Using Coloured Petri Nets for Resource Analysis of Active Objects

Anastasia Gkolfi[1], Einar Broch Johnsen[1],
Lars Michael Kristensen[2], and Ingrid Chieh Yu[1]

[1] Department of Informatics, University of Oslo, Norway
{natasa,einarj,ingridcy}@ifi.uio.no
[2] Western Norway University of Applied Sciences, Bergen, Norway
lmkr@hvl.no

Abstract. Pay-on-demand resource provisioning is an important driver for cloud computing. Virtualized resources open for resource awareness, such that applications may contain resource management strategies to modify their deployment and reduce resource consumption. The ABS language supports the modelling of deployment decisions and resource management for active objects. In this paper, the semantics of ABS is captured directly as a Coloured Petri Net (CPN) model capable of representing any ABS program by appropriate setting of the initial marking. We define an abstraction relation between the CPN model and the language semantics such that markings of the CPN model become abstract ABS configurations. We use a CPN model checker as an abstract interpreter to investigate resource distribution and starvation problems for deployed active objects in ABS.

1 Introduction

Pay-on-demand resource provisioning is an important driver for cloud computing. Using resources on the cloud, a service provider does not need to cater hardware resources upfront to deploy the service but can lease resources as required by the deployed service. Resources may be dynamically added or removed depending on the traffic to a service. The enabling virtualization technology introduces a software layer representing hardware resources, which means that deployment decisions can be programmed. Virtualized resources open for resource awareness, such that applications may contain resource management strategies to modify their deployment and reduce resource consumption. In this context, it is interesting to analyze deployment scenarios for services with respect to client traffic to, e.g., establish the amount of resources required for the timely delivery of a service.

Programming models which *decouple* control flow and communication, such as Actors [1, 2] and active objects [9,14, 23], inherently support both scalability (as argued with the Erlang programming language [5] and Scala's actors [19]) and compositional reasoning [10,15,16]. These features are also interesting for distributed services which should adapt to elastic cloud deployment. The formally

defined active object language ABS [23, 26] directly supports the modelling of deployment decisions and resource management for active objects.

In this paper, we develop a method to investigate resource distribution for deployed active objects in ABS by a translation into Coloured Petri Nets (CPNs) [22]. CPNs extend the basic Petri net model [30] with data and data manipulation. We extend previous work [17] from behavioural models to deployment models such that the formal semantics of deployment models in ABS is captured directly as a hierarchical CPN. Consequently, the number of places in the CPN model is independent of the size of a program, and different programs are captured by changing the initial marking of the CPN model. This also allows the dynamic launch of virtual resources by the firing of CPN transitions. We define an abstraction relation between the CPN model and the language semantics. The model checker of CPN Tools is used as an abstract interpreter to investigate resource distribution and starvation problems for deployed active objects in ABS.

The main contributions of this paper are: (1) a *formal model* of deployed ABS programs as a hierarchical CPN that reflects the ABS semantics with markings as abstract configurations; (2) an *abstraction relation* translating resource-aware ABS programs into CPN markings and a *proof of correctness* of the abstraction relation; and (3) management support for deployment decisions in terms of *automated resource analysis* of starvation freedom and resource redistribution.

The paper is organized as follows: Section 2 introduces the ABS language, focusing on the modelling of deployment, and Section 3 briefly introduces CPNs. Section 4 presents the CPN model of the ABS semantics and Sect. 5 the abstraction relation between the CPN model and the ABS semantics, as well as the sketch of the soundness proof. The interested reader can find the full proof in the Appendix A. Section 6 shows how the CPN Tools model checker can be used for the resource analysis of ABS programs. Finally, in Sect. 7 we sum up conclusions and discuss related work.

2 Deployment Modelling in ABS

ABS [23] is a formally defined language for the executable modelling of distributed, object-oriented systems. ABS supports *deployment modelling* by a separation of concerns between the resource costs of executions and the resource capacities of *deployment components* on which the executions take place [26]; deployment components can be understood as (virtual) locations for computation. Deployment decisions can be made inside models, by allocating objects to deployment components with given resources at creation time (e.g., [4, 24]).

ABS consists of a functional layer to express computation, an imperative layer to express communication and synchronization, and a deployment layer to express deployment decisions. In this paper we elide the functional layer to focus on control flow and deployment; the relevant syntax is shown in Fig. 1. A program consists of class definitions which again contain field declarations and method definitions, and a main block. We follow the syntactic conventions of Java and only explain syntax that differs from Java.

Syntactic categories.	*Definitions.*
s in STMT	$P ::= \overline{CL}\ \{\overline{T}\ \overline{x};\ ws\}$
e in EXPR	$CL ::= \textsf{class}\, C\, (\overline{T}\ \overline{x})\, \{\overline{T}\ \overline{x};\ \overline{M}\}$
g in GUARD	$M ::= T\ m\ (\overline{T}\ \overline{x})\ \{\overline{T}\ \overline{x};\ ws\}$
	$s ::= \textsf{skip} \mid x = rhs \mid [\mathsf{DC}{:}e]\ x = \textsf{new}\ C(\overline{e}) \mid \textsf{suspend} \mid \textsf{await}\ g$
	$\mid \textsf{if}\ e\ \{ws\}\ \textsf{else}\ \{ws\} \mid \textsf{while}\ e\ \{ws\} \mid \textsf{return}\ e$
	$ws ::= s \mid [\mathsf{Cost}{:}\ e]\ s \mid ws; ws$
	$rhs ::= e \mid e!m(\overline{e}) \mid x.\textsf{get}$
	$g ::= x? \mid \textsf{duration}(e, e) \mid g \wedge g$

Fig. 1. ABS syntax. Overbar notation is by convention used to denote lists.

The imperative layer of ABS is used for communication and synchronization between concurrent objects. Objects are instantiated from classes by the statement $[\mathsf{DC}{:}\ server]\ o = \textsf{new}\ C(\overline{e})$, where the optional annotation $\mathsf{DC}{:}\ server$ expresses the deployment component on which the object should be created and $\overline{e}$ are constructor arguments. A reserved field **thisDC** points to the object's deployment component, just like **this** points to the object's identifier. Concurrent objects execute processes which stem from asynchronous method calls and terminate upon method completion. Asynchronous method calls $f = o!m(\overline{e})$ are non-blocking and return a future, i.e., a placeholder for the method reply (see, e.g., [9]). The blocking expression $f.\textsf{get}$ retrieves the return value from a future f.

Objects combine reactive and active behaviour (i.e., a run method is automatically activated upon object creation) by means of *cooperative scheduling*: Processes in an object may suspend at explicit scheduling points, allowing the scheduler to transfer control to another enabled process. Between the scheduling points, only one process is active in each object, so race conditions are avoided. Unconditional scheduling points are expressed by the statement **suspend**, conditional scheduling points by **await** g, where g may be a *synchronization condition* on a future, written $f?$ (where f points to a future) or a *duration guard*, written $\textsf{duration}(b, w)$ where b and w are bounds on the time interval before the condition becomes true. ABS supports the modelling of dense time [8]; the local passage of time is expressed in terms of durations (as in, e.g., UPPAAL [28]).

Deployment models capture physical or virtual infrastructure in ABS using dynamically created *deployment components* [25, 26] to represent computing environments. A deployment component is a modelling abstraction which captures locations offering (restricted) resources to computations. Deployment components are created as instances of a special class *DC* which takes as parameter a number expressing the resource capacity of the deployment component per time interval. These components implement a method $\textsf{transfer}(dc, e)$ which enables *vertical scaling* by shifting up to e resources to a target deployment component dc. This is in contrast to the horizontal scaling which is realized by the dynamic allocation of deployment components. ABS also supports cost annotations to model resource consumption. Thus, *weighted statements* ws are statements $[\mathsf{Cost}{:}\ e]\ s$ which express that e resources are required to complete execution of the statement s. In this paper we model so-called elastic computing resources, where the computation

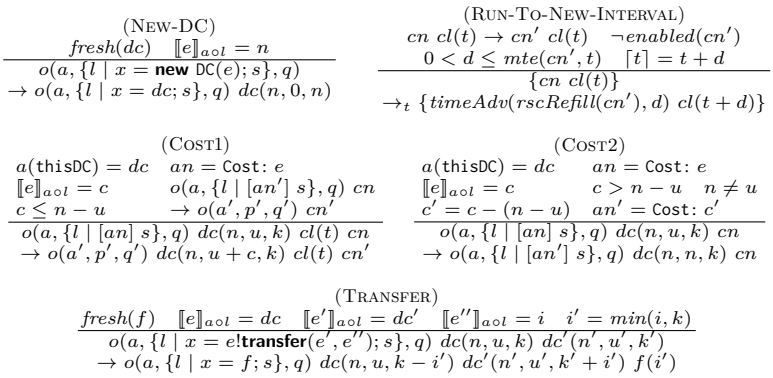


Fig. 2. Semantics of the deployment layer of ABS (based on [26]).

speed of virtual machines is determined by the amount of elastic computing resources allocated to these machines per time interval. The computation time of processes depends on the available resources of their deployment component and on how many other processes are competing for these resources.

Semantics. The semantics of ABS is given by a (transitive) transition relation $\rightarrow$ over configurations. We here focus on transition rules formalizing the cost and deployment aspects of the execution of ABS programs. Configurations include *objects* $o(a, p, q)$, where o is the identifier, a the state, p the active process, and q the queue of suspended processes; *futures* $f(v)$ with identifier f and return value v; and *deployment components* $dc(n, u, k)$ with identifier dc, n resources available in the current time interval, u resources already used in the current time interval, and k resources available in the next time interval. Technically, the deployment components book-keep the resource consumption of its allocated objects per time interval. Thus, in New-DC, a new deployment component with a fresh identifier dc is created, with n resources available in both the current and the next time interval. Rule Run-To-New-Interval captures the advance of time. Here, the brackets enclose all objects in the configuration such that time advances uniformly. The predicate $enabled(cn)$ expresses that a reduction is possible in cn without advancing time, and cn' represents the next state in which time must advance in a maximal progress semantics. Let $mte(cn', t)$ denote the maximal time advance until $enabled(cn')$. The condition $\lceil t \rceil = t + d$ expresses that time advance has arrived at the next resource provisioning (a corresponding rule without this condition advances time without resource provisioning). Two auxiliary functions recursively change the state cn': *timeAdv* decrements counters for **duration**-expressions and *rscRefill* provisions resources in the deployment components by changing each $dc(n, u, k)$ to $dc(k, 0, k)$.

Rule Cost1 removes the cost annotation of a statement if the associated deployment component has sufficient resources to execute the statement in the current time interval. Rule Cost2 reduces the remaining cost if the deployment component can provision some but not all of the required resources. Rule Trans-

FER shifts e'' resources from a deployment component e to another deployment component e', up to the amount of resources that e has allocated for the next time interval. This change only affects e' for the next time interval. For further details on the semantics of deployment components in ABS, we refer to [26].

3 Coloured Petri Nets

Petri nets capture true concurrency in terms of causality and synchronization [30]. A basic (low-level) Petri net constitute a directed bipartite graph comprised of places and transitions connected by arcs. 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. Places are used to model the states of the system and may hold tokens. A marking consists of a distribution of tokens on the places of the Petri net and represent a state of the modelled system. Transitions are used for modelling the actions of the system. A transition is enabled in a marking when there is a token on each of its input places. An enabled transition may occur (fire) and the effect of occurrence is to consume a token from each input place of the transition and add a token to each of its output places. This in turn changes the current marking of the Petri net model.

Coloured Petri Nets (CPNs) is a well-established form of high-level Petri nets [22]. High-level Petri nets extend the basic Petri net formalism to enable the modelling of data and data manipulation. Each place in a CPN has an associated type determining the data values that tokens residing on the place may have, i.e., the tokens in a place represent individual values of that type. The types, representing sets of values, are called *colour sets* and individual values are seen as colours. A type can be arbitrarily complex, defined by many sorted algebra in the same way as abstract data types. A place may in general hold a multi-set of tokens values over the type of the place. Arcs have associated arc expressions and transitions may have an associated boolean guard expression. These expressions may contain free variables which needs to be bound to values in order to determine whether a transition is enabled, i.e., whether the required tokens are present on input places and whether the guard evaluates to true. Similarly, the multi-set of tokens removed from input places and added to output places when an enabled transition occurs are determined by evaluating the arc expression of the transition according to the values bound to the free variables.

Below we formally define CPNs [22] in their basic form without hierarchical modules. Hierarchies enrich CPNs with modularity in order to support the practical modelling of large systems. The basic definition of CPNs suffices for our purposes as any hierarchical CPN can be unfolded to a semantically equivalent non-hierarchical CPN.

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

1. P *is a finite set of* places P *and* T *is a finite set of* transitions T *such that* $P \cap T = \emptyset$*;*

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 non-empty* types *(colour sets);*
4. *V is a set finite set of* typed variables *V such that type$(v) \in \Sigma$ for all $v \in V$;*
5. *$C : P \to \Sigma$ is a* colouring function *associating a type to each place;*
6. *$G : T \to Expr_V$ and $E : A \to Expr_V$ are* labelling functions *associating expressions with free variables from V to transitions and arcs, respectively such that type$(G(t))$ = bool for all $t \in T$ and type$(E(a)) = C(p)_{MS}$ where p is the place connected to a and $C(p)_{MS}$ denotes the multi-set type over $C(p)$.*
7. *$I : P \to Expr_\emptyset$ is an* initialization function *associating a closed expression to each place specifying the initial marking of the place such that type$(I(p)) = C(p)_{MS}$ for all $p \in P$*

4 A CPN Model of ABS Semantics

In [17] the authors presented a CPN, modelling the concurrency of ABS. Active objects in [17] was represented as tokens whose colour contains their identifier and process pool. The process pool was implemented as a list, the head of which was the active process and the tail the list of the processes that were candidates to be activated by the scheduler. This list was being updated according to the calling methods of the other objects following the communication mechanism of ABS. In this paper, we focus on the deployment part of ABS. We present a new CPN, modelling the deployment fragment of the language. We model the life time of program execution in a cyclic way, where the resources are refilled at the completion point of each cycle. This is illustrated in Fig. 3, where in the bottom part we see the resource refill before the process execution in the next cycle.

We take as an input tokens that can be produced from the imperative part [17] of ABS as described above and we add information concerning the cost of each process and the deployment component they are located. This information, together with the deployment semantics of ABS can be used to verify starvation freedom of active objects and explore resource management strategies.

In the rest of the section, we present the CPN model which follows the corresponding semantic rules of ABS that add resource awareness through a running example. It is inspired from the change of the calling behaviours of cellphone clients during new year's eve midnight. Based on this example, we show in Sect. 6 how resource reallocation between deployment components can be used for load balancing purposes.

4.1 Telephone and SMS Services at Midnight on New Year's Eve

We use a running example inspired from cellphone clients behaviour in order to illustrate the relation between the CPN model and ABS programs and to show how we can use the model checker of CPN Tools for load balancing scenarios.

The average demand on phone calls and SMS messages from cellphone clients during the year is relatively low and the available resources suffice in a current

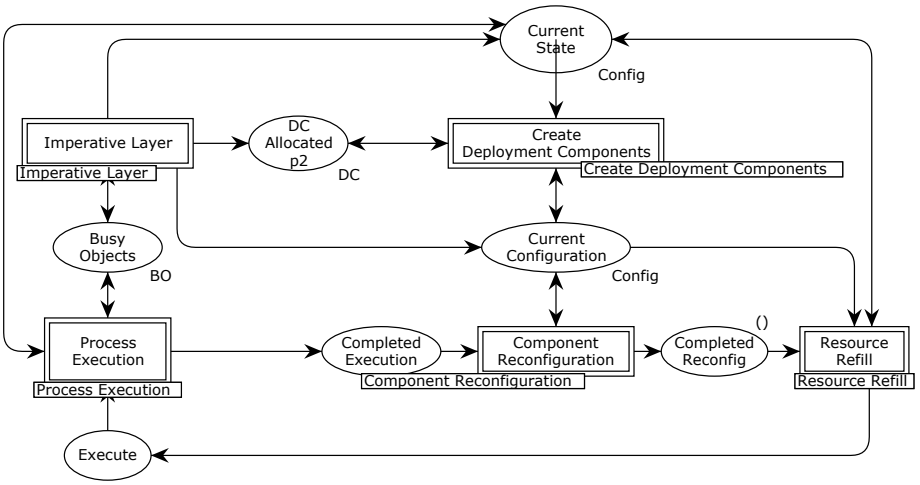


Fig. 3. Top-level module of the CPN deployment model

distribution. There are some particular moments of the year like, for example, around the midnight of new year's eve, where this behaviour changes and a large number of SMS is requested by the clients while the call requests are negligible. Then, the initial distribution is not adequate, since there is a lack of resources for the SMS and an overplus for the calls.

In Fig. 4, we provide the ABS implementation of the above scenario [26] where telephone and SMS servers have been realised with the two corresponding classes and the operational cost annotated in square brackets at the beginning of the statements. We see that each SMS has cost 1 and each call has cost proportional to its duration. Cellphone clients can be implemented with corresponding classes allowing objects to make method calls to the SMS and telephone services.

As mentioned above, we use CPNs to model the deployment part of ABS. The markings shown in the current section are related to our running example. It is important to note that our CPN model is parametric and different ABS programs can be analysed by setting the initial marking accordingly. In our example, we modelled the SMS and the telephone servers in CPNs as two different tokens representing the corresponding objects of Fig. 4 (TelephoneServer and SMSServer). Those tokens have as colour triples of the form (ob, dc, lst), where ob is the object identifier and dc is the deployment component of the object execution. The last component (lst) models the client behaviour. In particular, it represents the process pool of the server object that keeps all the processes created from the clients calls to the corresponding service. Each process comes along with the cost of its execution, so lst is a list of triples $(proc, cost, bool)$, where $bool$ is a flag indicating whether the process has completed its execution.

Figure 5 shows the CPN module representing the imperative layer of ABS. Initially, the model has one token in place Ready and the the transition Imperative Layer

```
1  class TelephoneServer{
2    Unit call (Int calltime) {
3      while (calltime > 0) { [Cost:1] calltime = calltime − 1; await duration (1,1); }
4    }
5  }
6  class SMSServer {
7    Unit sendSMS () { [Cost:1] skip; }
8  }
9  {// Main block
10   DC telcomp = new DC(1);
11   DC smscomp = new DC(2);
12   [DC: smscomp] SMSServer sms = new SMSServer();
13   [DC: telcomp] TelephoneServer tel = new TelephoneServer();
14   // Start client handsets...
15 }
```

Fig. 4. Implementation of Telephone and SMS Service.

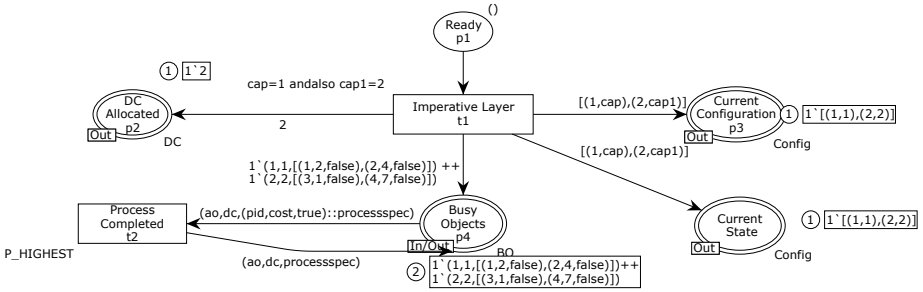


Fig. 5. CPN module of the imperative layer

is enabled. Recall that the colour of the object tokens have the form (ob, dc, lst) as explained above. In Fig. 5, we have two tokens produced in place Busy Objects. The first one represents the object TelephoneServer with identifier 1 located in the first deployment component having in its process pool two processes: one with identifier 1 and cost 2 and one with identifier 2 and cost 4. The boolean flags set to *false* indicate that the processes have not been executed yet (it can be changed to true after firing Process Completed). Similarly, the second token represents the SMSServer object. Place DC Allocated is a counter of the deployment components created so far (Fig. 9 of the Appendix shows the details of the implementation supporting dynamic deployment component creation). Places Current State and Current Configuration have as a colour set a list of pairs (dc, cap) referring to the capacities of each deployment component. Place Current State keeps the current resource distribution while place Current Configuration records the distribution that will take place in the next cycle (resp. next time interval in ABS).

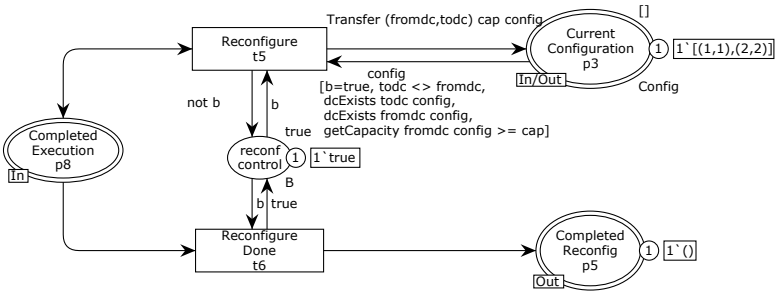


Fig. 6. CPN module for component reconfiguration

Figure 6 shows that when transition Reconfigure fires, the marking of the place Component Reconfiguration is updated according to the function Transfer of its incoming arc inscription:

```
fun Transfer (fromdc,todc) cap config = List.map (fn (dc,ccap) =>
if (dc = fromdc) then (dc,ccap - cap) else (if (dc = todc) then (dc,
    ccap+cap) else (dc,ccap))) config
```

This function transfer resources from one deployment component to another. When transition Reconfigure Done fires, then the reconfiguration has been completed. Then the resources can be refilled (details of the related implementation can be found in Fig.13 in the Appendix) and the marking of the place Current State can be updated according to the function Transfer and proceed to the execution.

Figure 7 shows the module related to the process execution and the resource consumption. Places Busy Objects and Current State are fusion places (i.e they appear in more than one module and share the same marking). Recall the meaning of their markings from Fig. 5. Object 2 needs for the execution of its first process in the list (having identifier 3) 1 resource and the availability of the second deployment component according to the marking of the place Current State is 2 resources (having colour $(2, 2)$). As a result, transition Fully Executable (Figure 7) can fire and set its cost to zero and the boolean flag to *true* (recall that the boolean flag is related to whether the process ha been fully executed or not). After this, transition Process Completed of Fig. 5 is enabled and the corresponding element of the list (head) is removed. Consider again Fig. 7: object 1 needs 2 resources to fully execute its first process while there is only 1 available, according to the marking of the place Current State. Hence it can only partially execute process 1 by consuming all available resources (here 1) when transition Partially Executable enabled. Then the token of the object 1 will be moved to the place Starving Objects with the remaining cost updated to 1, until the marking of place Current State will show resource availability at the deployment component greater or equal to 1. This can be done at a next cycle in the model, after possible resource transfer and refill. In such a case, transition Execute Starving will be enabled and send the

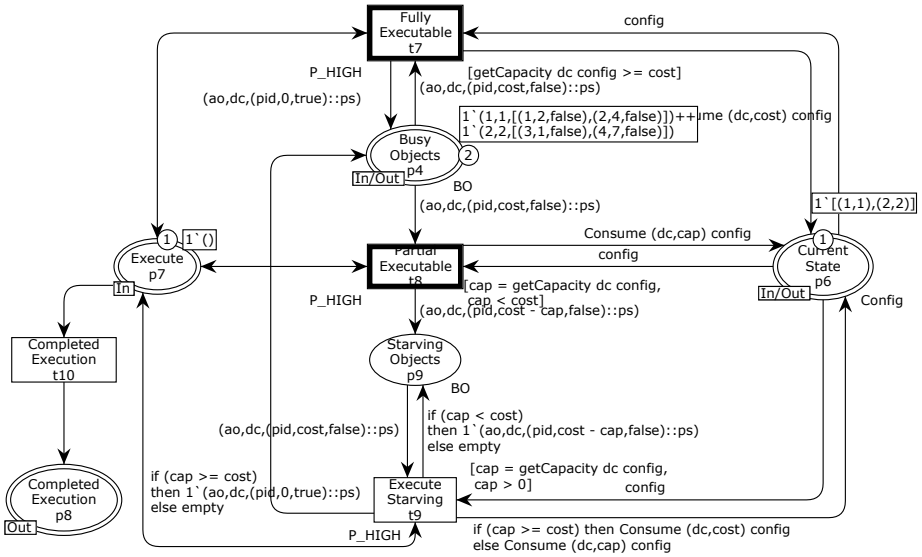


Fig. 7. CPN module for process execution

token back to the Busy Objects place; otherwise, in case of insufficient resource for completion, it will be placed again to the place Starving Objects.

5 Abstraction Relation and Soundness

The model presented in Sect. 4 translates faithfully the fragment of ABS which is responsible for the resource awareness of the language. An abstraction function matching program configurations with CPN markings followed by a soundness proof guarantee the faithfulness of this translation. In this section, we define the abstraction relation α that associates each ABS program configuration with a marking of the CPN model and each small step semantics of ABS with a finite sequence of enabled CPN transitions. This relation is an *abstract simulation relation*, where the abstraction stands for the extraction of the elements related to the resources from each ABS configuration.

Before introducing the abstraction function, we note that in ABS, configurations are multi-sets containing the objects and the deployment components. Recall also that each object contains a unique object identifier, information about the deployment component to which it is located, an active process that is currently under execution, and a pool of processes that are the candidates to be executed. Below we introduce the abstraction function α and represent all the above information at the level of tokens. Hence, we need first to define the corresponding sets and then to proceed to the definition of the abstraction function. We represent the set of ABS program configurations with $\mathcal{C}$, the set of

the active objects of the program as Obj, the set of the deployment components as Dc, and the set of the processes as $Proc$.

We can now define the functions ob and dc mapping configurations to objects and deployment components, respectively, as $X : \mathcal{C} \to Y$ where $X \in \{ob, dc\}$ and $Y \in \{Obj, Dc\}$, with the obvious matching. Similarly, we define the functions pr and pp mapping objects to processes and sets of processes (process pools) as $F : Obj \to Z$, where $F \in \{pr, pp\}$ and $Z \in \{Proc, \mathcal{P}(Proc)\}$. We also define $pdc : \mathcal{C} \to \mathcal{P}(Dc)$. Since each element among the objects, the deployment components and the processes has a unique identifier, we use natural numbers to translate it. This leads to the definition of the following injections: $L : W \to \mathbb{N}^*$ where $L \in \{obj\_id, dc\_id, proc\_id\}$ and $W \in \{Obj, Dc, Proc\}$ (with the obvious matching). We also define a cost function, assigning to each process the cost of its execution: $cost : Proc \to \mathbb{N}$ as well as two capacity functions cap and $ncap$ related to the resource capacity of each deployment component (cap for the current and $ncap$ for the next time interval), defined as $cap : Proc \to \mathbb{N}$ (resp. for $ncap$). Also, we define the function $cc : \mathcal{P}(Dc) \to \mathcal{P}(Dc \times \mathbb{N})$, where $cc(S) \stackrel{def}{=} \{(dc\_id(s), cap(s)) \in \mathbb{N}^* \times \mathbb{N} \mid s \in S\}$. Finally, we define the function $cpq : \mathcal{P}(Proc) \to \mathcal{P}(\mathbb{N}^* \times \mathbb{N} \times \mathbb{B})$, mapping the process pools (i.e. sets of processes) to sets of tuples, each representing the process identifier, the corresponding cost and a boolean flag (where $\mathbb{B}$ is the set of booleans). In particular, $cpq(S) \stackrel{def}{=} \{(proc\_id(s), cost(s), b) \in \mathbb{N}^* \times \mathbb{N} \times \mathbb{B} \mid s \in S \wedge b \in \mathbb{B}\}$. Similarly to the function cc, we define the function ncc where, instead of the function $cap(s)$ of cc we use the function $ncap(s)$.

So far we have defined elementary functions to represent the interesting information taken from ABS configurations in order to form the appropriate colour sets that are used in the model. Recall from Sect. 4 that the colour set of the tokens representing the objects of an ABS program is (ob, dc, lst) where ob is the object identifier, dc is the deployment component where the object is located, and lst is the process pool augmented with information about the cost. More concretely, lst is a list of triples $(proc, cost, bool)$ indicating the identifier of each process, its corresponding cost and a boolean flag showing whether the execution of a process has been completed or not. Recall also that the colour set of the deployment components is a pair (dc, cap) where dc is the deployment component identifier and cap its resource capacity. The colour set of the place Current State represents the current resource distribution and place Current Configuration the distribution that will take place at the next time interval.

Now, let us define the abstraction function, which induces an abstract simulation relation between ABS program configurations and CPN tokens. For all configurations $c \in \mathcal{C}$:

$$\begin{aligned} \alpha(c) = \bigcap \{M \mid \exists\, p, p', p'' \in P \text{ s.t. } p \neq p' \neq p'' \text{ and for all } ob(c) \in Obj, \\ ((obj\_id \circ ob)(c), (dc\_id \circ dc)(c), (cpq \circ pr \circ ob)(c)) \in M(p) \\ \wedge\ (cc \circ pdc)(c) \in M(p') \\ \wedge\ (ncc \circ pdc)(c) \in M(p'')\} \end{aligned} \tag{1}$$

where $\bigcap$ is the intersection over sets of multi sets. Recall the colour sets of the places Busy Objects, Starving Objects, Current Configuration and Current State. The above function extracts information from ABS configurations and matches that information to the colour of the tokens of those places appropriately. At the second line of the equation, we have the information related to the tokens representing objects. They can be located either in Busy Objects or in Starving Objects place (hence the existential quantifier of the equation). The third and the fourth lines of the equation concern the deployment components and, in particular, the resource distributions for either the current or the next time interval. This information is also available at each ABS configuration, and hence the abstraction function retrieves it and matches it with the tokens located in places Current State and Current Configuration respectively.

Now, we can proceed to the correctness of the behaviour of our translation by establishing an abstract weak simulation relation between program configurations and CPN markings. For the full proof, we direct the reader to Appendix A.

Theorem 1. *The markings of the CPN model are in abstract (weak) simulation relation with ABS program configurations.*

Proof. (sketch) We need to prove that, for any program configuration c, if $c \twoheadrightarrow_r c'$ for some ABS semantic rule r, then there exists a marking in the net M' and a *sequence* of enabled transitions w, such that $\alpha(c) \xrightarrow{w} M'$ and $\alpha(c') \subseteq M'$ (where, with $\subseteq$ we denote the subset relation between sets of multi sets) □

6 Resource Analysis and Management

We now show how state space exploration of the CPN model can be used to reason about starvation freedom of a ABS program. In presence of starvation, we show the state space of the CPN model can be used to synthesise a sequence of resource reconfigurations which can eliminate starvation. Finally, we show how the sequence of resource reconfiguration can be used to automatically obtain an implementation of a starvation free load balancer.

For the resource analysis, we rely on the model checker of CPN Tools. We use the running example from the previous sections for illustration purposes, but our analysis approach generalises to instantiations of the CPN model. The state space for the running example has 776 nodes (states) and 1069 arcs (occurring events) and could be generated in less than 1 second. For our analysis, we also rely on the strongly connected components (SCCs) of the state space. The SCCs could be generated in less than 1 second and contains 719 nodes (SCCs) and 988 arcs connecting the SCCs.

6.1 Resource Analysis

Section 4 covered the deployment layer as a CPN model. We obtained the execution cost of a program by adding cost tags to the tokens representing the active objects. More concretely, we matched each process of the process pool with

the corresponding cost. Recall that the colour of an active object is represented as a triple (ob, dc, lst), where ob is the object identifier, dc is the deployment component where the object is being executed, and lst is the process pool of the objects. The latter is represented as a list of triples $(proc, cost, bool)$ where $proc$ is the process identifier, $cost$ is the related execution cost to the current process, and $bool$ is boolean flag indicating whether the process has fully executed (value *true*) or not (value *false*). The head of lst represents the active process.

As can be seen in Fig. 7, the model has been constructed with place Current State which record the resource availability of each deployment component by hosting the corresponding tokens of colour (dc, cap), where dc is the deployment component identifier and cap its resource capacity. In addition, the place Starving Objects holds tokens representing the objects whose execution has been blocked because of lack of resources at the current time interval. In the following, we explain in detail how to perform resource analysis using the markings of those places.

The first important information related to resource management is whether the current resource distribution provides sufficient resources for the full execution of the processes the objects have in their process pools. In other words, we need to check for *starvation freedom*. By model construction, place Starving Objects keeps track of the starving objects, if any. For the analysis, we implement the following Standard ML queries in CPN Tools for checking starvation freedom:

```
fun findStarvingObjects n =
 let
   val mSO  = Mark.Process_Execution'Starving_Objects_p9 1 n
   val soid = List.map (fn (ao,_,_) => ao) mSO
 in soid end

fun anyStarving n = (findStarvingObjects n) <> nil
fun anySO ()      = PredAllNodes anyStarving
```

Function `findStarvingObjects` takes a state (marking) n as argument and extract the list of object identifiers from any tokens on place Starving Objects. Such object identifiers represent objects that are starving in state n. This function is then used in the predicate `anyStarving` which can be used to determine whether or not there are any starving objects in state n. The `anyStarving` predicate is then used as a higher-order argument to the built-in query function `PredAllNodes` which returns the list of all those states where the `anyStarving` predicate holds, i.e., all states where there are some object starving.

For our running example, CPN Tools returns a non-empty list containing several states which imply that starvation is present. Since the current resource distribution leads to starvation, an interesting question is *whether there exists a resource reallocation strategy leading to starvation freedom*. For that, we need to check for the existence of any terminal SCC containing states where there are no starving objects and then ask for a path leading to that state. This involves writing queries of similar complexity as was shown above and consists of:

– a function that checks whether a SCC is non-starving, i.e. whether it consist of only states where the marking of place Starving Objects is empty.

- a function searching for a non-starving SCC among the terminal SCCs ones.
- a function that returns the path from the initial marking to a state in one on the non-starving terminal SCCs.

In the last case, we are interested only in the information related to resource transfer. Recall that the module Component Reconfiguration (see Fig. 6) is related to the resource refill, and the Reconfigure transition related to the resource transfer. We therefore filtered the path returned from CPN Tools to show only the occurrences and binding elements of this transition, where the binding specifies the values bound to the variables of the transition. This synthesised sequence of the resource transfers we need to perform in order to avoid starvation. For our running example, this resulted in the following sequence of resource transfers:

[Component Reconfiguration‘Reconfigure
$(1, \{b = true, cap = 1, \text{config} = [(1,1),(2,2)], \text{fromdc} = 1, \text{todc} = 2\})$
$(1, \{b = true, cap = 3, \text{config} = [(1,0),(2,3)], \text{fromdc} = 2, \text{todc} = 1\})$
$(1, \{b = true, cap = 2, \text{config} = [(1,3),(2,0)], \text{fromdc} = 1, \text{todc} = 2\})]$

The first line shows the module and the transition of which we get the binding elements. In the tuples that follow, "b" is a guard of the transition, "cap" is the amount of the resources we need to move, "config" is the current resource distribution, "fromdc" is the source deployment component and "todc" is the target deployment component. The resource transfer represented by the sequence hence provides a non-starvation strategy.

6.2 Resource Management: Load Balancing

Above, we saw how the state space analysis of the CPN model can be used to prove starvation freedom or, in case of starvation to synthesise a path from the initial resource distribution to a starvation free state of a terminal SCC. In the rest of this section, we will see how this path can be used in load balancing.

Recall that in ABS, the discrete time follows maximal progress semantics: the time advances when no further execution can happen. In that case, the resources are refilled according to the transfer command, if any; otherwise they are updated as in the previous time interval. Recall also that the colour of the deployment components is (dc, cap) where the first element is the deployment component identifier and the second one its capacity. As an example, the pair $(1, 2)$ means that the deployment component 1 has a capacity of 2 resources.

Let us consider again our running example. Below follows a more detailed version of the path discussed in Sect. 6.1. For the sake of simplicity, we present only the name of the transition followed by the corresponding binding. The enumeration in the left corresponds to the respective ABS time point:

[Imperative Layer(1, {$cap = 1, cap1 = 2$})
t =0 Resource Refill(1, config = $[(1,1),(2,2)]$, oldconfig = $[(1,1),(2,2)]$})
...
Reconfigure(1, {$b = true$, $cap = 1$, config = $[(1,1),(2,2)]$, fromdc = 1, todc = 2})
Reconfigure Done(1, {$b = false$})
t=1 Resource Refill(1, config = $[(1,0),(2,3)]$, oldconfig = $[(1,0),(2,0)]$})
...
Reconfigure Done(1, {$b = true$})
t=2 Resource Refill(1, config = $[(1,0),(2,3)]$, oldconfig = $[(1,0),(2,0)]$})
...
Reconfigure(1, {$b = true$, $cap = 3$, config = $[(1,0),(2,3)]$, fromdc = 2, todc = 1})
Reconfigure Done(1, {$b = false$})
t=3 Resource Refill(1, config = $[(1,3),(2,0)]$, oldconfig = $[(1,0),(2,0)]$})
...
Reconfigure Done(1, {$b = true$})
t= 4 Resource Refill(1, config = $[(1,3),(2,0)]$, oldconfig = $[(1,0),(2,0)]$})
...
Reconfigure(1, {$b = true$, $cap = 2$, config = $[(1,3),(2,0)]$, fromdc = 1, todc = 2})]
Reconfigure Done(1, {$b = false$})
t=5 Resource Refill(1, config = $[(1,1),(2,2)]$, oldconfig = $[(1,1),(2,0))]$

In the above path, the highlighted lines are resource transfers that will lead to a starvation free state, as we saw in Sect. 6.1. In our example, we consider two objects located in two deployment components. The first line shows the resource initialisation. The variables *cap* and *cap*1 refer, respectively, to the capacities of the first and the second deployment component. Hence we obtain the initial distribution: $(1,1),(2,2)$. During the first time interval, the highlighted line shows that we need to transfer 1 resource (variable *cap*) from the first deployment component (variable fromdc) to the second one (variable todc). Here, we notice that the variables are local to each transition, hence a possible name reuse (e.g. *cap*) should not create confusion. As a result of the first transfer, we obtain the distribution $(1,0),(2,3)$, as we can see at the corresponding resource refill (variable config) of the beginning of the second time interval (when $t = 1$). During the second time interval, we do not need to transfer resources, hence the refill of the beginning of the third time interval (when $t = 2$) updates the resources according to the last distribution, i.e. $(1,0),(2,3)$. Similarly, we obtain the distributions $(1,3),(2,0)$ when $t = 3$, $(1,3),(2,0)$ when $t = 4$ (no transfer) and $(1,1),(2,2)$ when $t = 5$.

The variable oldconfig of the transition Resource Refill shows the available resources that we have before time advances. Because of the maximal progress semantics of ABS, the second component of each pair should be zero in all the time intervals except the extremal ones: the first is the initialisation and the last one shows that we have remaining 1 resource at the first deployment component after the full execution of the processes of the first object. This is possible since the last state is starvation free.

From the above path information we can implement very easily a load balancer like the one of Fig. 8. We match object "1" with the telephone service and object "2" with the SMS service and we assume they are located at the deployment

```
1  class Balancer(DC telcomp, DC smscomp) {
2    Unit run() {
3      telcomp!transfer(smscomp,1);
4      await duration(1,1);
5      smscomp!transfer(telcomp, 3);
6      await  duration(1,1);
7      telcomp!transfer(smscomp,2);
8  }}
9  {// Main block
10   ... // deployment components, etc. as before
11   new Balancer(telcomp,smscomp);
12 }
```

Fig. 8. Implementation of Load Balancer.

components "telcomp" and "smscomp", respectively, having the capacities as in the model, i.e. 1 and 2. In our load balancer we applied the strategy given by the path explained above, so we transfer 1 resource from the deployment component "telcomp" to the "smscomp" during the first time interval, 3 resources from the deployment component "smscomp" to the "telcomp" during the third time interval, and 2 resources from the deployment component "telcomp" to the "smscomp" during the fourth time interval. Notice here that each time we transfer resources, they take place at the next time interval according to the semantics of ABS.

7 Conclusions and Related Work

We have presented a CPN model of the deployment fragment of ABS [26], a resource aware programming language suitable for cloud applications. A key characteristics of our approach is that the compact modelling supported by CPNs allowed us to develop a CPN model capable of simulating any ABS program by only changing the initial marking. The main benefit of our approach is the ability to use model checking techniques to identify starvation of resource aware active objects, and to synthesise reconfiguration sequences that eliminates starvation and which in turn can be used to automatically obtain load-balancer implementations.

Some of the earliest applications of CPNs for analysis of distributed objects appeared in [27] focussing on spatial distribution of objects and not resource consumption. Early work [32] also considered simulation-based capacity planning of web-servers, but not in a context with dynamically configurable resources. More recent work [12] has considered the COSTA language [3] for deployment and management of cloud applications. Their work, however, focused on the deployment language and management operations. COSTA is able to approximate the computational cost of a program, but do not provide resource management. Recent work [20] has also explored evaluation of cloud deployment strategies for distributed NoSQL databases using CPN simulation, but without dynamic reconfiguration. In contrast to previous modelling of programming languages

into Petri nets like Ada, Java, Orc ([21] [29] [13]) where the model depends on the program, we suggest a fixed sized model where the markings are program configuration abstractions, hence different programs can be analysed by one single model upon different initialisation (according to the abstraction function).

More broadly, process algebras [7], priced [11] and probabilistic [6] automata have been proposed for performance analysis of embedded systems with resource constraints. Also, other resource analysis on resource aware programs like [31] and [18], target to guarantee that the program cost does not exceed a resource threshold. Our work is not restricted only to the guarantee of resource sufficiency, but also in case of starvation, proposes strategies for vertical scaling that can be retrieved by the counter examples of CPN Tools.

Our present work extends [17] by taking as input the communication status of resource aware active objects and performing resource analysis. We demonstrated how to statically construct a load balancer. A direction for future work will be to extend the model to support dynamic load balancing and investigate optimal vertical scaling using the CPN model checker. Another direction will be to perform a comprehensive experimental evaluation on a larger set of ABS programs.

References

1. Agha, G., Hewitt, C.: Concurrent programming using actors. In: Object-Oriented Concurrent Programming, pp. 37–53. The MIT Press (1987)
2. Agha, G.: ACTORS: A Model of Concurrent Computations in Distributed Systems. The MIT Press, Cambridge, Mass. (1986)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theor. Comput. Sci. **413**(1), 142–159 (2012)
4. Albert, E., de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L., Wong, P.Y.H.: Formal modeling and analysis of resource management for cloud architectures: An industrial case study using Real-Time ABS. Journal of Service-Oriented Computing and Applications **8**(4), 323–339 (2014)
5. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
6. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.: Performance evaluation and model checking join forces. Commun. ACM **53**(9), 76–85 (2010)
7. Barbanera, F., Bugliesi, M., Dezani-Ciancaglini, M., Sassone, V.: Space-aware ambients and processes. Theor. Comput. Sci. **373**(1-2), 41–69 (2007)
8. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: User-defined schedulers for real-time concurrent objects. Innovations in Systems and Software Engineering **9**(1), 29–43 (2013)
9. de Boer, F., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (Oct 2017)
10. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: Proc. of ESOP'07. LNCS, vol. 4421, pp. 316–330. Springer (Mar 2007)
11. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N.: Quantitative analysis of real-time systems using priced timed automata. Comm. ACM **54**(9), 78–87 (2011)
12. Brogi, A., Canciani, A., Soldani, J., Wang, P.: Petri net-based approach to model and analyze the management of cloud applications. ToPNoC **XI**, 28–48 (2016)

13. Bruni, R., Melgratti, H.C., Tuosto, E.: Translating Orc features into Petri nets and the join calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) Proc. of WS-FM'06. LNCS, vol. 4184, pp. 123–137. Springer (2006)
14. Caromel, D., Henrio, L.: A Theory of Distributed Objects. Springer (2005)
15. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) Proc. of CADE'15. LNCS, vol. 9195, pp. 517–526. Springer (2015)
16. Din, C.C., Owe, O.: Compositional reasoning about active objects with shared futures. Formal Asp. Comput. **27**(3), 551–572 (2015)
17. Gkolfi, A., Din, C.C., Johnsen, E.B., Steffen, M., Yu, I.C.: Translating active objects into Colored Petri Nets for communication analysis. In: Proc. of FSEN'17. LNCS, vol. 10522, pp. 84–99. Springer (2017)
18. Gordon, A.D. (ed.): Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, LNCS, vol. 6012. Springer (2010)
19. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. **410**(2–3), 202–220 (2009)
20. Huang, X., Wang, J., Qiao, J., Zheng, L., Zhang, J., R.K, R.W.: Performance and replica consistency simulation for quorum-based nosql system cassandra. In: Proc. of PETRI NETS'17. vol. 10258, pp. 78–98. Springer (2017)
21. Ichbiah, J., Barnes, J.G.P., Heliard, J.C., Krieg-Brückner, B., Roubine, O., Wichmann, B.A.: Modules and visibility in the Ada programming language. In: On the Construction of Programs, pp. 153–192 (1980)
22. Jensen, K., Kristensen, L.M.: Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer (2009)
23. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Proc. of FMCO'10. LNCS, vol. 6957, pp. 142–164. Springer (2011)
24. Johnsen, E.B., Owe, O., Schlatte, R., Tapia Tarifa, S.L.: Dynamic resource reallocation between deployment components. In: Proc. of ICFEM'10. LNCS, vol. 6447, pp. 646–661. Springer (Nov 2010)
25. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In: Aoki, T., Tagushi, K. (eds.) Proc. of ICFEM'12. LNCS, vol. 7635 (Nov 2012)
26. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. Journal of Logical and Algebraic Methods in Programming **84**(1), 67—91 (2015)
27. Jørgensen, J., Mortensen, K.: Modelling and analysis of distributed program execution in beta using coloured petri nets. In: Proc. of ICATPN'96. lncs, vol. 1091, pp. 249–268. Springer (1996)
28. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer **1**(1–2), 134–152 (1997)
29. Long, B., Strooper, P.A., Wildman, L.: A method for verifying concurrent Java components based on an analysis of concurrency failures. Concurrency and Computation: Practice and Experience **19**(3), 281–294 (2007)
30. Reisig, W., Rozenberg, G. (eds.): Lectures on Petri Nets I: Basic Models - Advances in Petri Nets. Springer (1998)
31. Shao, Z., Pierce, B.C. (eds.): Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009. ACM (2009)
32. Wells, L., Christensen, S., Kristensen, L.M., Mortensen, K.H.: Simulation based performance analysis of web servers. In: Proc. of PNPM'01. pp. 59–68 (2001)

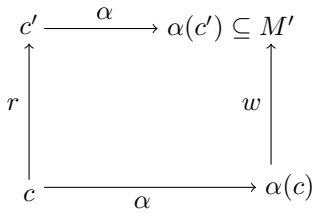
A Soundness proof

In this section, we use sequential semantics of coloured Petri nets to prove the soundness of the model. We prove an abstract weak simulation relation between ABS program configurations and net markings. For the sake of simplicity, we label (uniquely) the transitions and the places of the model with the elements of the (finite) sequences $(t_i)_{i\in\{1,\ldots,|T|\}}$ and $(p_i)_{i\in\{1,\ldots,|P|\}}$, respectively, as shown in the figures of the model. From now on, instead of referring to the transitions and the places of the model, we will be using their labels, which define the alphabets T and P respectively. A sequence $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} M_n$ of reachable markings M_i together with the enabled transitions is called an *occurrence sequence*. The transition sequences $t_1 t_2 \ldots t_n$ induced by the occurrence sequences of the net will be equivalently seen as words over the above alphabets. We will call those words *occurrence words* to distinguish them from arbitrary combinations of transitions that cannot fire the one after the other.

Let *Sem* be the set of the semantic rules of Fig. 2 related to the deployment fragment of the language, i.e.

$$Sem \stackrel{def}{=} \{Newdc, Cost1, Cost2, Transfer, RunToNewInterval\}$$

In the following, we will prove that, for any ABS configuration c, if $c \rightarrowtriangle_r c'$ for some semantic rule $r \in Sem$, then there exists a marking M' and an occurrence word w, such that $\alpha(c) \xrightarrow{w} M'$ and $\alpha(c') \subseteq M'$, as depicted in the following diagram:

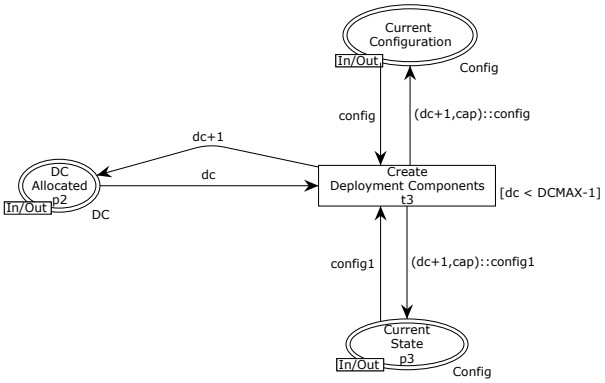


In the rest of the section, we will prove the above relation for every semantic rule of the set *Sem* exhaustively. We will denote the marking of the place p_i as M_{p_i}. When we use the operator $<$ between transitions we denote their enabling order, for example $t_1 < t_2$ means that t_1 is fired before t_2 and the $=$ for the concurrent enabledness. When some place belongs to more than one modules, we will argue about the arcs connected to it in each such a module.

A.1 Rule: *Newdc*

The module Create Deployment Components (see Fig. 9) contains one single transition (t_3) which, upon firing, creates new deployment components so, the rule *Newdc* shown in Fig. 2 is related to this transition.

Fig. 9. Module: Create Deployment Components



Transition t_3 cannot fire before t_1 of module Imperative Layer (Fig. 10), since p_2 and p_3 have initially empty markings. In addition, t_1 can fire only once, because of p_1.

Place p_3 has as colour the list of pairs consisting of the deployment component identifiers and the corresponding capacities. So, $M_{p_3} \in \alpha(c)$.

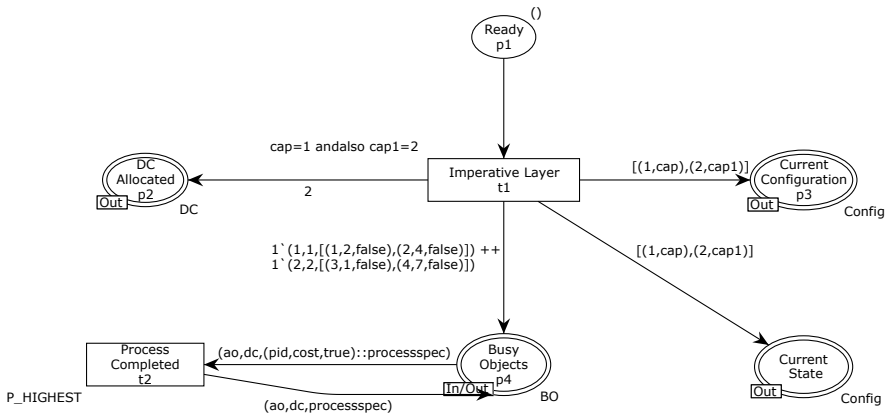


Fig. 10. Imperative Layer

Observe that the marking of p_3 is not affected by t_4 of module Resource Refill of Fig. 13 since the incoming and outcoming arc inscriptions coincide. Also, t_5 (Fig. 12) is not affecting the dc identifier because of the definition of the function "Transfer". So, we can see that *the value of the variable "dc" cannot change in any other module.*

Place p_2 has two incoming arcs: one in the module Imperative Layer (Fig. 10) and one in the Create Deployment Components module (Fig. 9). As we mentioned above, t_1 can fire only once and then t_3 is enabled. So, M_{p_2} can change only from t_1 first and then only by t_3 ($t_1 t_3$ is an occurrence word).

The colour of dc is an integer which can only increase by firing t_3. Since dc identifier cannot change from other modules, the value of dc after each firing of t_3 is unique (fresh).

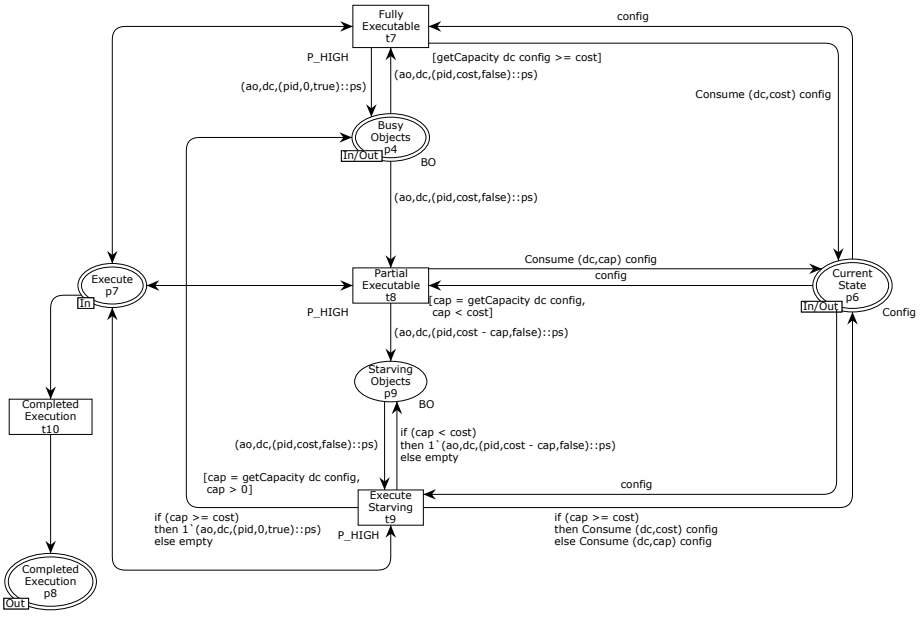
Let M' be the marking of the net after firing t_3. From the definition of α, $M'_{p_3} \in M'$, hence $\alpha(c') \subseteq M'$.

A.2 Rules: *Cost1* and *Cost2*

With respect to the resources, an (active) object can be located either in p_4 or in p_9, shown in Fig. 11. In particular, $M_{p_4} + M_{p_9} = n(c)$ is an invariant, where $n : C \to \mathbb{N}$ is the number of the active objects of some configuration $c \in C$ which are neither idle (no active process) nor blocked due some blocking call. As a consequence, both of the rules related to the cost (Fig. 2) are associated to the module Process Execution shown in Fig. 11.

Convention: In the rules *Cost1* and *Cost2* in Fig. 2 the cost is denoted as c and c'. In our proof so far we use c and c' to denote ABS configurations. For avoiding confusion, when we will refer to the costs appearing in Fig. 2 we will use cst and cst' instead and keep the notations c and c' for the configurations.

Fig. 11. Module: Process Execution



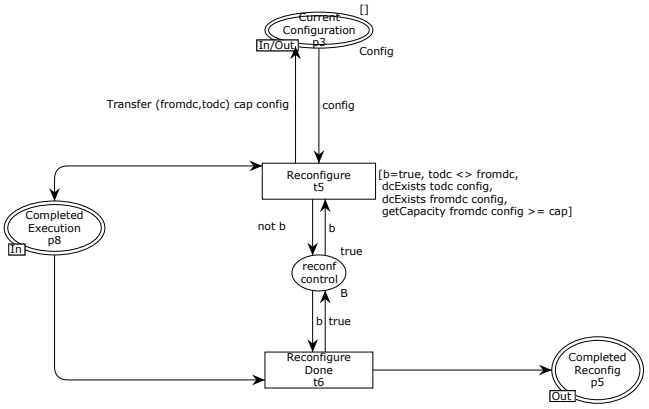
Cost1 Rule *Cost1* of Fig. 2 is related to the transitions t_7 and t_9 of the Process Execution module (Fig. 11).
case: non-starving object.
Let us assume that the rule *Cost1* has been applied to an ABS configuration c. Then, the cost is less than the available resources. From the definition of the function "getCapacity", $[\![G(t_7)]\!]_b = true$. From the abstraction function, place p_4 contains the token-objects given by $\alpha(c)$. Also, p_6, which is of colour *list*, has as elements pairs, containing the available amount of resources (in the semantic rule *Cost1* , denoted as $n - u$). Place p_7 contains a token since we are in an ABS time interval with the resources already distributed. So, t_7 can fire. In the resulting tokens, in the case of p_4, the cost has been set to zero, which corresponds to the missing cost annotation in the semantic rule of Fig. 2 after the execution of the cost-transition while, in the case of p_6, from the definition of the function "Consume", the corresponding element of the list, contains the updated amount of available resources $(n - (u + cst))$. So, in the resulting marking M', $M' \supseteq \alpha(c')$.
case: starving object .
Here, from the hypothesis of the rule, $cost \leq cap$, where *cap* is the amount of the available resources. So, according to the outcoming arc inscriptions, the object token will be moved from p_9 to p_4 updated as before. The rest of the proof is similar to the previous case.

Cost2 Similarly to the previous rule, an (active) object can be either starving or not. So, we have to distinguish as before into the two corresponding cases.
case: non-starving object In this case $o(c) \in M_{p_4}$. From the hypothesis of *Cost2*, $cost > cap$, so $[\![G(t_8)]\!]_b = true$. For p_7 and p_6 similar to the previous case. So t_8 can fire and the proof can be deduced similarly to the previous case, since in $M'(p_9)$ the cost of the objects has been updated according to the reduced cost in the semantic rule and $M' \supseteq \alpha(c')$.
case: starving object.
Easy from the previous case and *Cost1*.

A.3 Rules: *Transfer* and *RunToNewInterval*

The rule *Transfer* of Fig. 2 is related to the module Component Reconfiguration of Fig. 12 and, in particular, to the transition t_5. Variables fromdc, todc and *cap* are associated to the values dc, $dc'$and i of the hypothesis of the rule. From $\alpha(c)$ and from the definition of the function *Transfer*, t_5 can fire and reach a marking containing two pairs $(dc, cap_{new}), (dc', cap'_{new}) \in M'_{p_3}$ such that $cap_{new} = cap - i$ and $cap'_{new} = cap' + i$. These elements of M'_{p_3} correspond to the values $k - i$ and $k + i$ of the configuration c', hence $M' \supseteq \alpha(c')$. What remains to be proved is that those values are associated to the parameter k of the semantic rule and not to the parameter n, i.e. that the transfer of the resources will take place at the new time interval. For this, observe that t_5 is connected to p_3 and not to p_6 and that, in the top level module of Fig. 3, substitution transition Process Execution is connected only to the place Current State and not to the Current Configuration.

Fig. 12. Module: Component Reconfiguration



In the module Resource Refill of Fig. 13, if M and M' are the markings of the net before and after firing t_4, then $M_{p_3} = M'_{p_3}$ and $M'_{p_6} = M_{p_3}$, i.e. the marking of p_6 is updated with the marking of p_3. Place p_3 appears also in the modules Component Reconfiguration (Fig. 12) and Create Deployment Components (Fig. 9). Also $t_6 < t_4$ because of p_5, while $t_5 = t_6$. Also $t_4 < t_{10}$ because of p_7 and $t_{10} < t_6$. Also, the priorities in the module Process Execution (Fig. 11) guarantee the maximal process execution before t_{10}, hence the rule *RunToNewInterval* (Fig. 2) is also satisfied. Obviously, for both the above rules $M' \supseteq \alpha(c')$ holds for some marking M' reached after firing transitions described above.

Fig. 13. Module: Resource Refill