions on the arcs, capture the transition semantics of CPNs. Since tokens are individually typed values and expressions (including guards) contain free variables, the *enabledness* of transitions depends on the choice of values for the free variables.

We assume that expressions are appropriately typed, as follows. Guards for transitions are Boolean expressions (i.e., for result type of a guard g, $\mathit{type}(g) = \mathit{Bool}$). Expressions for arcs, on the other hand, are “multi-set typed”: Assume an arc connected to a place p (either as source or as target of the arc) with $C(p)$ as the type of p. Then the expression labeling the arc has multi-sets over T as resulting type, (i.e., $\mathit{type}(e) = C(p) \to \mathbb{N}$). With the “tokens” of the classical Petri nets now generalized to (appropriately typed) closed expressions, the initialization function attaches a multiset of such closed expressions to each place of the net, such that $\mathit{type}(I(p)) = C(p) \to \mathbb{N}$, for all places p. The initialization function corresponds to the so-called initial marking of the net (for the definition of markings, see below).

Example 2 *(Colored Petri nets).* Let’s use the very small net from Fig. 4 to illustrate the definition of CPNs and related concepts. The example has two places p_1 and p_2 and one transition t, connected by two arcs. $C(p_1)$ and $C(p_2)$ are respective types

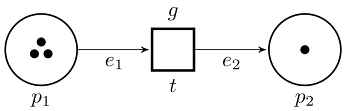


Fig. 4. Illustration of CPN.

from Σ of the two places (attached to the places by the "coloring" function C). The two arcs carry expressions e_1 and e_2, correspondingly, and the transition is decorated by a guard g. The black "bullets" in the places are *tokens* as in classical Petri nets; for CPNs, tokens are individual values and each token on p_1 and p_2 must have a data value that belongs to $C(p_1)$ and $C(p_2)$, respectively. See also Example 4 below.

Tokens in a CPN correspond to (values of) closed expressions. The guards on transitions and the expressions attached to the arcs, which together govern the dynamic behavior of a CPN, contain free variables, so their interpretation depends on choosing values for those variables. Variable bindings (or variable assignments) b are mappings from variables to values; we assume that bindings respect the types of the involved variables. Let $Var(t)$ denote the variables of a transition (where $Var(t) \subseteq V$), which are the variables in the guard of t and the variables of the arc expressions of the arcs connected to t. The binding of a transition covers all variables from $Var(t)$. Let $[\![e]\!]_b$ denote the value of expression e under variable binding b. A binding satisfies a guard expression g if $[\![g]\!]_b$ evaluates to true.

The current state or configuration of a CPN is given by a so-called *marking* and the dynamic behavior of such a net is described by sequences of *steps*.[1] Markings associate multi-sets of appropriately typed values to places. Steps transform markings in the way specified by the expressions on the arcs and under the condition, that the guards of the concerned transitions are satisfied. Given a CPN, a *marking* M attaches to each place p a multiset of appropriately typed values, i.e. $M(p) : C(p) \to \mathbb{N}$. As mentioned shortly earlier, the net's initialization function I is used to provide the *initial marking*: $M_0(p) = [\![I(p)]\!]$. Note that the expressions in $I(p)$ don't contain variables, hence $[\![I(p)]\!]$ is well-defined without a variable binding (resp. under the empty variable binding).

A *step* is a selection of a subset of the net's transitions together with appropriate bindings for the variables of these transitions such that the selected transitions are enabled, as defined below. Technically, a *step* Y is a function of type $Y : T \to ((Var \to Val) \to \mathbb{N})$, assigning ("selecting") a multi-set of bindings to each transition, where for all transitions t and all bindings b from $Y(t)$, $[\![G(t)]\!]_b = true$. As usual, the bindings are assumed to be consistent with the typing and additionally, must cover the variables of the transition. It's required that the step is non-empty in that for at least one transition, the multi-set of bindings is non-empty (an empty step Y would correspond to a stutter-transition without effect). Note that the notion of step is defined *independent* from a marking, i.e., independent from the current configuration of a net. Steps can be seen equivalently as a multi-set of pairs of transitions and bindings, i.e., of type $(T \times (Var \to Val)) \to \mathbb{N}$. We call the elements (t, b) of such multi-sets also *binding elements.*

To use a *step* to transform a marking into a successor marking, not only must the selected bindings satisfy the guards of the selected transitions. In addition, the marking prior to doing the step must assign "enough" tokens (i.e., values) to the places feeding into to the selected transitions. This is captured by expressions $E(p, t)$ on the arcs of the selected transitions.

Definition 3 *(Enabledness of transitions and steps).* Let $\subseteq_m$ denote the ordering relation over multisets. A transition t is *enabled* for binding b in a marking M if

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

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

In abuse of notation, we use the multi-set comparison symbols $\subseteq_m$, $\supseteq_m$, etc. also for comparing markings in a pointwise manner, i.e., $M_1 \subseteq_m M_2$ iff $M_1(p) \subseteq_m M_2(p)$, for all places p. Note that the condition $[\![G(t)]\!]_b = true$ for guard satisfaction is not needed for the enabledness of steps, as the notion of enabling of a binding already requires that. When t is enabled for b in M, the transition may *occur* or "fire" (given b, leading to the marking M' where $M'(p) = (M(p) - [\![E(p,t)]\!]_b) + [\![E(t,p)]\!]_b$, for all places p, and where $+$ and $-$ on the multiplicity of the elements correspond to multi-set union and multi-set difference (where for the latter $S_1 - S_2$ is defined only if $S_2 \subseteq_m S_1$, which is assured by the condition on steps for being enabled). 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

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

[1] One also finds the words "token distribution" and (non-empty) "binding distribution" as alternative terminology for markings and steps.

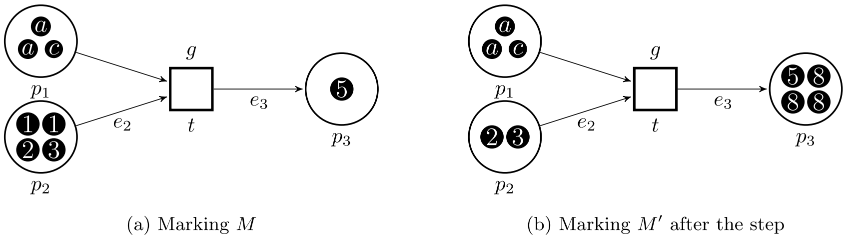


Fig. 5. Illustration of markings, bindings, and steps.

We also need to refer to the projection of an occurrence sequence onto the involved transitions, i.e., omitting bindings and markings: an *occurrence word* is a finite sequence of transitions, such that there exist bindings and markings such that it gives rise to an occurrence sequence as in equation (1).

The "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 pair of transition t and binding b, is denoted $\xrightarrow{t,b}$. A step semantics restricted to such single transition steps is equivalent to the unrestricted semantics, but corresponds to "interleaving concurrency". In the rest of the paper we consider such single transition steps and, when obvious, the binding b is omitted.

Example 4 *(Bindings, markings, steps).* Let's use Fig. 5 to illustrate the introduced concepts. Assume place p_1 on the upper left is typed by a type $C = \{a, b, c\}$, and the other two places, p_2 and p_3 are typed by the natural numbers. As for the expressions on the arcs, assume the following: let variables x and y be typed by natural numbers and z_1 and z_2 by the three-element type C.

$$\begin{aligned} e_2 &= 2 \times_m y \\ e_3 &= 3 \times_m (4x) \\ g &= z_1 \neq z_2 \wedge x \geqslant y \end{aligned}$$

Multi-set "multiplicity" is stated by the natural number left to $\times_m$. The arc from p_1 to t is not decorated by an expression e_1, which is assumed as $e_1 = 0 \times_m z$.

Assume further the following two bindings

$$b_1 = [x \mapsto 2, y \mapsto 1, z_1 \mapsto a, z_2 \mapsto b] \quad \text{and} \quad b_2 = [x \mapsto 3, y \mapsto 2, z_1 \mapsto a, z_2 \mapsto b].$$

Both bindings satisfy the guard g, i.e., $[\![g]\!]_{b_1} = \mathit{true}$ as well as $[\![g]\!]_{b_2} = \mathit{true}$. For the marking M given in Fig. 5a, though, only b_1 enables the transition t. This is due to the arc (p_2, t), whose expression e_2 requires a multiplicity of 2 or more for any choice of y, and in the current marking, that requires $y \mapsto 1$. Arc (p_1, t) does not impose any restrictions on the presence of values in the marking for p_1. Taking (t, b_1) as a step, we get $M \xrightarrow{t,b_1} M'$, with M' given in Fig. 5b on the right. Note that in p_3 in the post-marking M' contains 3 "copies" of a token 8, where $8 = 4x$ for $x = 2$ and with multiplicity 3, as specified by expression e_3 on the arc outgoing from transition t.

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 ideas of the translation in Section 4.1, we proceed to a more in-depth presentation: Sections 4.2 and 4.3 highlight crucial parts of how the ABS semantics is represented on the Petri net level, focusing on the active objects creation and the communication mechanism, respectively and, in the latter case, we focus on asynchronous method calls and the resolution of futures via `get` statement.

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. Remember that, among other questions, reachability in Petri nets is decidable (see [22] for a survey of decidability issues for (classical) Petri nets) but then again, whether data-abstracted ABS programs are Turing complete or not, is an open question. Still, there are two remaining sources of *infinity*: creation of (active) objects and creation of processes and accompanying future references via asynchronous method calls. It should be noted 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.

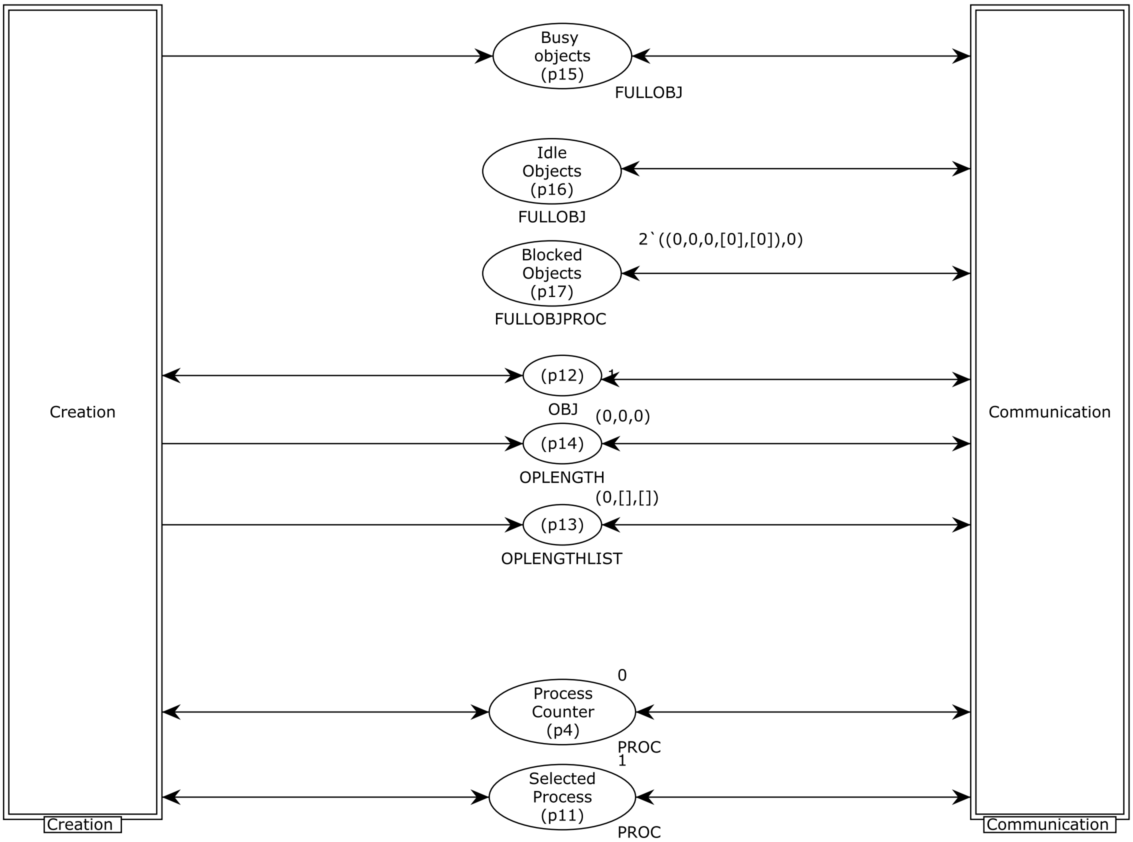


Fig. 6. Top-level module of the ABS semantics implementation in CPN Tools.

Roughly, each semantic rule from the operational semantics of Fig. 2 is represented by transitions and places, with appropriate types and guards. As a result, one particular program, resp. a 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 operate on typed values making it possible to represent the components of a configuration in a clean manner. For instance, objects, processes and futures are represented naturally by integers used as identifiers.

Notice that the implementation of the translation has been done in CPN Tools. In the following, we present this translation with figures that are screenshots from the implementation. We use hierarchical CPNs while Definition 1 of Section 3 introduces CPNs without hierarchies. This is done for the sake of simplicity at the level of the definitions. But note further that hierarchical CPNs can be reduced to non-hierarchical ones through unfolding, hence the two definitions are equivalent. Hierarchies can be seen as splitting a CPN into smaller parts. They often are practically more convenient, as they allow organizing the model into smaller parts or modules. This offers better understanding for large models and increases reusability, if parts of the net are needed repeatedly. Modules can be split further into submodules with a hierarchical structure. Submodules can be viewed as “hidden” parts of the net, appearing in the upper module as a black-box in the form of a so-called *substitution transition*.

Substitution transitions are not ordinary Petri net transitions, so they cannot fire as the normal transitions. They indicate the existence of a subnet from which they can be replaced. To distinguish them from standard transitions, they are drawn with double outer lines. As they represent subnets, they need to be connected to the rest of the net. This is done with some places that play the role of an input for the substitution transitions, or output, or both. They are distinguished from the other places by their double outer lines. Their role as an input, output, or input/output is indicated by a corresponding rectangular tag next to them. They appear for each submodule they are connected to but they share the same marking and they are used to connect different parts of the net. In other words, there is a “copy” of those places in each module they are connected to. Places of this kind are also called fusion places.

From now on, we refer to the whole of our implementation of ABS semantics in CPN Tools as CPN–ABS. Fig. 6 shows the top-level module of CPN–ABS. It consists of two substitution transitions, one related to the object creation and one related to the communication mechanism of ABS. Their corresponding submodules are “Creation” and “Communication” as shown in the tags below them. In the figure, we can also see some inscriptions next to some places, which refer to their initial marking. Recall that in CPNs tokens are typed (they have some “color”) according to the place they are located to and that

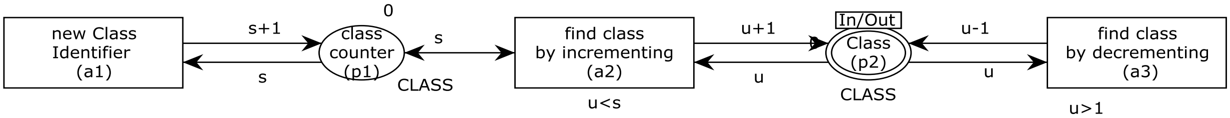


Fig. 7. CPN–ABS module for class identifier creation (*"Class"*).

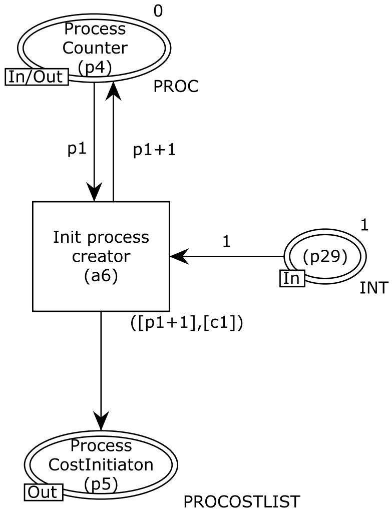


Fig. 8. CPN–ABS module for process creation (*"InitProcessCreator"*).

each place hosts a multiset of such tokens. Hence, its marking contains not only the tokens, but also the multiplicity of each token. For example, in the marking $2`((0,0,0,[0],[0]),0)$ number 2 is the multiplicity and the rest is the token of a color defined as a tuple of integers and lists of integers. When omitted, the multiplicity is trivially one, as for example on the markings $(0,0,0)$ of p_{14} and $(0,[],[])$ of p_{13}.

As we shall see in the following sections, the representation of active objects in the net is made by tokens carrying appropriate information. Their creation is related to the substitution transition *"Creation"*. Following the semantics of ABS, they can be located in exactly one among the places *"Busy Objects"*, *"Idle Objects"*, and *"Blocked Objects"*, which are connected to the substitution transition *"Communication"*.

Recall from Section 2 that the communication between objects is achieved through method calls. The rest of the places in Fig. 6 supports the soundness of the model keeping information related to the processes (i.e. they contain tokens acting as identifiers or achieving firing order of some transitions). In the implementation, we name most of the transitions and the places in a way reflecting their role in the ABS semantics. We also use an indexed naming scheme: p_i for the places and a_j for the transitions. This "naming" via integer indices is used mostly in Section 6 but also in the rest of the current section whenever this simplifies or shortens matters. Also, when referring to the marking of some place p_i, we will use the notation $M(p_i)$. Finally, also for simplicity, in the rest of this section, we omit referring to details like places, arcs and inscriptions which have an indirect relation with the semantics or with an obvious meaning.

4.2. Dynamic creation of active objects

In this section, we describe the modules for dynamic object creation. In the semantic rule of Fig. 2 the new objects are idle where in the model, each object is equipped with an initial process upon creation, which however leads to the same behavior. Recall from Section 2 that each object is an instance of some class. This class appears in the hypothesis New-Object rule of Fig. 2 as an argument in the auxiliary function *atts*. In CPN–ABS we represent classes by unique positive integer identifiers and Fig. 7 shows the CPN module associated with class creation. The uniqueness of the identifiers is guaranteed by a class counter of type integer: for each program class, transition a_1 increments the marking of p_1 by one, with an initial value of 0. This marking shows the number of the classes the program has and provides a constraint on the marking of p_2. As mentioned above, each new object is an instance of some class appearing in the hypothesis of the New-Object rule. In CPN–ABS, the marking of the place p_2 provides this class information to the new object. In particular, $M(p_2)$ can reach the value of the class identifier of the new object by firing a_2 or a_3 appropriately. For example, if the class id of the new object is 5 and $M(p_2)=2$, then a_2 should fire 3 times (noted in Section 6 as a_2^3). Similarly, if $M(p_2)>5$ then a_3 should fire $M(p_2)-5$ times. Remark that a_2 cannot make $M(p_2)$ exceed the maximum class id because of its guard $u<s$ and also that a_3 cannot decrease $M(p_2)$ to zero. In Fig. 8 we see the process creation mechanism. Place p_{29} has

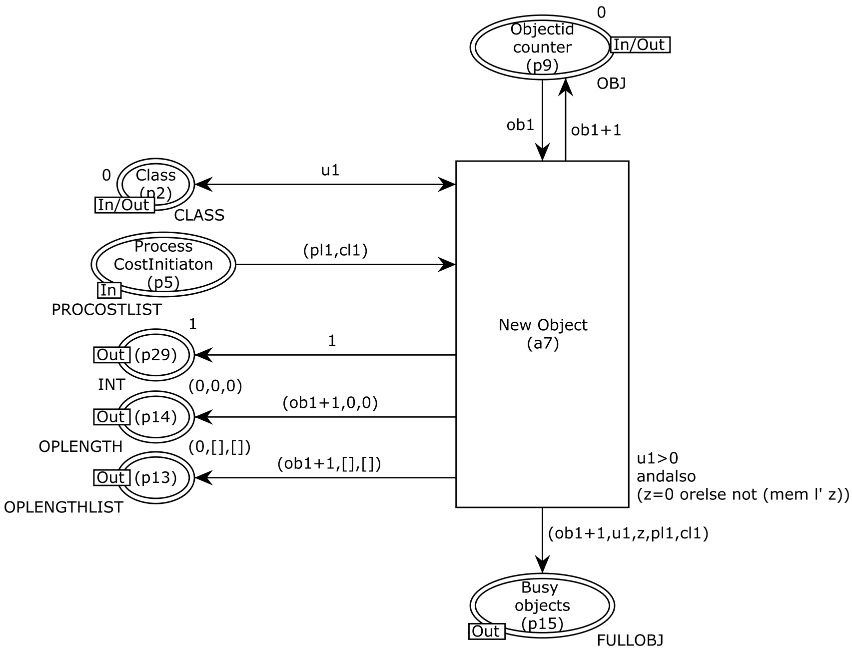


Fig. 9. CPN–ABS object creation (*"New Object"*).

initially one token. This enables transition a_6 which produces a new process id at place p_5. Place p_4 is a process counter similar to the class counter place p_1 of Fig. 7. Note that firing a_6 "consumes" the token of the place p_{29} and transition a_6 cannot fire again until it will appear again.

The token thereby produced in place p_5 enables transition a_7 of Fig. 9, which produces the tokens representing the objects in the translation. More concretely, a_7 creates a fresh object id (also represented as a positive integer) by adding one to the marking of the object id counter place p_9 (similarly to p_1 and p_4 of the previous cases). It also takes the information about the class id from place p_2 which is common with the module of Fig. 7. It also produces the token of place p_{29} common in Fig. 8. By producing and consuming the tokens of places p_{29} and p_5, the transitions a_6 and a_7 of the two figures alternate such that there is a new initial process for each new object. All the information taken from the markings of the places with incoming arcs to the transition a_7 are used in the tuple representing the object tokens in CPN–ABS. They are produced in place *"Busy Objects"* (p_{15}). An overview over object creation in CPN–ABS described above is shown in Fig. 10.

4.3. CPN–ABS communication mechanism

In CPN–ABS, communication takes place between objects represented as tokens which carry information about their identity, their class, and their process pool. These are represented as triples of the form $(id, class, q)$, where the process pool is implemented as a FIFO queue. CPN–ABS supports not only the construction of the information each object carries (hence dynamic creation of objects) as seen in Section 4.2, but also the communication between objects.

In the following, we concentrate on the CPN–ABS part related to the communication mechanism of ABS, associated to the operational rules Return, Activate, Suspend, Async-Call, Bind-Mtd, Read-Fut, Self-Sync-Call and Self-Sync-Return-Sched of Fig. 2.

Fig. 11 shows an overview of the communication mechanism. As mentioned above, there are three places where the object tokens can be located: *"Busy Objects"* (p_{15}), *"Idle Objects"* (p_{16}), and *"Blocked Objects"* (p_{17}). All three are fusion places, i.e., places that are common in more than one module and share the same marking. In ABS, when a method of an active object returns, it resolves a future (see rule Return from Fig. 2). In CPN–ABS, this behavior is simulated by the subnet related to the substitution transition *"Return"*, shown in Fig. 12: the object is removed from the place *"Busy Objects"* and added to the place *"Idle Objects"* and, on the same time, a token representing the resolved future is being produced at place p_{23}. Since place *"Busy Objects"* may contain more than one object, transition a_{19} should select the correct one. Recall that each object, contains variables representing (i) the identity of the corresponding object and (ii) the identity of the process executed the returned method. They should bind to the markings of places p_{11} and p_{12}. Figs. 13 and 14 show in detail how these markings can be appropriately changed to match the corresponding process and object identifiers. Their mechanism is similar to the one of the class identifier of Fig. 7 of the previous section: transitions a_{11} and a_{12}, as well as a_{14} a_{15} should fire the appropriate amount of times depending on the previous marking of the places p_{11} and p_{12}, respectively. Then, a_{19} can fire and, at the same time, the process related to the returned method is removed from the process list of the object

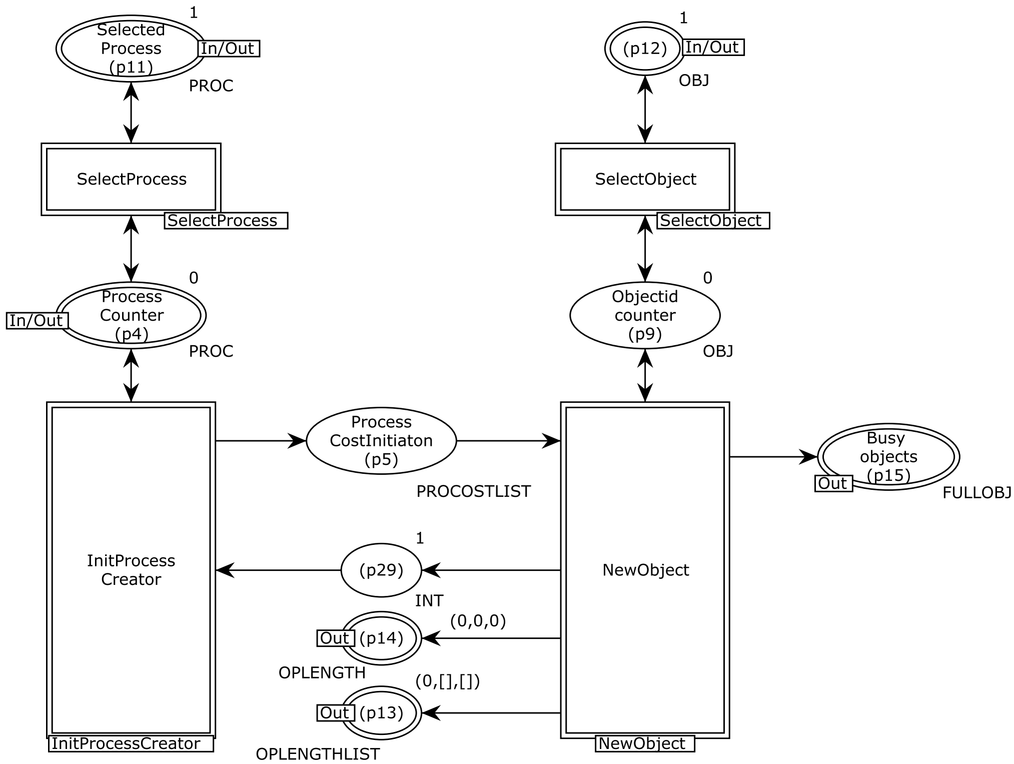


Fig. 10. CPN–ABS module for dynamic object creation (i) (*"Creation"*).

token, following the Return rule of the ABS semantics (see the inscription of the arc connecting a_{19} and p_{16}, where the head of the process list has been removed).

With the Activate rule of ABS, a process from the pool is activated and the object changes status from idle to busy and the inverse with the Suspend rule from Fig. 2. This is simulated by the CPN module shown in Fig. 15: In the case of Activate, transition a_{18} fires and moves the object from *"Busy Objects"* (p_{15}) place to the *"Idle Objects"* (p_{16}) one. The selection of the correct object is being done by matching the markings of the places p_{11} and p_{12} in the same way as for the case of the Return rule as described above (see Figs. 13 and 14) The inverse is for the Suspend rule and transition a_{17}.

Communication between objects involves two parts: the caller and the callee. The module from Fig. 16 covers the caller, where transition *"Caller"* selects the calling object from the *"Busy Objects"* place by matching the marking of the place p_{12} as in the previous cases. So, transitions a_{14} and a_{15} should fire first. There are two cases of communication through asynchronous method calls: immediately followed by a **`get`** statement or not. Both are simulated by firing the transition a_{24} (*".get"*). It is connected to the place p_{20} which is of colorset *Bool*. The color of its token is related to the presence of the **`get`** statement in the obvious way. Firing the transition *".get"* alternates the value of the token. So, transition *"Caller"* (a_{20}) takes the information on whether the asynchronous call is followed by a **`get`** statement or not. In the latter case, i.e., when the value of b_1 is false, the transition *"Caller"* maintains the object in the *"Busy Objects"* place following the Async-Call rule of the semantics; otherwise it sends the caller object to the *"Blocked Objects"* place until the waiting future has been resolved.

In the module of Fig. 17 we can see the details about the communication concerning the callee object. As the places related to the status of an object are disjoint, the callee object can reside only in one of the three corresponding places. Therefore, one among the transitions *"Idle callee"* (a_{21}), *"Active callee"* (a_{22}) and *"Blocked callee"* (a_{23}) can fire each time for the selected object. Recall that in CPN–ABS, the process pool is implemented as a FIFO queue. As a result, all three of the transitions that refer to the callee object, update the process queue by adding a new process related to this particular method call at the end of the list (see Bind-Mtd rule of Fig. 2).

A detailed view of how the future resolution is simulated according to the ABS rule Read-Fut is shown in Fig. 18: When a_{25} fires, it matches the value of the future token of place p_{23} with the expected future appearing as a variable p_{19} on the inscription of the arc connecting the *"Blocked Objects"* place with a_{25} and it moves the object token from the *"Blocked Objects"* place to the *"Busy Objects"* one.

CPN–ABS supports synchronous reentrant self calls as shown in Fig. 19 corresponding to the operational rules Self-Sync-Call and Self-Sync-Return-Sched from Fig. 2. Object selection is being done by firing a_{14} and a_{15} the appropriate amount of times as in the previous cases in order to update the marking of p_{12}. When a_{16} fires, a new process is added to the head

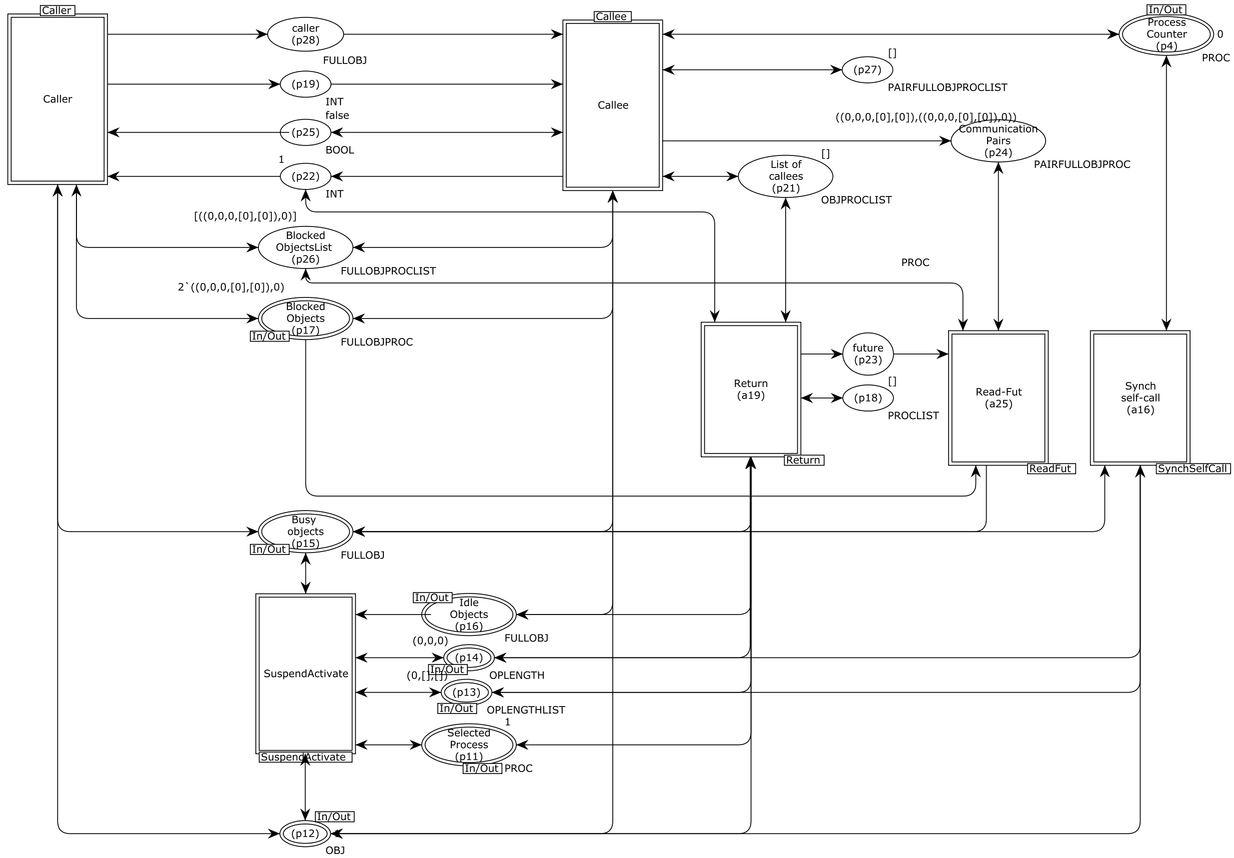


Fig. 11. Communication mechanism of CPN–ABS (*"Communication"*).

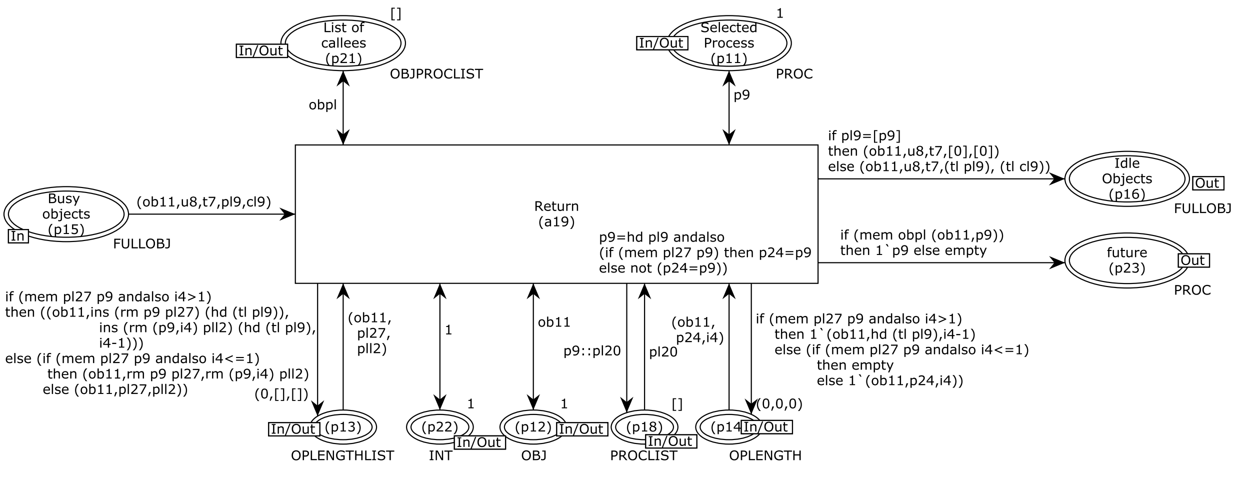


Fig. 12. Module for the return of a method (*"Return"*).

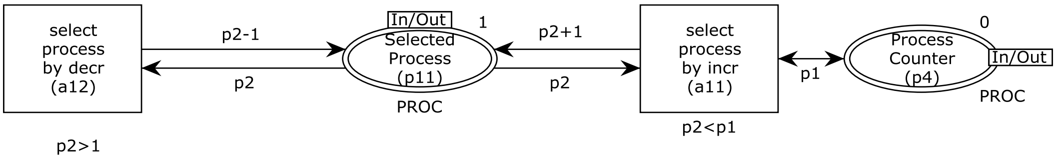


Fig. 13. CPN–ABS module for process selection.

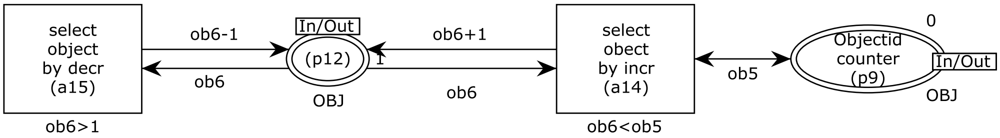


Fig. 14. CPN–ABS module for object selection.

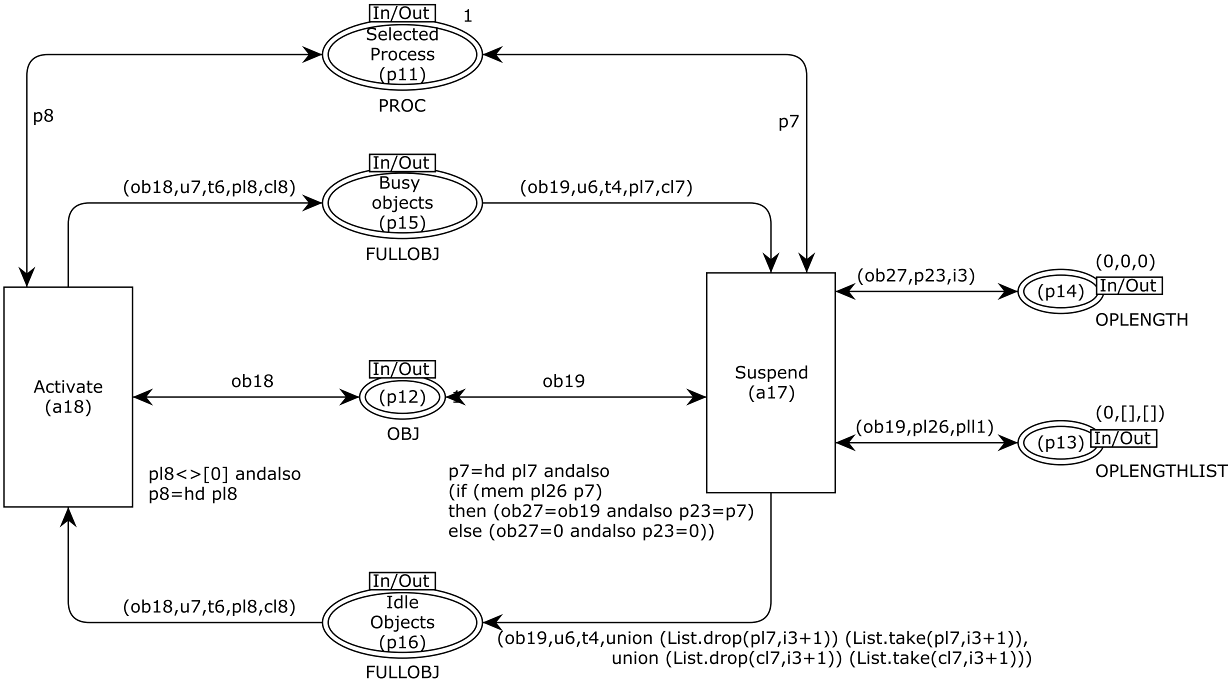


Fig. 15. Module for Activate and Suspend in CPN–ABS (*"Suspend-Activate"*).

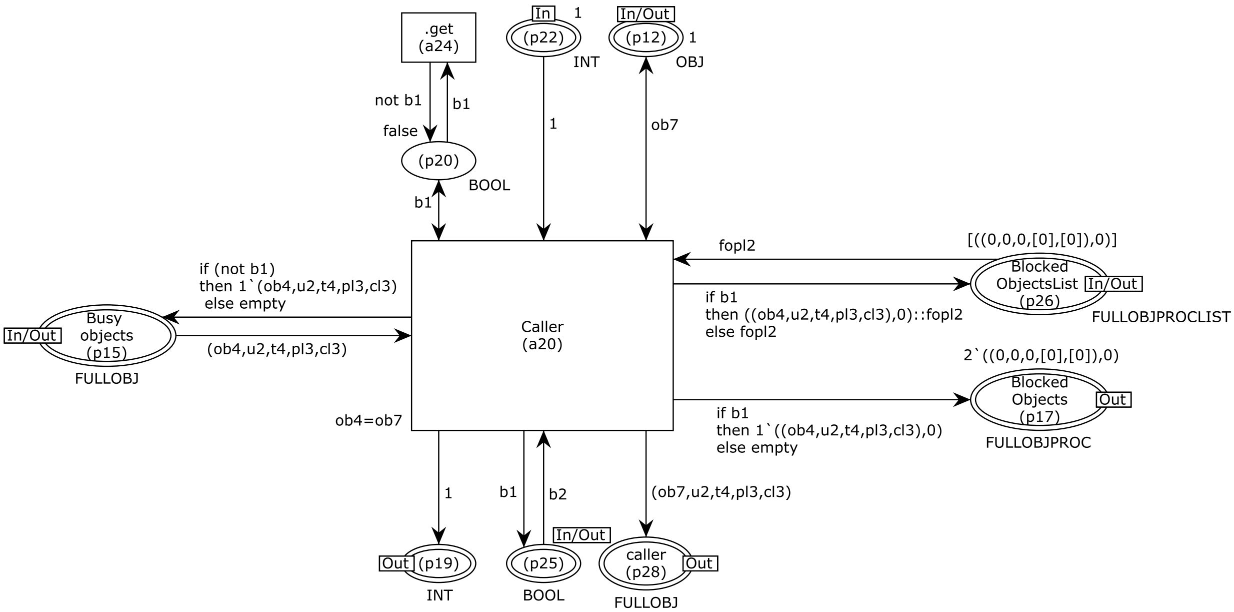


Fig. 16. Module for a method caller (*"Caller"*).

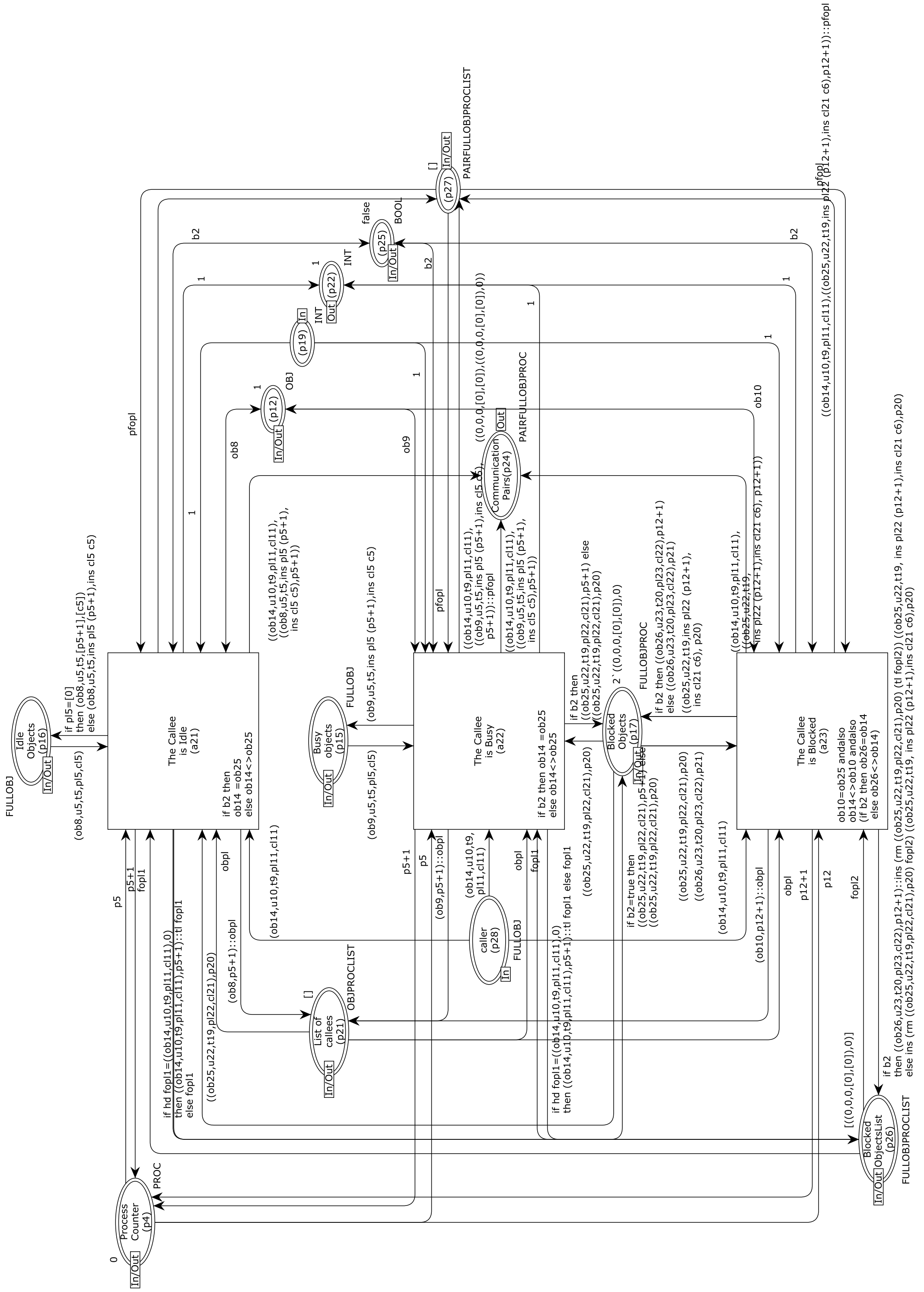


Fig. 17. Module for a method callee (*"Callee"*).

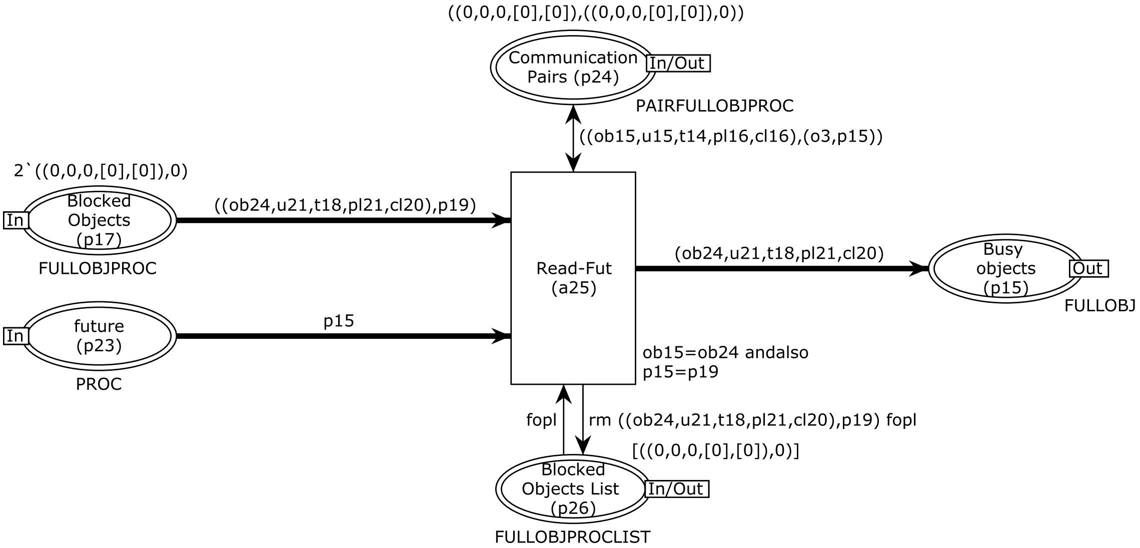


Fig. 18. Module for the future resolution (*"Read Fut"*).

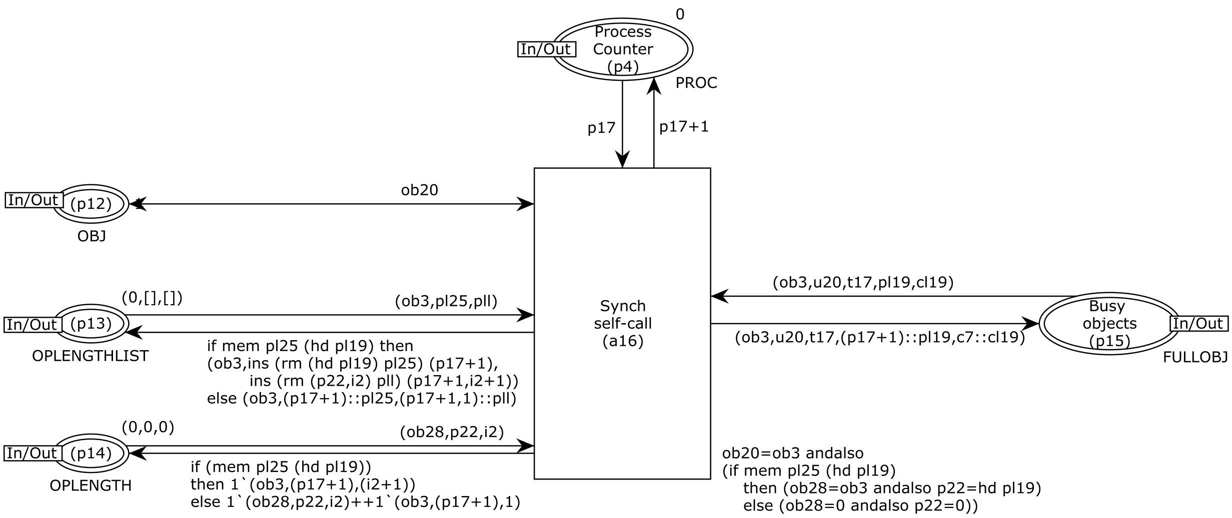


Fig. 19. Module for synchronous self-call in CPN–ABS (*"Synch self-call"*).

of the process list of the object (located in *"Busy Objects"* place). In addition, the marking of place p_4 is updated and hence the (global) process counter is aware of the new process.

5. The abstraction function

In this section, we define a translation from ABS configurations to CPN–ABS markings. In its core, it is 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 CPN–ABS marking over-approximates the original behavior, due to this form of data abstraction.

The translation is given in the form of an abstraction function α. Note that in ABS, a configuration is a multiset of objects, invocation messages and futures [12] and each object contains its identifier, an active process and a process pool. In the following, we define the abstraction function which selects all this information from a configuration and maps it at the abstract level in the form of CPN–ABS tokens. Then, in Section 6, we prove that those markings abstractly simulate ABS program behaviors.

Let $\mathcal{C}$ be the set of the configurations of an ABS program. Let also *Obj* be the set of the objects, *Class* the set of its classes, *Proc* the set of the processes, *Msg* the set of the method invocation messages and *F* the set of the resolved futures. Then, we define the functions that project the above sets from ABS configurations as follows:

$$\begin{tikzcd} c \arrow[rrr, Rightarrow] \arrow[d, "\alpha"'] & & & c' \arrow[d, "\alpha"] \\ \alpha(c) \arrow[r, "*"] & M' \arrow[r, dash] & \subseteq_m \arrow[r, dash] & \alpha(c') \end{tikzcd}$$

Fig. 20. Abstract (weak) simulation relation between ABS configurations and CPN–ABS markings.

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

We define the following injections from the above sets to the set of the positive integers: $h : Obj \to \mathbb{Z}^+$, $d : Class \to \mathbb{Z}^+$, and $g : Proc \to \mathbb{Z}^+$. Function h is an injection from the set of objects *Obj* of a ABS program to the set of positive integers representing the object identifiers, whereas d and g returns the unique identifiers of classes and processes, respectively. 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*. 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 (class identifier) of type *Int* and q is the process pool of the object of type *list of integers LInt*. Those object-tokens can be located in places corresponding to the particular status of the object (idle, active or blocked). CPN–ABS also supports other useful information taken from the configurations which are necessary for the communication between the objects, as for example which process has been created after a method invocation or which process corresponds to a resolved future related to a **get** statement. In both cases, processes are represented as tokens of type *Int* but this information comes from different parts of the concrete configuration (messages and futures), hence we use different functions for its extraction.

Now, we can define the abstraction function α, mapping ABS configurations to CPN–ABS tokens carrying the information discussed above. In the following, P is the set of the places and $M(p)$ is the marking of a place p in CPN–ABS. Then, for all configurations $c \in \mathcal{C}$:

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

where, $\bigcap$ denotes intersection over sets of multisets. With the above equation we define the abstraction of an ABS configuration as the intersection of the CPN–ABS markings containing (i) the objects (second line of equation (2)), (ii) the invocation messages of the method calls, if any (third line of equation (2)) and (iii) the resolved future from a method call containing a **get** statement (fourth line of equation (2)). We used composition of the functions defined earlier in this section to obtain the appropriate color of the CPN–ABS tokens, starting from ABS (concrete) configurations. The existential quantifiers for the places mean that the above tokens can be located at different places depending every time on the configuration. For example, place p can be either *"Busy Objects"*, *"Idle Objects"* or *"Blocked Objects"*, since each object can be located to one of them depending on the configuration. 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. As we will prove in the next section, CPN–ABS *abstractly simulates* ABS programs. As a consequence, there exist "extra" markings in CPN–ABS which are not important at the level of the abstraction though they are structurally important for the model.

6. Soundness proof of the translation

This section proves the soundness of the translation. Since the translation from Section 4 involves abstraction on data, the result of the translation over-approximates the behavior of the ABS program and the soundness of the construction is proven in a standard manner by a *simulation* relation between the small-step operational semantics of ABS and the transitions of CPN–ABS.

In particular, we prove that, for any ABS configuration c, if $c \twoheadrightarrow c'$, then there exists a marking M' and an occurrence *sequence* $\alpha(c) \to^* M'$ such that the diagram from Fig. 20 commutes.

The ABS semantics as such will *structurally* be translated into one global CPN, but the dynamic behavior executing an individual rule gives rise to a finite sequence of steps in the resulting CPN, as depicted in the simulation of Fig. 20. Remember from equation (1), that an occurrence sequence is a sequence of markings and steps where we will focus on singleton steps of the form (t, b), with t being a transition and b a binding. In the translation, the run-time information, i.e., the bindings and the markings, are not in the picture yet. Omitting that dynamic information from an occurrence sequence, we called such a sequence an *occurrence word*.

So, each transition step from c to c' is justified by one rule of the operational semantics, and such steps are thereby translated into a sequence $\vec{t} = t_1 t_2 \ldots t_k$ of *transitions* from T of the given net. To establish the simulation relation therefore means to prove that the transitions from $\vec{t}$ can in fact fire in the order given by the translation, in other words that $\vec{t}$ is an occurrence *word*. For occurrence sequences, remember the definition from equation (1) and that we focus here on "single transition" steps.

In the following we give this proof in detail. After some preliminary definitions on colored Petri nets in Section 6.1, we continue in Section 6.2 mapping each semantic rule of ABS to a sequence of CPN–ABS transitions. For that mapping, we prove that these sequences correspond to occurrence sequences, thereby establishing the relation of Fig. 20 (see Theorem 12).

6.1. Preliminaries

We start with some definitions and lemmas to achieve modularity for the construction. Let's call a transition t enabled in a marking M, if there exists a binding b s.t. (t, b) is enabled in M. Similarly, we write $M \xrightarrow{t} M'$ if $M \xrightarrow{(t,b)} M'$ for some b. Let $En(M)$ represent the set of *enabled transitions* for a given marking M, and $\mathcal{M}_{reach}$ the set of reachable markings of a net.

In the definition of the translation and the subsequent proof, we often refer to the figures showing the corresponding parts of the CPN, i.e., Figs. 6–18 from Section 4. The transitions in the figures are identified by labels $a_1 \ldots a_{25}$ and the places by $p_1 \ldots p_{29}$. So, when describing the translation later and in the proof, we use those labels to identify the transitions from the net. We use the transitions and their labels interchangeably, i.e., also speak of a "transition a_i" when a_i is the label as used in the tool. We also use the notion of occurrence words for sequences of labels, (not just for sequences of transitions). We write ϵ for the empty word, for instance, for the empty sequence of transitions.

In the proof later, some transitions are always enabled and can fire, when needed. That will be the case typically for transitions capturing a "generative" or "counting" nature.

Definition 5 *(Uniformly enabled transition).* A transition t is called *uniformly enabled* if, for any reachable M, $t \in En(M)$.

A transition, uniformly enabled in a given (reachable) marking, can be taken arbitrarily many times in a row:

Lemma 6 *(Uniformly enabled transitions). For a uniformly enabled transition t, t^* is an occurrence word.*

Proof. A straightforward consequence of the definition of uniform enabledness. □

Based on the notion of enabled transitions, we define a transition's successor in an occurrence sequence as follows:

Definition 7 *(Post-transitions).* The *post-transitions* of a transition $t \in T$ for a given reachable marking M is defined as $PostTrans(t, M) = \{t' \in En(M') \mid M \xrightarrow{t} M'\}$.

Lemma 8 *(Composability of occurrence sequences). An occurrence sequence $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_{n+1}$ is* composable *with another occurrence sequence $M'_1 \xrightarrow{t'_1} M'_2 \xrightarrow{t'_2} \ldots \xrightarrow{t'_m} M'_{m+1}$ producing an 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 $\bigwedge_{2 \leqslant i \leqslant m} [\![G(t'_i)]\!]_{b_i} = true$ and $M'_j \subseteq_m M''_j$, for all $2 \leqslant j \leqslant m+1$.*

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. □

For composition of occurrence words, we get as immediate corollary:

Corollary 9 *(Composition of occurrence words). The concatenation of two occurrence words is an occurrence word if the corresponding occurrence sequence is the composition of the occurrence sequences of the concatenated words.*

Proof. Trivial, from the labeling function and the composition Lemma 8. □

Table 1
Mapping from rules to sequences of transitions.

Rule	Sequence of transitions
SKIP, ASSIGN$_i$	ϵ
ACTIVATE	$a_{11}^{n_{11}} a_{12}^{n_{12}} a_{14}^{n_{14}} a_{15}^{n_{15}} a_{18}$
SUSPEND	$a_{11}^{n_{11}} a_{12}^{n_{12}} a_{14}^{n_{14}} a_{15}^{n_{15}} a_{17}$
ASYNC-CALL	$a_{14}^{n_{14}} a_{15}^{n_{15}} a_{24}^{n_{24}} a_{20} a_{14}^{n'_{14}} a_{15}^{n'_{15}} a_{21}^{n_{21}} a_{22}^{n_{22}} a_{23}^{n_{23}}$
RETURN	$a_{11}^{n_{11}} a_{12}^{n_{12}} a_{14}^{n_{14}} a_{15}^{n_{15}} a_{19}$
READ-FUT	a_{25}
NEW-OBJECT	$a_1^{n_1} a_2^{n_2} a_3^{n_3} a_6 a_7$
SYNC-SELF-CALL	$a_{14}^{n_{14}} a_{15}^{n_{15}} a_{16}$

6.2. Soundness of CPN–ABS

The colored Petri net CPN-ABS representing the ABS semantics has 25 transitions; we assume them labeled $a_1, a_2, \ldots, a_{25}$. The label that corresponds to each transition appears in the form (a_i) in each transition in the figures of the model. Recall that $M(p)$ is the marking of each individual place p, while M is the marking of the total CPN.

6.2.1. The translation function

The translation maps ABS each semantic rule of Fig. 2 to a sequence of net transitions, as shown in Table 1. Transitions here are represented by their labels a_i as shown in the figures. To establish the simulation relation, some of the transitions have to fire not a fixed number of times, but a variable number, where the number of iterations depends on the current run-time configuration in ABS (resp. the current marking in the CPN when doing the simulation). Instead of writing a^* for an arbitrary iteration of a transition labeled a, the mapping from Table 1 indicates this number when needed, writing $a_i^{n_i}$ where n_i indicates the number of times a_i needs to fire.

To define these numbers, we make use of the following notation: for two natural numbers n and n', let $n \dot{-} n'$ denote the non-negative difference, i.e., $n \dot{-} n' = n - n'$ if $n > n'$, and 0 otherwise. Often, the translation uses two numbers $m = n \dot{-} n'$ and conversely $\hat{m} = n' \dot{-} n$ to define the sequence $a^m \hat{a}^{\hat{m}}$. In such situations, at least one of a^m and $\hat{a}^{\hat{m}}$ equals the empty sequence ϵ (since at most one among m and $\hat{m}$ can be non-zero due to the law of trichotomy – according to which every number is either negative, positive, or else zero).

In the following definitions and the subsequent proof, we also make use of the following notation. In a number of cases, it will be an invariant for a place p to have exactly one value, i.e., for the multi-set $M(p)$, $|M(p)| = 1$. In these cases, we write also $M(p)$ to refer to that value (as opposed to its multiplicity, which is uniformly 1).

Now, for rules ACTIVATE and SUSPEND (covering also SYNC-SELF-CALL and partially ASYNC-CALL, as well), we set

$$
\begin{aligned}
n_{11} &= g(p) \dot{-} M(p_{11}) \\
n_{12} &= M(p_{11}) \dot{-} g(p) \\
n_{14} &= h(o) - M(p_{12}) \\
n_{15} &= M(p_{12}) - h(o) \ ,
\end{aligned} \tag{3}
$$

where place $p = select(q, a)$ according to the ACTIVATE rule. Recall from Section 5 that, in CPN-ABS, processes appear as (unique) natural numbers through the injective function $g : Proc \to \mathbb{Z}^+$ (where $Proc$ is the set of processes for each ABS program). In addition, $h : Obj \to \mathbb{Z}^+$ is an injective function from the set of objects Obj of the ABS program to the set of positive integers (i.e. for each object, it returns a unique identifier). The meaning of n_{11}, n_{11}, n_{11} and n_{11} is the repetition of firing the transitions a_{11}, a_{12}, a_{13} and a_{14} appropriately, as explained in Section 4.3.

For rule ASYNC-CALL, we set $n_{24} \in \{0, 1\}$. This is related to the presence of the **get** statement which makes the call blocking until the return of the waiting future. Numbers n'_{14} and n'_{15} are defined as n_{14} and n_{15}, respectively. Recall from Section 4.3 that these numbers applied to transitions a_{14} and a_{15} determine the selection of the correct object to several transitions, hence they have to fire first. The value of $h(o)$ for n_{14} and n_{15} refers to the caller, while for n'_{14} and n'_{15}, it refers to the callee object of the asynchronous method call. In the corresponding semantic rule of ABS, those values (i.e. the values of h^{-1}) are denoted as o and o' respectively for the caller and the callee. Furthermore, for $a_{21}a_{22}a_{23}$, the numbers are such that $n_{21} + n_{22} + n_{23} = 1$. Remark here that, $M(p_{15})$, $M(p_{16})$ and $M(p_{17})$ are pairwise disjoint (in particular $M(p_{15}) + M(p_{16}) + M(p_{17}) = |Obj|$ is a place invariant), so, at most one among the transitions a_{21}, a_{22} and a_{23} can fire. Cf. Fig. 17 for the part of the net representing the callee of a method.

Recall from Section 5 that $d : Class \to \mathbb{Z}^+$ is a function indicating the class where an object o belongs to (class identifier). For rule NEWOBJECT we set

$$
\begin{aligned}
n_1 &= d(o) \dot{-} M(p_1) \\
n_2 &= d(o) \dot{-} M(p_2) \\
n_3 &= M(p_2) \dot{-} d(o) \ .
\end{aligned} \tag{4}
$$

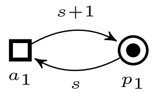
These are the numbers of repetitions that transitions a_1, a_2 and a_3 should fire, as described in detail in Section 4.2

6.2.2. Soundness

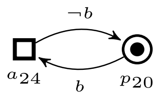
As the transitions of the target net are uniquely labeled by elements a_i, we use the labels interchangeably with the transitions, i.e. we write, for example, $a \in En(M)$ for $t \in En(M)$, etc. We also omit referring to the binding elements where obvious, writing, for example, $G(a)$ instead of $[\![G(t)]\!]_b$ and $E(p,a)$ instead of $[\![E(p,t)]\!]_b$. We write $|M(p)|$ for the number of tokens of the place p, i.e. the cardinality of the multiset $M(p)$.

Lemma 10 *(Uniform enabledness for a_1 and a_{24}). Transitions a_1 and a_{24} are uniformly enabled.*

Proof. For transition a_1 from Fig. 7, the relevant situation, i.e., the involved places and transitions look as follows, the loop being responsible to increment the counter:



The guard of p_1 is uniformly true, and a_2, the only other transition adjacent to p_1, leaves any marking of p_1 unchanged (the inscriptions on the incoming and outcoming arcs from p_1 to this transition coincide); no other arcs are adjacent to a_1. Hence, transition a_1 is uniformly enabled. For transition a_{24} from Fig. 16, the relevant portion of the nets looks similar



toggling the boolean token in p_{20} (initially being false). Again, the guard of the considered transition is true. Furthermore, transition a_{20}, also adjacent to p_{20}, leaves any marking of p_{20} unchanged, and since no other transition interferes with p_{20}, a_{24} is uniformly enabled, which concludes the argument. □

The next lemma is the core of the simulation argument, establishing, rule by rule, that the translation simulates the steps of the operational semantics.

Lemma 11. *For each rule of the operational semantics from Fig. 2, if the rule's hypothesis is satisfied, then the translation of the rule is an occurrence word.*

Proof. Proceed by case analysis on the rules of the operational semantics.

Case: Skip, Assign$_1$, Assign$_2$
These rules are translated to the empty sequence ϵ of transitions; hence their cases are immediate.

Case: Await$_1$, Await$_2$
These rules are syntactic sugar for Skip and Suspend; hence they are omitted from the proof.

Case: New-Object with translation $a_1^{n_1} a_2^{n_2} a_3^{n_3} a_6 a_7$.
Section 4.2 presents in detail how this sequence of transitions is related to the New-Object rule of the ABS semantics of Fig. 2. For the definition of n_1, n_2, and n_3, see equation (4). From that definition, either n_2 or n_3 is 0, since the marking of the place p_2 associated with those transitions (see Fig. 7) can be either greater or smaller than the class id of the object to be created (where class id is the translation of the parameter C of the New-Object rule in CPN–ABS).

We start by establishing that the first three transitions $a_1^{n_1} a_2^{n_2} a_3^{n_3}$ are an occurrence word. With a_1 being uniformly enabled by Lemma 10, thus, with Lemma 6, $a_1^{n_1}$ is an occurrence sequence.

Recall that n_1 is defined in equation (4) and $d(o)$ refers to the class identifier of o. After firing $a_1^{n_1}$ we know $M(p_1) \geqslant d(o)$ because p_1 is the class counter (for any object o, $d(o)$ cannot exceed this number). More precisely, for $n_1 \geqslant 1$, i.e., firing a_1 at least once, we know $M(p_1) = d(o)$ after $a_1^{n_1}$. In other words, this means that the class id we need was bigger than the marking of counter hence we incremented the counter by firing a_1 n_1 times. This is necessary to ensure that the marking of p_2 can reach the class id given by $d(o)$, since otherwise, the guard in the transition a_2 could not allow that. Recall from Section 4.2 that the marking of the place p_2 should match the value of the class id of the new object. As a result, either transition a_3 or transition a_2 should fire the appropriate amount of times until $M(p_2)$ reaches the value $d(o)$. So, we have to distinguish three subcases:

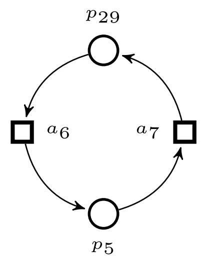
Subcase: $d(o) > M(p_2)$
Consequently $n_2 = d(o) - M(p_2)$ in this case. Note that the guard of a_2 corresponds to requiring $M(p_1) > M(p_2)$ as precondition for firing (the guard reads $u < s$, where u and s are the variables used in the arcs in Fig. 7). Furthermore, in this case, $n_3 = 0$ and $a_3^{n_3} = \epsilon$, and thus $a_1^{n_1} a_2^{n_2} a_3^{n_3}$ is an occurrence word.

Subcase: $d(o) < M(p_2)$

This time $n_2 = 0$ and $n_3 = M(p_2) - d(o)$. The guard of a_3 (stating $u > 1$) evaluates to true for all successive steps of a_3 (each decrementing the corresponding marking of place p_2 by 1). Choosing also $n_1 = 0$ makes $a_1^{n_1} a_2^{n_2} a_3^{n_3}$ an occurrence word.

Subcase: $d(o) = M(p_2)$
In this case, $n_1 = n_2 = n_3 = 0$, in which case $a_1^{n_1} a_2^{n_2} a_3^{n_3}$ is trivially an occurrence word.

Now for subsequent firing of a_6 (see Fig. 8). First, we prove that place p_4 always has a token. For place p_4 (representing a counter for processes), we have as invariant $|M(p_4)| = 1$: From the initial marking, $|M_0(p_4)| = 1$, and $\sum |E(p_4, t)| = \sum |E(t, p_4)|$ where the sums range over the expressions on the arcs adjacent to place p_4 (cf. Figs. 8, 13, 17 and 19). As it has been explained in Section 4.2, in order for a_6 to fire, it needs also a token from p_{29} (cf. Fig. 8 and for place p_{29}, see Figs. 8 and 9). Here follows a simplified picture of how transitions a_6 and a_7 are connected:



(transition a_6 can occur, either initially or only after an occurrence of a_7. Similarly, a_7 can occur only after an occurrence of a_6).

Now, for the firing of a_7, we prove similarly to the above that p_9 has always one token. For p_9 (representing a counter for objects, cf. Figs. 9 and 14, we have as invariant $|M(p_9)| = 1$: Initially $M_0(p_9) = 1$, and $\sum |E(p_9, t)| = \sum |E(t, p_9)|$, where the sums range over the expressions on the arcs adjacent to place p_9.

From the above, we can conclude that, after $a_1^{n_1} a_2^{n_2} a_3^{n_2}$, $M(p_2) > 0$ (cf. Fig. 10). Also, that a_6 can fire and that since $M(p_2) > 0$ the guard of a_7 is true and it can fire as well. As a result, the translation $a_1^{n_1} a_2^{n_2} a_3^{n_3} a_6 a_7$ of New-Object is an occurrence sequence, concluding the case.

Case: Activate with translation $a_{11}^{n_{11}} a_{12}^{n_{12}} a_{14}^{n_{14}} a_{15}^{n_{15}} a_{18}$
See Fig. 15 and Section 4.3 for description details. A similar argument as before for $a_2^{n_2} a_3^{n_3}$ shows that $a_{11}^{n_{11}} a_{12}^{n_{12}}$ as well as $a_{14}^{n_{14}} a_{15}^{n_{15}}$ are occurrence sequences, and so is their composition by Lemma 8. From the definition of the function g in Section 5 and the hypothesis $p = select(q, a)$ of rule Activate, there exists an object s.t. $h(o) \in M(p_{16})$, $h(o) \in M(p_{12})$, and $g(p) \in M(p_{11})$. Thus, the guard of a_{18} is true and a_{18} can fire. As a result, $a_{11}^{n_{11}} a_{12}^{n_{12}} a_{14}^{n_{14}} a_{15}^{n_{14}} a_{18}$ is an occurrence word, which concludes the case.

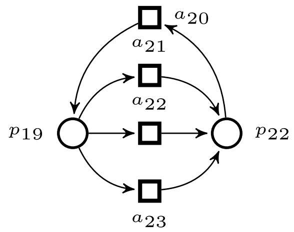
Case: Suspend with translation $a_{11}^{n_{11}} a_{12}^{n_{12}} a_{14}^{n_{14}} a_{15}^{n_{15}} a_{17}$
Similar to the previous case.

Case: Return, with translation $a_{11}^{n_{11}} a_{12}^{n_{11}} a_{14}^{n_{14}} a_{15}^{n_{15}} a_{19}$
See Fig. 12 and Section 4.3 for description details. Similar to the previous cases, making use of the invariant $|M(p_{18})| = |M(p_{14})| = |M(p_{21})| = |M(p_{22})| = 1$, which is easy to establish, as done in a previous case, for instance for $|M(p_9)|$.

Case: Read-Fut with translation a_{25}
Immediate (cf. Fig. 18).

Case: Async-Call and Bind-Mtd (taken as a single rule since they occur sequentially) with translation $a_{14}^{n_{14}} a_{15}^{n_{15}} a_{24}^{n_{24}} a_{20} a_{14}^{n'_{14}} a_{15}^{n'_{15}} a_{21}^{n_{21}} a_{22}^{n_{22}} a_{23}^{n_{23}}$
It's straightforward to establish that $a_{14}^{n_{14}} a_{15}^{n_{15}}$ and as well as $a_{14}^{n'_{14}} a_{15}^{n'_{15}}$ are occurrence words. In addition, a_{24} is a uniformly enabled transition, so its iteration $a_{24}^{n_{24}}$ is an occurrence word (cf. Lemma 6), and consequently, with Lemma 8, the composition $a_{14}^{n_{24}} a_{15}^{n_{15}} a_{24}^{n_{24}}$ is an occurrence word, as well.

Places p_{22} and p_{19} play similar role for alternating occurrences of a_{20} and one among a_{21}, a_{22} and a_{23} as it was the case for places p_{29} and p_5 and transitions a_6 and a_7 in the case for New-Object above. See Fig. 16 for a_{20} (a transition crucial for dealing with the behavior of the caller), and Figs. 11 and 17 for transitions a_{21}, a_{22}, and a_{23} (covering behavior of a callee), resp. the following:



In addition, we have as invariant $|M(p_{25})| = 1$. So, a_{20} can fire as soon as $h(o) = M(p_{12})$, which is the case after $a_{14}^{n_{14}} a_{15}^{n_{15}} a_{24}^{n_{24}}$. So, from Lemma 8, $a_{14}^{n_{14}} a_{15}^{n_{15}} a_{24}^{n_{24}} a_{20}$ is an occurrence word. As stated, also $a_{14}^{n'_{14}} a_{15}^{n'_{15}}$ is an occurrence word. Now since $M(p_{15})$, $M(p_{16})$, and $M(p_{26})$ are pairwise disjoint, only one among n_{21}, n_{22}, and n_{22} can be different from 0. In addition we have $|M(p_{21})| = |M(p_4)| = |M(p_3)| = |M(p_{25})| = |M(p_{25})| = |M(p_{26})| = |M(p)| = 1$, $M(p_{17}) \neq \emptyset$ and $|M(p_{28})| = 1$ after the occurrence of a_{20}. So, with Lemma 8, also $a_{14}^{n_{14'}} a_{15}^{n'_{15}} a_{21}^{n_{21}} a_{22}^{n_{22}} a_{23}^{n_{23}}$ is an occurrence word and then, so is $a_{14}^{i} a_{15}^{j} a_{24}^{t} a_{20} a_{14}^{i'} a_{15}^{j'}$ (again with Lemma 8).

Case: Sync-Self-Call and Self-Sync-Return-Sched (taken as a single rule since they occur sequentially) with translation $a_{14}^{n_{14}} a_{15}^{n_{15}} a_{16}$
Similar. See Fig. 19 and Section 4.3 for description details. □

With the simulation theorem that follows, the soundness proof of the translation ABS programs into colored Petri nets is completed.

Theorem 12 *(Simulation). CPN–ABS markings are in an abstract (weak) simulation relation with ABS program configurations.*

Proof. We need to prove that, for any ABS configuration c, if $c \twoheadrightarrow_r c'$ for some semantic rule $r \in Sem$, then there exists a marking M' and an occurrence word, such that $\alpha(c) \to M'$ and $\alpha(c') \subseteq_m M'$. This follows straightforwardly from the definition of the abstraction function α and from Lemma 11. □

7. Communication analysis

As we saw in detail in the previous section, CPN–ABS markings abstractly simulate ABS program configurations. By construction, CPN–ABS can follow the concurrency of ABS and contains its full communication mechanism (see Section 4). As we have already mentioned in Sections 1 and 4, CPN–ABS was implemented in CPN Tools, a model checker for colored Petri nets. This, together with the abstraction relation described in Sections 5 and 6, allows CPN–ABS to behave as an abstract interpreter for ABS programs. In particular, it overapproximates the communication topologies of ABS programs upon initializations arising from the application of the abstraction function to the (static view of the) program. This makes CPN–ABS useful for communication analysis of ABS programs. In the rest of this section we illustrate this by applying deadlock and livelock analysis. In particular, we express the above notions of concurrency in terms of CPN–ABS and explain how we can use it in order to detect deadlocks and livelocks.

7.1. Deadlocks

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.

Definition 13 *(Deadlocks in CPN–ABS).* In CPN-ABS, *there is a deadlock cycle [23] 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 $1 \leqslant i < n$, $p_i \in q_{i+1}$ and $p_n \in q_1$* (where q_i is the process queue of the ith object).

We detect the deadlock situation using the state space report of the model checker of the CPN Tool used to implement CPN-ABS.

Deadlock detection

We detect possible communication deadlocks fully automatically using the model checker of CPN Tools to construct the state space of the CPN model for a given ABS program. To enable this, we have implemented a query function in CPN Tools that extracts a directed graph from the marking of the place "Blocked Objects" describing the waiting conditions between

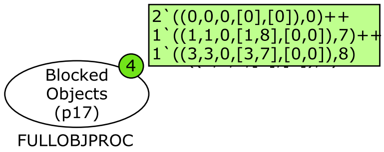


Fig. 21. Deadlock detection by CPN-ABS.

objects. The nodes of the graph represent objects and there is an edge from an object (node) o to an object (node) o' if o is waiting for o'. We then use Tarjan's algorithm for computing the strongly connected components of the graph in order to identify cycles. Any strongly connected component containing at least two nodes (objects) represents communication deadlocks (since any two nodes in a strongly connected component are mutually reachable and hence on a cycle). By extracting this directed graph in all reachable markings, we characterize the communication deadlocks of CPN-ABS (if any). Furthermore, using standard query functions of the model checker we are able to automatically construct an occurrence sequence starting from the initial marking to any marking containing a communication deadlocks.

If we let O be an upper bound on the number of objects $|Obj|$ in the ABS program, then we can extract the directed graph from the marking of "Blocked Objects" in $\mathcal{O}(O + O^2) = \mathcal{O}(O^2)$ time (in worst case all objects are mutually waiting for each other). As Tarjan's algorithm is linear in the size of the graph then strongly connected components can be computed in time $\mathcal{O}(O^2)$. In the worst case this computation must be done for all reachable markings $\mathcal{M}_{reach}$ giving a worst case time complexity of $\mathcal{O}(O^2 \cdot |\mathcal{M}_{reach}|)$ for deadlock detection.

We now use the publisher-subscriber example of Fig. 3 to illustrate how CPN-ABS detects communication deadlocks. By applying the model checker (CPN Tools 4.0) on an Intel i7 3.4 GHz running Windows 7, in less than 1 second we get the full state space report in which tokens of color $((o_1, Service, q), p)$ and $((o_3, Proxy, q'), p')$ 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. 21, 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. 21, 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. Recall that the abstraction function maps the information related to the objects and the processes of each configuration to the corresponding numbers appearing as arguments in the tokens. Hence, from the reachable marking, we can trace back to the program code and determine straightforwardly the deadlock represented by the call chain.

Note 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.

7.2. Livelocks

A *communication livelock* is a status of two unblocked objects, in which the generated processes cannot progress due to the busy-waiting of a process on another object. No progress is made at the suspension points, the scheduler is continuously activating and releasing processes. Such a situation may happen, for example, when many processes are competing for entering a critical section. In ABS, process suspension can be done explicitly through the corresponding statement provided by the syntax (**suspend**) or implicitly, through the **await** statement (see Fig. 1 and 2 of Section 2). Activation is an internal procedure, as described in Section 2 with the Activate rule.

The above transitions (suspension and activation) are considered as internal, hence they can be seen as τ transitions, since their effect is at the level of the scheduler rather than at the program state. Notice here that, whenever two τ transitions (concerning the same object) happen, they should be different, i.e. not both of them Suspend or both of them Activate. This is implied by the semantics of the language (see Fig. 2). For the sake of simplicity, we do not distinguish between them, while alternation between them from now on will be considered as obvious.

A livelock path for an object in ABS is an infinite path, where the only transitions related to the object are τ transitions and there exist infinitely many of them. In ABS, there is a *livelock* if there exist a reachable configuration, such that all infinite paths starting from it are livelock paths for some object and there exists at least one such an infinite path.

The CPN–ABS analog of the livelock can be defined in a similar way. Observe that the abstract version of the τ transitions described above (noted as τ_α) are the CPN–ABS transitions *"Suspend"* and *"Activate"* (labeled as a_{17} and a_{18} respectively). They are linked to the places *"Busy Objects"* (p_{15}) and *"Idle Objects"* (p_{16}) and, similarly to the ABS transitions, they alternate since they "move" the object-token from p_{15} to p_{16} and vice versa.

In Section 6 we defined the occurrence words over the labels of the transitions of CPN–ABS. So far, whenever obvious, we have omitted the binding from a transition firing. Here, bindings are important, hence occurrence words will be sequences of binding elements, i.e. sequences of pairs consisting of a transition label and the binding (a, b). When interested in the

```
class Client(Service server) {
  News ns = null;
  Void run(){ server!subscribe(this) }
  Void pay(){ Fut<Void> f1; f1 = server!subscribe(this); await f1?;
              Fut<Void> f2; f2 = server!receive(500); f2.get }
  Void signal(News n){ ns = n }
}

class Service(Int limit, Producer prod) {
  Proxy proxy = new Proxy(limit,this,prod);
  Proxy lastProxy = proxy;
  Int profit = 0;

  Void run() { this!produce() }
  Void subscribe(Client cl){ Fut<Void>f3; f3 = cl!pay(); await f3?;
                             Fut<Proxy> f4; f4 = lastProxy!add(cl); lastProxy = f4.get }
  Void produce(){ proxy!start_publish() }
  Void receive(Int amount){ profit = profit + amount }
}
```

Fig. 22. Implementation of the publisher-subscriber example.

binding of a subset of the variables of a transition t we will write it as $b(s)$, where $Var(()s) \subseteq Var(t)$. Below, we provide a definition for *livelocks in CPN–ABS*.

Definition 14 *(Livelocks in CPN–ABS).* A *livelock occurrence word for an object ob* is an infinite occurrence word ω such that $\omega_{|ob} = (\tau_\alpha, b(ob))^\infty$, where by $\omega_{|ob}$ we denote the projection of the occurrence word ω to the object *ob*. Let $\Omega(M) \stackrel{def}{=} \{\omega \mid \omega$ starts from marking $M\}$. Then, we say that there is a *livelock in CPN–ABS* iff all of the following conditions are satisfied:

- $\Omega(M) \neq \emptyset$
- all $\omega \in \Omega(M)$ are livelock occurrence words for some object *ob*
- $M \in \mathcal{M}_{reach}$.

Livelock detection

As in the case of deadlocks, CPN–ABS is able to detect possible ABS program livelocks such as, for example, in the version of the publisher-subscriber example of Fig. 22, where the yellow lines induce a livelock: Each client agrees to pay the subscription fee once the server grants the subscription. The server grants the subscription of a client only when the client pays the fee. Both are waiting for each other to act first.

Livelocks in CPN–ABS are detected based on the state space of the model by exploiting the support in CPN Tools for computing the Strongly Connected Component (SCC) graph of the state space. The SCC-graph has a node for each strongly connected component of the state space containing the states and their connecting arcs, and it has an arc from one SCC c_1 to an SCC c_2 whenever there is a state in c_1 with an outgoing arc to a state in the c_2. The SCC-graph is a directed acyclic graph and the basic idea in checking for livelocks is to identify states which have a livelock for an object o by conducting a bottom-up classification of the strongly connected components starting from the leaf (terminal) nodes of the SCC-graph.

A terminal SCC is classified as a *non-livelock SCC* for an object o iff it contains arcs corresponding to occurrences of binding elements for other than the suspend/activate transitions for the object o. A terminal SCC is classified as a *livelock SCC* for an object o iff the only occurrences of binding elements related to o in the SCC correspond to suspend/activate transitions. By construction of the CPN model occurrence sequences (paths) inside such a component will have alternating occurrences of suspend and activate. Hence, all states inside such a strongly connected component will be states that have a livelock for the object o. Finally, we classify an SCC as *livelock-neutral* if it does not contain any occurrences of binding elements for the given object.

We compute the classification of an SCC based on its outgoing arcs and the classification of its successor SCCs. An SCC is classified as a livelock SCC for an object o if:

- The SCC itself only contains occurrence of suspend and activate transitions for the object, all its successor SCCs are either neutral or livelock SCCs, and the arcs connecting the SCC to its successor SCCs concern other objects or suspend/activate for the given object; or
- the SCC does not contain any occurrences of binding elements for the given object, at least one successor SCC is a livelock SCC and the rest are either livelock or neutral SCCs, and the arcs connecting the SCC to its successor SCCs concern other objects or suspend/activate for the given object.

Similar conditions can be obtained for the cases of non-livelock and neutral SCCs and they are similar to how the terminal SCCs are classified.

When the SCC-graph has been computed in CPN Tools the classification is implemented by exploiting the API in CPN Tools for traversing and querying the SCC-graph. The SCC-graph is computed in linear time in the size of the state space, and to compute the classification we need to visit each strongly connected component (which is bounded by the number of nodes in the state space) and the arcs inside and between the components (which are bounded by the number of arcs in the state space). If we let O be an upper bound on the number of objects in the ABS program and A the number of arcs in the state space, then this gives a worst case time complexity of $\mathcal{O}(O \cdot (|\mathcal{M}_{reach}| + A))$ for livelock detection.

8. Conclusion and related work

We have developed an encoding of the formal semantics of ABS as a colored Petri net, such that a program is given as a marking for this net. The key idea in our encoding is to exploit the colored tokens such that our net can support dynamic program behavior and different programs can be represented without making changes to the net structure, but only requires changing the initial marking. We provided a detailed soundness proof for our encoding and showed how a model checker for colored Petri nets can be used for communication analysis of active objects in ABS considering detection of deadlocks and livelocks.

Livelock and deadlock detection is traditionally concerned with the usage of locks for thread-based concurrency. This line of work in the case of deadlocks is surveyed in [24], 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 [25], cost analysis [26], protocol specifications [23], and Petri nets [17]. Previous work on deadlock analysis for active objects using Petri nets [17] 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. This makes the approach approximative, in that 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. This makes our approach less error-prone and easier to automate as we only need a compiler that given an ABS program generates the corresponding initial marking of our CPN model. Our modeling approach is in this respect similar to the work in [27] and [28], where a CPN model was developed for execution of workflows and for simulation in the planning domain.

Deadlocks, livelocks, and similar concurrency-related errors have been widely studied, outside the Petri net community, as well. The calculus here studied a version of ABS without so-called object groups. Deadlock detection in the presence of that feature has been formalized in [29], in a calculus *FJg*, which builds upon the well-known Featherweight Java calculus and integrates asynchronous method calls as in ABS and object groups [30]. [31] investigates the analysis of synchronization problems and deallocks in an active-object calculus *gASP*, a sub-calculus of the ASP actor model [32]. The analysis is based on type-based techniques, using behavioral types and effects.

Petri nets and their extensions are popular formalisms to model and analyze systems with concurrency, communication, and synchronization [14,15]. Petri nets have in particular been applied to protocol and workflow analysis, but have also been used to study process algebra (e.g., [33,34]), and also including asynchronous communication [35]. CPNs and state spaces have also been used for deadlock analysis of Ada programs in [36], but in contrast to our approach this work did not involve livelock analysis and also employed a non-parametric CPN model. Approaches which encode programming language features into Petri nets have been developed for e.g. Ada [16] and Java [37], which focusses on how threads interact with a single synchronized object, and for choreography languages like Orc [38]. CPNs were used in [39] in order to visualize the execution of actor-based concurrent programs. 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. Petri nets was also used as a semantic foundation to support a concurrent programming model in [40], and colored Petri nets was used in order [41] to formally define actor semantics.

The work presented in this paper provides several directions for future work. In this paper, we have focused on communication and synchronization for ABS programs. ABS also supports the specification of real-time behavior, deployment architectures, and resource-aware systems [12]. One direction for future work is to extend the CPN model to cover also these language features, and explore the usage of colored Petri nets for resource analysis and to compare resource-management strategies for distributed ABS programs. For the real-time aspects, we may rely on earlier work on scheduling analysis for actor-based system [42]. For the communication analysis we used explicit state space exploration and model checking in its most basic form. For large ABS programs, we will inevitably encounter the state explosion problem during communication analysis. A direction for future work is therefore to investigate state space reduction methods and identify those that are most suited for the domain of ABS program and the communication properties we want to verify. Exploiting symmetries between objects [43] and local progress in the execution of ABS programs [44] are potential candidates in this direction. A third future direction is to further automate our approach by exploring the automatic generation of the initial marking for our CPN model directly from ABS program under analysis, and to be able to visualize any error-traces obtained from the communication analysis at the level of the ABS program being analyzed. The recent [45] investigates deadlock in connection with the notion of futures and synchronization based on forks and joins, in particular the question of deadlock *avoidance*, i.e., assuring dynamically that deadlocks do not occur. The investigations identify in that setting data races as the root cause for deadlocks and proposes policies to assure deadlock freedom with a corresponding tool for dynamic runtime deadlock prevention in a Java setting.

References

[1] G. Agha, C. Hewitt, Concurrent programming using actors, in: Object-Oriented Concurrent Programming, The MIT Press, 1987, pp. 37–53.
[2] G. Agha, ACTORS: A Model of Concurrent Computations in Distributed Systems, The MIT Press, Cambridge, Mass, 1986.
[3] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2007.
[4] P. Haller, M. Odersky, Scala actors: unifying thread-based and event-based programming, Theor. Comput. Sci. 410 (2–3) (2009) 202–220.
[5] F.S. de Boer, D. Clarke, E.B. Johnsen, A complete guide to the future, in: R. de Nicola (Ed.), Proc. 16th European Symposium on Programming, ESOP'07, in: Lecture Notes in Computer Science, vol. 4421, Springer, 2007, pp. 316–330.
[6] C.C. Din, R. Bubel, R. Hähnle, KeY-ABS: a deductive verification tool for the concurrent modelling language ABS, in: A.P. Felty, A. Middeldorp (Eds.), Automated Deduction – CADE-25 – 25th International Conference on Automated Deduction, Berlin, Germany, August 1–7, 2015, Proceedings, in: Lecture Notes in Computer Science, vol. 9195, Springer, 2015, pp. 517–526.
[7] C.C. Din, O. Owe, Compositional reasoning about active objects with shared futures, Form. Asp. Comput. 27 (3) (2015) 551–572.
[8] D. Caromel, L. Henrio, A Theory of Distributed Objects, Springer, 2005.
[9] E.B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: a core language for abstract behavioral specification, in: B. Aichernig, F.S. de Boer, M.M. Bonsangue (Eds.), Proc. 9th International Symposium on Formal Methods for Components and Objects, FMCO 2010, in: Lecture Notes in Computer Science, vol. 6957, Springer, 2011, pp. 142–164.
[10] F.S. de Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C.C. Din, E.B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, A.M. Yang, A survey of active object languages, ACM Comput. Surv. 50 (5) (2017) 76:1–76:39.
[11] K.M. Chandy, J. Misra, L.M. Haas, Distributed deadlock detection, ACM Trans. Comput. Syst. 1 (2) (1983) 144–156.
[12] E.B. Johnsen, R. Schlatte, S.L. Tapia Tarifa, Integrating deployment architectures and resource consumption in timed object-oriented models, J. Log. Algebraic Methods Program. 84 (1) (2015) 67–91.
[13] K. Jensen, L.M. Kristensen, Coloured Petri Nets – Modelling and Validation of Concurrent Systems, Springer, 2009.
[14] C.A. Petri, Kommunikation mit Automaten, Ph.D. thesis, Universität Bonn, 1962 (in German).
[15] W. Reisig, Petri Nets, EATCS Monographs in Computer Science, vol. 4, Springer, 1985.
[16] J. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Brückner, O. Roubine, B.A. Wichmann, Modules and visibility in the Ada programming language, in: On the Construction of Programs, Cambridge University Press, 1980, pp. 153–192.
[17] F.S. de Boer, M. Bravetti, I. Grabe, M. Lee, M. Steffen, G. Zavattaro, A Petri net based analysis of deadlock for active objects and futures, in: C.S. Pasareanu, G. Salaün (Eds.), Revised Selected Papers of the 9th International Workshop on Formal Aspects of Component Software, FACS 2012, in: Lecture Notes in Computer Science, Springer, 2013, pp. 110–127.
[18] A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, K. Jensen, CPN tools for editing, simulating, and analysing coloured Petri nets, in: Applications and Theory of Petri Nets 2003, 24th International Conference, ICATPN 2003, Eindhoven, The Netherlands, June 23–27, 2003, Proceedings, in: Lecture Notes in Computer Science, vol. 2679, Springer, 2003, pp. 450–462.
[19] A. Gkolfi, C.C. Din, E.B. Johnsen, M. Steffen, I.C. Yu, Translating active objects into Colored Petri Nets for communication analysis, in: M. Dastani, M. Sirjani (Eds.), Proc. 7th International Conference on Fundamentals of Software Engineering, FSEN 2017, in: Lecture Notes in Computer Science, vol. 10522, Springer, 2017, pp. 84–99.
[20] K. Jensen, Coloured Petri nets: a high level language for system design and analysis, in: G. Rozenberg (Ed.), Advances in Petri Nets 1990, in: Lecture Notes in Computer Science, vol. 483, Springer, 1991, pp. 342–416.
[21] K. Jensen, Coloured Petri nets, in: W. Brauer, W. Reisig, G. Rozenberg (Eds.), Petri Nets: Central Models and Their Properties, (Advances in Petri Nets 1986) Part I, in: Lecture Notes in Computer Science, vol. 254, Springer, 1987, pp. 248–299.
[22] J. Esparza, M. Nielsen, Decidability issues for Petri nets – a survey, Bull. Eur. Assoc. Theor. Comput. Sci. 52 (1994) 245–262.
[23] O. Owe, I.C. Yu, Deadlock detection of active objects with synchronous and asynchronous method calls, in: 27th Norsk Informatikkonferanse, NIK 2014, Høgskolen i Østfold, Fredrikstad, Norway, November 17–19, 2014, Bibsys Open Journal Systems, Norway, 2014.
[24] K.I. Pun, Behavioural Static Analysis for Deadlock Detection, Ph.D. thesis, Department of Informatics, University of Oslo, Norway, 2014.
[25] E. Giachino, C. Laneve, M. Lienhardt, A framework for deadlock detection in core ABS, Softw. Syst. Model. 15 (4) (2016) 1013–1048.
[26] A. Flores-Montoya, E. Albert, S. Genaim, May-happen-in-parallel based deadlock analysis for concurrent objects, in: D. Beyer, M. Boreale (Eds.), Proc. International Conference on Formal Techniques for Distributed Systems, FMOODS/FORTE 2013, in: Lecture Notes in Computer Science, vol. 7892, Springer, 2013, pp. 273–288.
[27] N.C. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, Designing a workflow system using coloured Petri nets, Trans. Petri Nets Other Models Concur. 3 (2009) 1–24.
[28] B. Mitchell, L.M. Kristensen, L. Zhang, Formal specification and state space analysis of an operational planning process, STTT 9 (3–4) (2007) 255–267.
[29] E. Giachino, C. Laneve, Analysis of deadlocks in object groups, in: R. Bruni, J. Dingel (Eds.), Formal Techniques for Distributed Systems, FMOODS-FORTE, in: Lecture Notes in Computer Science, vol. 6722, Springer, 2011, pp. 168–182.
[30] V. Mastandrea, Deadlock analysis with behavioral types for actors, in: Proceedings of the 17th Italian Conference on Theoretical Computer Science, Lecce, Italy, September 7–9, 2016, 2016, pp. 257–262.
[31] L. Henrio, C. Laneve, V. Mastandrea, Analysis of synchronisations in stateful active objects, in: N. Polikarpova, S. Schneider (Eds.), Proceedings of the International Conference on Integrated Formal Methods, IFM'17, in: Lecture Notes in Computer Science, vol. 10510, Springer, 2017, pp. 195–210.
[32] D. Caromel, L. Henrio, B.P. Serpette, Asynchronous sequential processes, Inf. Comput. 207 (4) (2009) 459–495.
[33] N. Busi, R. Gorrieri, A Petri net semantics for pi-calculus, in: I. Lee, S.A. Smolka (Eds.), CONCUR '95: Concurrency Theory, 6th International Conference, Philadelphia, PA, USA, August 21–24, 1995, Proceedings, in: Lecture Notes in Computer Science, vol. 962, Springer, 1995, pp. 145–159.
[34] E. Best, R.R. Devillers, M. Koutny, Petri net algebra, in: Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2001.
[35] P. Baldan, F. Bonchi, F. Gadducci, G.V. Monreale, Modular encoding of synchronous and asynchronous interactions using open Petri nets, Sci. Comput. Program. 109 (2015) 96–124.
[36] W. McLendon, R. Vidale, Analysis of an Ada system using coloured Petri nets and occurrence graphs, in: Proc. of Application and Theory of Petri Nets, in: Lecture Notes in Computer Science, vol. 616, Springer, 1992, pp. 384–388.
[37] B. Long, P.A. Strooper, L. Wildman, A method for verifying concurrent Java components based on an analysis of concurrency failures, Concurr. Comput. 19 (3) (2007) 281–294.

[38] R. Bruni, H.C. Melgratti, E. Tuosto, Translating Orc features into Petri nets and the join calculus, in: M. Bravetti, M. Núñez, G. Zavattaro (Eds.), Proc. Third International Workshop on Web Services and Formal Methods, WS-FM'06, in: Lecture Notes in Computer Science, vol. 4184, Springer, 2006, pp. 123–137.

[39] B. Mikolajczak, J. Rumbut, Distributed dynamic programming using concurrent object-orientedness with actors visualized by high-level Petri nets, Comput. Math. Appl. 37 (11–12) (1999) 23–34.

[40] M. Odersky, Functional nets, in: G. Smolka (Ed.), Programming Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 1–25.

[41] Y. Sami, G. Vidal-Naquet, Formalisation of the behavior of actors by colored Petri nets and some applications, in: E.H.L. Aarts, J. van Leeuwen, M. Rem (Eds.), PARLE '91 Parallel Architectures and Languages Europe, Springer, 1991, pp. 110–127.

[42] L. Nigro, F. Pupo, Schedulability analysis of real time actor systems using coloured Petri nets, in: G. Agha, F. de Cindio, G. Rozenberg (Eds.), Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets, in: Lecture Notes in Computer Science, vol. 2001, Springer, 2001, pp. 493–513.

[43] E.M. Clarke, R. Enders, T. Filkorn, S. Jha, Exploiting symmetry in temporal logic model checking, Form. Methods Syst. Des. 9 (1–2) (1996) 77–104.

[44] K. Jensen, L. Kristensen, T. Mailund, The sweep-line state space exploration method, Theor. Comput. Sci. 429 (2012) 169–179.

[45] T. Cogumbreiro, R. Surendran, F. Martins, V. Sarkar, V.T. Vasconcelos, M. Grossman, Deadlock avoidance in parallel programs with futures: why parallel tasks should not wait for strangers, PACMPL 1 (OOPSLA) (2017) 103:1–103:26, https://doi.org/10.1145/3143359.