Model Checking Starvation for Resource-aware Active Objects with Coloured Petri Nets

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. Dynamic resource provisioning is an important driver for pay-on-demand cloud computing. Virtualized resources open for resource awareness, such that applications may use resource management strategies to modify their deployment resource consumption at run-time. The ABS language supports the modeling of deployment decisions and resource management for active objects. An important property in this context is to ensure that the resource management does not lead to starvation of the executing objects. In previous work, we have formally translated the semantics of the ABS language into a parameterized Coloured Petri Net (CPN) model, such that any ABS program can be represented by setting the initial marking accordingly. In this paper, we characterize starvation using Computation Tree Logic (CTL), and demonstrate how CTL model checking of the CPN encoding of the ABS model, in combination with path finding, can be used to detect starvation and synthesize load balancers that guarantee starvation freedom.

1 Introduction

Pay-on-demand resource provisioning is an important driver for cloud computing [16]: Using resources on the cloud to deploy a service, the service provider does not need to cater hardware resources upfront to launch the service but can lease resources as required depending on demand. 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. This software layer allows deployment decisions to be programmed. Virtualized resources open for resource-aware applications; these applications may contain resource management strategies to modify their own or other applications' deployment and reduce resource consumption. In this context, it is interesting to model and analyze deployment scenarios for services with respect to client traffic in order to, e.g., establish the amount of resources required for timely delivery of a service.

Programming models that *decouple* control flow and communication, such as Actors [1, 2] and active objects [9, 17, 33], inherently support both scalability (as argued with the Erlang programming language [5] and Scala's actors [29])

and compositional reasoning [12, 20–22]. These features are also interesting for distributed services that should adapt to elastic cloud deployment. For these services, this decoupling may be exploited to make deployment decisions and their validation a part of the design phase rather than a post-hoc activity [28]: The elasticity of software executed in the cloud gives designers control over the execution environment's resource parameters, such as the number and kind of processors, memory, storage capacity, and bandwidth. ABS is a formally defined active object language [33, 36], which directly supports the modeling of deployment decisions and resource management for active objects, and which has been used for industrial case studies of cloud computing services [4]. When executing an ABS program, it is clearly important to ensure that *computing resources are not over-provisioned* as this leads to unused computing resources. At the same time, it is equally important to ensure that the *computing resources are not under-provisioned* as it may lead to starvation of objects, i.e., that the execution of some objects are blocked due to the lack of computing resources.

In this paper, we develop a method to investigate resource distribution for deployed active objects in ABS programs by a translation into Coloured Petri Nets (CPNs) [32]. The method provides an automated approach to reason about the starvation of objects in the ABS program under different deployment scenarios. Our approach uses Computation Tree Logic (CTL) [19] to formalize the notions of strongly starvation free states (states from which starvation cannot occur), weakly starvation free states (states from which starvation may occur), and inevitable starving states (states from which starvation will eventually occur). We use the ASK-CTL model checker [18] of CPN Tools to automatically identify such states and synthesize resource allocation strategies that ensure starvation freedom. We extend previous work on encoding behavioral ABS models [23, 24] and deployment models [25] in CPN, such that the formal semantics of deployment models in ABS is captured directly as a hierarchical CPN. The number of places in the CPN model is independent of the size of a program, and a specific ABS program can be represented by setting the initial marking (i.e., the initial state) of the CPN model accordingly. The modeling approach captures how computation in the behavioral ABS model interacts with virtual resources and allows virtual resources to be dynamically launched in the CPN model by the firing of CPN transitions.

The paper is organized as follows: Section 2 introduces the ABS language, focusing on the modeling of deployment. Section 3 presents the CPN encoding of the ABS semantics. Section 4 shows how the CTL model checking and path finding can be used for the resource analysis of ABS programs, and in Section 5 discusses related work and summarizes our conclusions.

2 Deployment Modeling in ABS

ABS [33] is a formally defined actor-based language for the executable modeling of distributed, object-oriented systems. ABS supports *deployment modeling* by separation of concerns between the resource costs of executions and the resource

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

Fig. 1. ABS syntax. Overbar notation denotes lists.

capacities of *deployment components* on which executions take place [36]; deployment components can be understood as (virtual) locations for computation. Deployment decisions can be made inside models, by allocating active (also called concurrent) objects to deployment components with given resources at creation time (e.g., [4, 34]).

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 Figure 1. A program P consists of class definitions CL which contain field declarations $T\ x$ (where T is the type of field x) and method definitions M, and a main block. We follow the syntactic conventions of Java and only explain syntax that differs from Java.

The Imperative Layer. The imperative layer of ABS is used for internal control flow, and for communication and synchronization between concurrent objects. Objects are instantiated from classes by the statement [DC: *server*] $o =$ **new** $C(\overline{e})$, where $\overline{e}$ are constructor arguments and the optional annotation DC: *server* expresses the deployment component *server* on which the new object should be created. 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.**get** retrieves the return value from a future f.

Objects combine reactive and active behavior (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 **duration**(b, w) where b and w are bounds on the time interval before the condition

$$\frac{fresh(dc) \quad [\![e]\!]_{a \circ l} = n}{o(a, \{l \mid x = \textbf{new}\ \mathtt{DC}(e); s\}, q) \to o(a, \{l \mid x = dc; s\}, q)\ dc(n, 0, n)}\ (\text{New-DC})$$

$$\frac{blocked(cn, t) \quad 0 < d \leq mte(cn, t) \quad \lceil t \rceil = t + d}{\{cn\ cl(t)\} \to_t \{timeAdv(rscRefill(cn), d)\ cl(t + d)\}}\ (\text{Run-To-New-Interval})$$

$$\frac{a(\textbf{thisDC}) = dc \quad an = \mathtt{Cost}\colon e \quad [\![e]\!]_{a \circ l} = cst \quad cst \leq n - u \quad o(a, \{l \mid [an']\ s\}, q)\ cn \to o(a', p', q')\ cn'}{o(a, \{l \mid [an]\ s\}, q)\ dc(n, u, k)\ cl(t)\ cn \to o(a', p', q')\ dc(n, u + cst, k)\ cl(t)\ cn'}\ (\text{Cost1})$$

$$\frac{a(\textbf{thisDC}) = dc \quad an = \mathtt{Cost}\colon e \quad [\![e]\!]_{a \circ l} = cst \quad cst > n - u \quad n \neq u \quad cst' = cst - (n - u) \quad an' = \mathtt{Cost}\colon cst'}{o(a, \{l \mid [an]\ s\}, q)\ dc(n, u, k)\ cn \to o(a, \{l \mid [an']\ s\}, q)\ dc(n, n, k)\ cn}\ (\text{Cost2})$$

$$\frac{fresh(f) \quad [\![e]\!]_{a \circ l} = dc \quad [\![e']\!]_{a \circ l} = dc' \quad [\![e'']\!]_{a \circ l} = i \quad i \leq k}{o(a, \{l \mid x = e!\textbf{transfer}(e', e''); s\}, q)\ dc(n, u, k)\ dc'(n', u', k') \to o(a, \{l \mid x = f; s\}, q)\ dc(n, u, k - i)\ dc'(n', u', k' + i)\ f(i)}\ (\text{Transfer})$$

Fig. 2. Semantics of the deployment layer of ABS.

becomes true. ABS supports the modeling of dense time [8]; the local passage of time is expressed in terms of durations (as in, e.g., UPPAAL [38]).

The Deployment Layer. Deployment models capture physical or virtual infrastructure in ABS using dynamically created *deployment components* [35, 36] to represent computing environments. A deployment component is a modeling abstraction which represents 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 **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 [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 *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 $\to$ over configurations realizing a maximal progress time model, in which time will only advance if the execution is otherwise blocked. We here focus on the transition rules formalizing the *cost and deployment aspects* of the execution of ABS programs (shown in Figure 2). Configurations include *objects* $o(a, p, q)$, where o is an object identifier, a a state, p an active process, and q a 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.

The deployment components keep track of the resource consumption of their 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 each time interval. Rule Run-To-New-Interval captures the advancement of time. Here, the brackets enclose all objects in the configuration as well as a global clock *cl(t)* to ensure that time advances uniformly. The predicate $blocked(cn, t)$ expresses that no (further) reduction is possible in cn at time t, so time may advance. 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 of executing a statement if the deployment component can provision some but not all of the required resources. Rule Transfer 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 [36].

3 A CPN Model of ABS Semantics

The communication and concurrency aspects of ABS was presented in [24] as a CPN model, where active objects were 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. This paper extends [25] where we focused on the deployment part of ABS. In [25] we presented a new hierarchical CPN, modeling the deployment fragment of the language. We modeled 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 Figure 3 where we have the top-layer of the model with dotted line around the program execution cycle. In the bottom part of this figure, we see the resource refill happens before the process execution in the next cycle.

This model takes as input tokens that can be produced from the imperative part [24] of ABS as described above, and we add information concerning the cost of each process and the deployment component they are located in. This

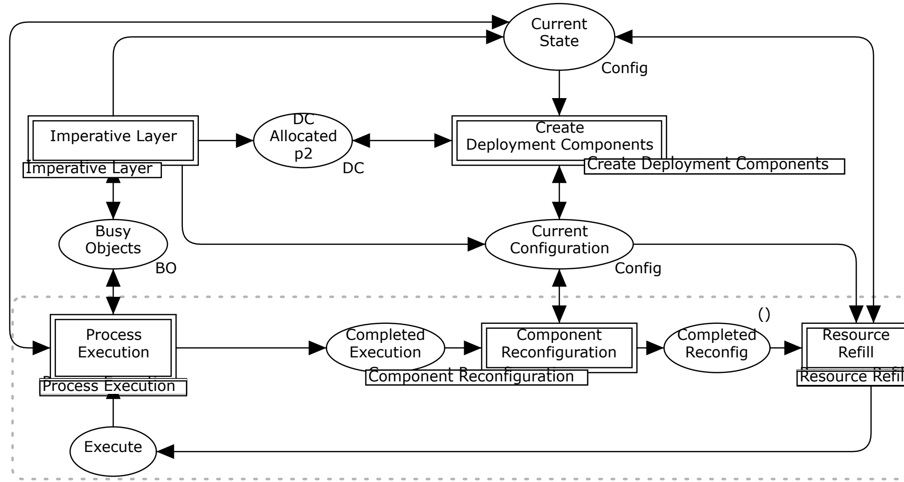

Fig. 3. Top-level module of the CPN deployment model

information, together with the deployment semantics of ABS can be used to verify starvation freedom of active objects and explore resource management strategies. In this paper, we extend the resource analysis of [25] by a characterization of starvation based on CTL model checking as will be presented in Section 4. In the rest of this section, we briefly present the CPN model of [24] on top of which we extend the starvation analysis.

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

The ABS program that we use as an example is inspired by the behavior of cellphone clients at midnight on New Year's eve, with a sudden change from regular to irregular behavior. Our aim is 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 the current distribution. However, there are some particular moments of the year like, for example, around the midnight of new year's eve, where this behavior 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 Figure 4, we provide the ABS implementation of the above scenario [36] where telephone and SMS servers have been realized with the two corresponding classes TelephoneServer and SMSserver. The operational costs are annotated in square brackets at the beginning of the statements (line 3 and line 7). 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.

```
 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.

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 analyzed by setting the initial marking accordingly. In our example, we modeled the SMS and the telephone servers in CPNs as two different tokens representing the corresponding objects of Figure 4 (tel and sms). Those tokens have as colour (values) 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 behavior. 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 transition Imperative Layer is enabled. Recall that the colour of the object tokens have the form (ob, dc, lst) as explained above. In Figure 5, we have two tokens produced in place Busy Objects. The first one represents the TelephoneServer object with the identifier 1 located in the first deployment component and has two processes in its process pool: 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 (for details, see [26]).

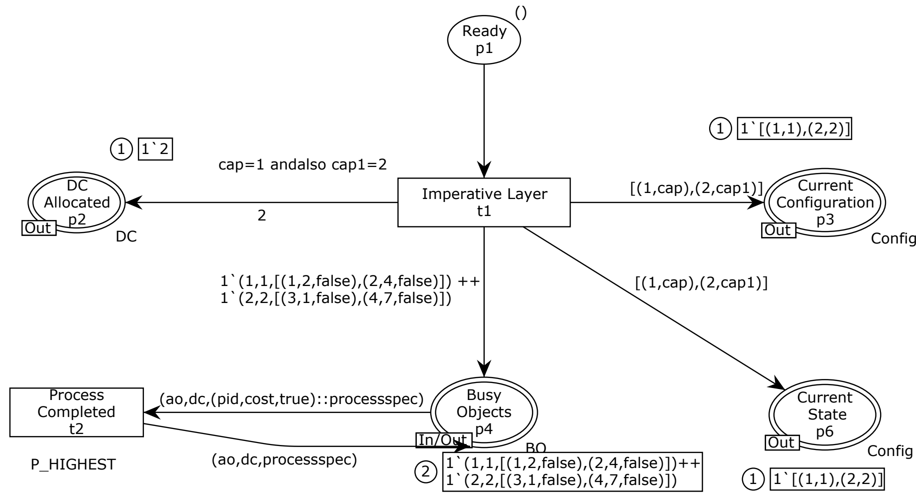

Fig. 5. CPN module of the imperative layer

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).

Figure 6 shows that when transition Reconfigure fires, the marking of the place Current Configuration 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 transfers resources from one deployment component to another. When transition Reconfigure Done fires, the reconfiguration has been completed. Then the resources can be refilled (details of the related implementation can be found in [26]), 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 Figure 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

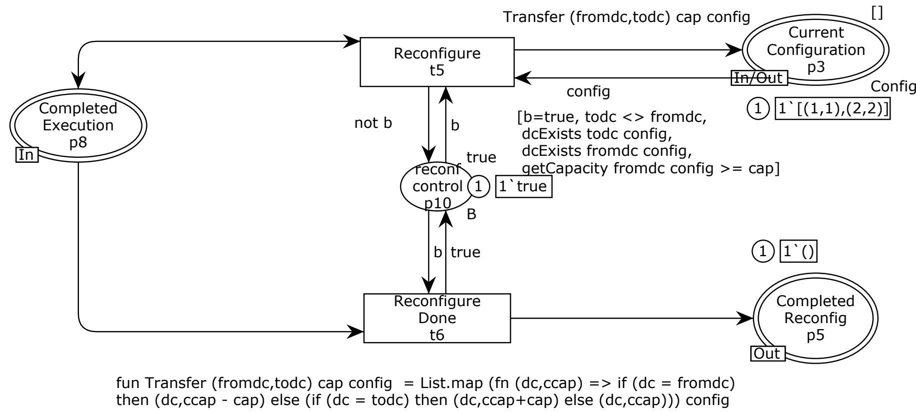

Fig. 6. CPN module for component reconfiguration

(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 has been fully executed or not). After this, transition Process Completed of Figure 5 is enabled and the corresponding element of the list (head) is removed.

Consider again Figure 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 is enabled. Then the token of object 1 will be moved to the place Starving Objects with the remaining cost updated to 1, until the marking of place Current State shows resource availability at the deployment component greater or equal to 1. This can be done at the next cycle in the model, after possible resource transfer and refill. In such a case, transition Execute Starving will be enabled and send the 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.

4 Resource Analysis and Management

We now show how explicit state space exploration and CTL model checking of the CPN model can be used to reason about starvation properties of an ABS program. In presence of starvation, we demonstrate how the state space of the CPN model can be used to synthesize a sequence of resource reconfigurations which can eliminate starvation. Finally, we show how a sequence of resource reconfigurations can be used to automatically obtain an implementation of a starvation free load balancer.

For the experiments related to resource analysis, we rely on the state space exploration support in CPN Tools in combination with the ASK-CTL library [18] supporting model checking using a state- and action-oriented variant of the

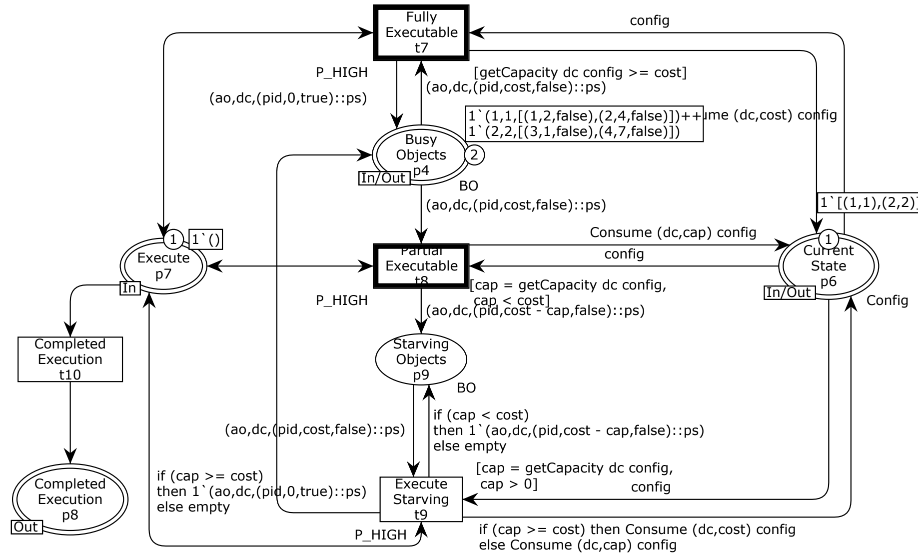

Fig. 7. CPN module for process execution

Computation Tree Logic (CTL) [19]. All experiments have been conducted on an Intel i7 3.4 GHz PC with 16 GB of memory. We use the running example from the previous section for illustration purposes, but our analysis approach generalizes to all instantiations of the CPN model, i.e., independently of which ABS program is considered. The state space for the running example has 776 nodes (states) and 1,069 arcs (occurring events), and could be generated in less than 1 second.

4.1 Starvation Analysis

Section 3 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 shown in Figure 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. By model construction, place Starving Objects keeps track of starving objects, i.e., objects whose execution has been blocked because of lack of resources at the current time interval. This in turn means that the current marking of this place gives information about whether the current resource distribution provides sufficient resources for the full execution of the processes the objects have in their process pools. This makes it possible to implement a state predicate `starving` as a Standard ML function in CPN Tools shown below to determine whether there is any starving objects in a given marking.

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

fun starving M = (findStarvingObjects s) <> nil
```

The function `starvingObjects` takes a marking (state) M 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 M. This function is then used in the predicate `starving` which can be used to determine whether or not there are any starving objects in state M.

For our running example, CPN Tools returns a non-empty list containing several markings when invoking the `starving` predicate on each marking of the state space. This means that starvation is possible. In order to provide a more detailed account of starvation freedom, we now use the temporal operators from CTL to characterize various forms of starvation. We use ϕ_{strv} as an atomic proposition on states equivalent to the `starving` state predicate implemented in Standard ML above, and we use the standard notation from temporal model checking $M \models \Phi$ to denote that a CTL formula Φ is satisfied in the marking (state) M.

The characterization of starvation freedom is based on the standard CTL operators of **AG** (always globally), **EG** (exists globally) and **AF** (always eventually), where $\mathbf{AG}\,\Phi$ holds in a marking M if Φ holds in all markings reachable from M; $\mathbf{EG}\,\Phi$ holds in a marking M if there exists a path starting in M such that Φ holds in all markings along the path; and $\mathbf{AF}\,\Phi$ if all paths starting in M contains a marking where Φ holds. Based on this, we formalize the characterization of starvation freedom as follows:

Definition 1. *Let $\mathcal{M}_{reach}$ be the set of reachable states of the CPN model, and let $\phi_{strv} : \mathcal{M}_{reach} \rightarrow \mathbb{B}$ be an atomic state proposition which is satisfied in a marking iff there are starving objects in the marking. Let $M \in \mathcal{M}_{reach}$ be a reachable state, then:*

– M *is* ***strongly starvation free*** *iff* $M \models \boldsymbol{AG}\neg\phi_{strv}$
– M *is* ***weakly starvation free*** *iff* $M \models \boldsymbol{EG}\neg\phi_{strv}$
– M *is* ***inevitably starving*** *iff* $M \models \boldsymbol{AF}\phi_{strv}$

The model is ***starvation free*** *if the initial state* M_0 *is strongly starvation free.*

It follows from the above definition that a marking being strongly starvation free implies that starvation will never occur once that marking has been entered. For weakly starvation free markings, there exist executions in which starvation can be avoided depending on the execution of objects and the reconfigurations performed. For our running example, there are 128 strongly starvation free markings which by definition are also weakly starvation free. There are no weakly starvation free markings that are not also strongly starvation free. So in this case, the two sets of markings coincides. The running example has 648 markings in which starvation is inevitable. The set of inevitable starving markings includes the initial marking which implies that with the given initial resource allocation, starvation is inevitable when starting the system from the initial resource distribution.

Since the initial resource distribution of our CPN model may lead to starvation, an interesting question is *whether there exists a resource reallocation strategy leading to a strongly starvation free state*. To determine this, we conduct a breadth-first search of the state space in order to find one of the shortest paths leading from the initial marking to a strongly starvation free marking. For such a path, we are interested only in the information related to resource transfer. Recall that the module Component Reconfiguration (see Figure 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 bindings of this transition, where the binding specifies the values bound to the variables of the transition. This is the synthesized 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 representing by the bindings of the Reconfigure transition:

```
{cap = 1, config = [(1,1),(2,2)], fromdc = 1, todo = 2}
{cap = 3, config = [(1,0),(2,3)], fromdc = 2, todc = 1}
{cap = 2, config = [(1,3),(2,0)], fromdc = 1, todc = 2}
```

where *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.

4.2 Implementation of 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 possible starvation to synthesize a path from the initial resource distribution to a starvation free marking. 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 function, 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. 4.1. For the sake of simplicity, we present only the name of the transition followed by the corresponding binding. The enumeration on the left corresponds to the respective ABS time point:

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

In the above path, the highlighted lines are resource transfers that will lead to a starvation free state, as we saw in Sect. 4.1. In our example, we consider two objects located in two deployment components. The first line shows the resource initialization. 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 initialization 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 Figure 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 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.

```
1  class Balancer(DC telcomp, DC smscomp) {
2
3    Unit run() {
4
5      telcomp!transfer(smscomp,1);
6      await duration(2,2);
7      smscomp!transfer(telcomp, 3);
8      await  duration(2,2);
9      telcomp!transfer(smscomp,2);
10
11   }
12 }
13
14 {
15   // Main block
16   ... // deployment components, etc. as before
17   new Balancer(telcomp,smscomp);
18 }
```

Fig. 8. Synthesized ABS load balancer implementation.

5 Related Work

Some of the earliest applications of CPNs for analysis of distributed objects appeared in [37] focussing on spatial distribution of objects and not resource consumption. Early work [41] also considered simulation-based capacity planning of web-servers, but not in a context with dynamically configurable resources. CPNs have also recently been used to analyze deadlock situations for active objects with futures by de Boer et al. [10, 11] and by the authors [24], as found in the ABS language. More recent work [14] 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 does not provide resource management. Recent work [30] has also explored evaluation of cloud deployment strategies for distributed NoSQL databases using CPN simulation, but without dynamic reconfiguration. In contrast to previous modeling of programming languages into Petri nets like Ada [31], Java [39], Orc [15], where the model depends on the program, we suggest a fixed sized model where the markings are program configuration abstractions. This means that different programs can be analyzed by one single model upon different initialization according to the abstraction function.

More broadly, process algebras [7], priced [13] 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 [40] and [27], aim 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 possible starvation, proposes strategies for vertical scaling that can be retrieved by the counter examples of CPN Tools.

Our present work extends [24] 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 CTL model checker. Another direction will be to perform a comprehensive experimental evaluation on a larger set of ABS programs.

6 Conclusion

We have presented a CPN model of the deployment layer of ABS [36], a resource aware programming language suitable for cloud applications. A key characteristic 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. This work adds the deployment layer of ABS to our previous work translating the imperative part of the language and analysing the communication of the active objects according to the semantics of the language [23, 24]. In the current work, we focus on the representation of the

deployment features of the language which induce resource awareness and take as an input to our previous analysis the communication status of the objects. This separation of concerns follows naturally the corresponding layers of ABS, since the communication of the active objects (i.e. the method calls) is related to the process creation, hence the cost of the program. Here, we combine this information to the resource availability and use the model checker of CPN Tools for resource management options over ABS time intervals. The main benefit of our approach is the ability to use model checking techniques to identify starvation for resource-aware active objects, and to synthesize reconfiguration sequences that eliminate starvation and which in turn can be used to automatically obtain load-balancer implementations.

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. 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., Bravetti, M., Grabe, I., Lee, M.D., Steffen, M., Zavattaro, G.: A Petri Net based analysis of deadlocks for active objects and futures. In: Pasareanu, C.S., Salaün, G. (eds.) Proc. 9th International Symposium Formal Aspects of Component Software (FACS 2012). LNCS, vol. 7684, pp. 110–127. Springer (2013)
11. de Boer, F.S., Bravetti, M., Lee, M.D., Zavattaro, G.: A Petri Net based modeling of active objects and futures. Fundam. Inform. **159**(3), 197–256 (2018)
12. 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)
13. 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)
14. 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)

15. 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)
16. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Comp. Syst. **25**(6), 599–616 (2009)
17. Caromel, D., Henrio, L.: A Theory of Distributed Objects. Springer (2005)
18. Cheng, A., Christensen, S., Mortensen, K.: Model checking coloured petri nets exploiting strongly connected components. In: Proc. of the International Workshop on Discrete Event Systems (WODES). pp. 169–177 (1996)
19. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press (2000)
20. 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)
21. Din, C.C., Owe, O.: Compositional reasoning about active objects with shared futures. Formal Asp. Comput. **27**(3), 551–572 (2015)
22. Din, C.C., Tapia Tarifa, S.L., Hähnle, R., Johnsen, E.B.: History-based specification and verification of scalable concurrent and distributed systems. In: Butler, M.J., Conchon, S., Zaïdi, F. (eds.) Proceedings of the 17th International Conference on Formal Engineering Methods (ICFEM 2015). LNCS, vol. 9407, pp. 217–233. Springer (2015)
23. Gkolfi, A., Din, C.C., Johnsen, E.B., Kristensen, L.M., Steffen, M., Yu, I.C.: Translating active objects into colored Petri nets for communication analysis. Sci. Comput. Program. **181**, 1–26 (2019)
24. 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)
25. Gkolfi, A., Johnsen, E.B., Kristensen, L.M., Yu, I.C.: Using coloured petri nets for resource analysis of active objects. In: Bae, K., Ölveczky, P.C. (eds.) Proceedinngs of the 15th International Conference on Formal Aspects of Component Software (FACS 2018). LNCS, vol. 11222, pp. 156–174. Springer (2018)
26. Gkolfi, A., Johnsen, E.B., Kristensen, L.M., Yu, I.C.: Using Coloured Petri Nets for resource analysis of active objects (full version). Tech. Rep. 484, Dept. of informatics, University of Oslo (2018)
27. Gordon, A.D. (ed.): Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, LNCS, vol. 6012. Springer (2010)
28. Hähnle, R., Johnsen, E.B.: Designing resource-aware cloud applications. IEEE Computer **48**(6), 72–75 (2015)
29. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. **410**(2–3), 202–220 (2009)
30. 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)
31. 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. Cambridge University Press (1980)
32. Jensen, K., Kristensen, L.M.: Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer (2009)
33. 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)

34. 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)
35. 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, pp. 71–86. Springer (Nov 2012)
36. 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)
37. 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)
38. 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)
39. 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)
40. Shao, Z., Pierce, B.C. (eds.): Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009. ACM (2009)
41. 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)