Towards the typing of resource deployment *

Elena Giachino and Cosimo Laneve

Dept. of Computer Science and Engineering, Università di Bologna – INRIA FOCUS
{giachino,laneve}@cs.unibo.it

Abstract. In cloud computing, *resources* as files, databases, applications, and virtual machines may either scale or move from one machine to another in response to load increases and decreases (*resource deployment*). We study a type-based technique for analysing the deployments of resources in cloud computing. In particular, we design a type system for a concurrent object-oriented language with dynamic resource creations and movements. The type of a program is *behavioural*, namely it expresses the resource deployments over periods of (logical) time. Our technique admits the inference of types and may underlie the optimisation of the costs and consumption of resources.

1 Introduction

One of the prominent features of cloud computing is elasticity, namely the property of letting (almost infinite) computing resources available on demand, thereby eliminating the need for up-front commitments by users. This elasticity may be a convenient opportunity if resources may go and shrink automatically at a fine-grain when user's needs change. However, current cloud technologies do not match this fine-grain requirement. For example, Google AppEngine automatically scales in response to load increases and decreases, but it charges clients by the cycles (type of operations) used; Amazon Web Service charges clients by the hour for the number of virtual machines used, even if a machine is idle [2].

Fine-grained resource management is an area where competition between cloud computing providers may unlock new opportunities by committing to more precise cost bounds. In turn, such cost bounds should encourage programmers to pay attention to resource managements (that is, releasing and acquiring resources only when necessary) and allow more direct measurement of operational and development inefficiencies.

In order to let *resources*, such as files or databases or applications or memories or virtual machines, be deployed in cloud machines, the languages for programming the cloud must include explicit operations for creating, deleting, and moving resources – *resource deployment operations* – and corresponding software development kits should include tools for analysing resource usages. It is worth to observe that the leveraging of resource management to the programming language might also open opportunities to implement Service Level Agreements

* Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services.

(SLAs) validation via automated test infrastructure, thus offering the opportunity for third-party validation of SLAs and assessing penalties appropriately.

We study resource deployment (in cloud computing) by extending a simple concurrent object-oriented model with lightweight primitives for dynamic resource management. In our model, resources are *groups of objects* that can be dynamically created and can be moved from one (*virtual*) machine to another, called *deployment components*. We then define a technique for analysing and displaying resource loads in deployment components that is amenable to be prototyped.

The object-oriented language is overviewed in Section 2 by discussing in detail a few examples. In Section 3, we discuss the type system for analysing the resource deployments. Our technique is based on so-called *behavioural types* that abstractly describe systems' behaviours. In particular, the types we consider record the creations of resources and their movements among deployment components. They are similar to those ranging from languages for session types [7] to process contracts [17] and to calculi of processes as Milner's CCS or pi-calculus [19, 20]. In our mind, behavioural types are intended to represent a part of SLA that may be validated in a formal way and that support compositional analysis. Therefore they may play a fundamental role in the negotiation phase of cloud computing tradings.

The behavioural types presented in Section 3 are a simple model that may be displayed by highlighting the resource load of deployment components using existing tools. We examine this issue in Section 4. Related works are discussed in Section 5.

The aim of this contribution is to overview our type system for analysing resource deployments. Therefore the style is informal and problems and (our) solutions are discussed by means of examples. The details of the technique, such as the system for deriving behavioural types automatically and the correctness results, can be found in the forthcoming full paper.

2 `dcABS` in a nutshell

Our study targets an `ABS`-like language. `ABS` [13] is a basic abstract, executable, object-oriented modelling language with a formal semantics. In this language, method invocations are asynchronous: the caller continues after the invocation and the called code runs on a different task. Tasks are cooperatively scheduled: every group of objects, called *cog*, has at most one active task at each time. Tasks running on different cogs may be evaluated in parallel, while those running on the same cog must compete for the lock and interleave their evaluation. The active task of a cog explicitly returns the control in order to let other tasks progress. Synchronisations between caller and callee is explicitly performed when callee's result is strictly necessary by using *future variables* (see [5] and the references in there).

In our language, which is called `dcABS`, programmers may define *a fixed number* of (virtual) cloud computing machines, called *deployment components* (de-

```
1  // class C declaration:
2    class C {
3      Bool m (C x) {
4        if (@this != @x) moveto @x else moveto d1;
5        return true; }
6    }
7
8  // available deployment components declaration:
9    data DCData = d0, d1, d2, d3;
10
11 //main statement:
12   C x1 = new cog C( ); moveto d1;
13   C x2 = new cog C( ); moveto d2;
14   C x3 = new cog C( ); moveto d3;
15   Fut<Bool> f1 = x1!m(x2);
16   Fut<Bool> f2 = x2!m(x3);
17   Bool b1 = f1.get;
18   Fut<Bool> f3 = x3!m(x2);
19   Bool b2 = f2.get;
20   Bool b3 = f3.get;
```

Table 1. A simple `dcABS` program

ployment component *do not scale*), and may use a very basic management of resources that enables cogs movements from one deployment component to another (*cogs represents generic resources*, such as group of computing entities, databases, virtual memories and the corresponding management processes). In `dcABS`, we also assume that *method invocations are synchronised in the same method body where they occur*, except for the main statement. This constraint largely simplifies the analysis and augment its precision because it reduces the nondeterminism.

We illustrate the main features of `dcABS` by means of examples. The details of the syntax and semantics of `dcABS` can be found in the (forthcoming) full paper. Table 1 displays a simple `dcABS` program. Programs consist of three parts: (i) a collection of class definitions, (ii) a declaration of the deployment components that are available, and (iii) a main statement to evaluate. Classes contain field and method declarations. In the above table, there is one class definition that covers lines 2–6, the deployment components are declared at line 9, and the main statement covers lines 12–20. The evaluation of the main statement is performed in the special cog `start` that is located on the deployment component that is declared first; in our example this is `d0`.

Line 12 contains a definition of `dcABS`: it creates a new object of class `C` in a new cog, locally deployed, and stores a reference to the new object in the variable `x1`. The subsequent statement `moveto d1` specifies the migration of the

```
21  class D {
22    Bool move ( ) { moveto d1 ; return true ; }
23
24    Bool multi_create (Int n) {
25      if (n<=0) return true ;
26      else { D x = new cog D ( ) ;
27        Fut<Bool> f = x!multi_create(n-1) ;
28        Bool u = f.get ;
29        Fut<Bool> g = x!move( ) ;
30        Bool v = g.get ;
31        return true ; } }
32  }
```

Table 2. A `dcABS` recursive program

current cog, i.e. cog `start`, from the current deployment component `d0` to the deployment component `d1`.

Lines 15, 16 and 18 display method invocations. As mentioned above, in `dcABS` invocations are *asynchronous*: the caller continues executing *in parallel with* the called method, which runs in a dedicated task within the cog where the receiver object resides. For example, line 15 corresponds to spawning the instance of the body of method `m` in a new task that is going to run in the cog of the object referred by `x1`. A *future reference* to the returned value is stored in the variable `f1` that has type `Fut<Bool>`. This means that the value is not ready yet and, when it will be produced (in the future), it will have type `Bool`. Line 17 enforces the retrieval of such value by accessing to the corresponding future reference and waiting for its availability, by means of the operation `get`. Since method invocations are asynchronous, the two invocations in lines 15 and 16 are executed concurrently. The invocation at line 15 is then synchronised at line 17, but the one at line 16 may continue concurrently with the invocation of line 18, until they are both synchronised.

The invocations in the main statement execute three instances of method `m`. Every instance verifies if the receiver object is co-located with the argument object and, in case, it performs either a deployment to let the corresponding cogs be co-located or a deployment to the component `d1`. The expression `@x` of line 4 points to the deployment component where the (cog of the) object referred by `x` resides.

Table 2 shows a class definition `D` with a recursive method `multi_create`. This method creates n new cogs co-located with the caller object and moves them to the deployment component `d1`.

Analysing the cog-deployment of the programs in Tables 1 and 2 is not straightforward. For example, significant questions regarding Table 1 are: (i) *what is the cog-load of the component* `d1` *during the lifetime of the main state-*

ment? (ii) *Can the component* `d0` *be garbage-collected after a while in order to optimise resource usages*? Let the main statement of Table 2 be

```
33 // available deployment components declaration:
34   data DCData = d0, d1 ;
35
36 //main statement:
37     D x = new cog D() ;
38     Fut<Bool> f = x!multi_create(10) ;
```

Then, an important question about Table 2 is: (iii) *is there an upper bound to the number of cogs deployed to* `d0`? The technique we study in the following sections lets us to answer to such kind of questions in a formal way.

3 Behavioural types for resource deployment

Our technique for analysing resource deployments in `dcABS` programs is mostly based on our past experience in designing type inference systems for analysing deadlock-freedom of concurrent (object-oriented) languages [8–10].

A basic ingredient of every type system is the definition of the association of types with language constructs. The type system of `dcABS` associates an abstract deployment behaviour to every statement and expression. Formally, the association is defined by the typing judgment

$$\Gamma ; n \vdash_c s \,:\, \mathbb{b} \,\rhd\, \Gamma' ; n' \tag{1}$$

to be read as: in an environment Γ and at a timestamp n, the statement s of an object whose cog is c has a type $\mathbb{b}$ and has effects Γ' and n'. The pair Γ' and n' is used to type the continuation. To explain (1), consider the line 12 of Table 1:

```
12   C x1 = new cog C( ); moveto d1;
```

The statement `C x1 = new cog C( );` has two effects: (i) creating a new co-located cog (with a fresh name, say c_1) , and (ii) populating this new cog with a new object whose value is stored in `x1`. As regards (i), there is a deployment of the new cog at the deployment component where the current cog c resides. We define this behaviour by means of the type

$$c_1 \mapsto c$$

As regards (ii), we record (in the typing judgment) the name of the cog of `x1`. In particular, variable assignment may propagate cog names throughout the program and this may affect the behavioural types. That is, our type system includes the *analysis of aliases* (*c.f.* Γ' in (1) is an update of Γ). In particular, in order to trace propagations of names, we associate to each variable a so-called *future record*, ranged over by $\mathbb{r}$ and defined in Table 3. A future record may be either (*i*) a dummy value $\_$ that models primitive types, or (*ii*) a record

$$\mathbb{r} ::= \_ \mid X \mid [cog{:}c, \overline{x}{:}\overline{\mathbb{r}}] \mid \mathit{fut}(\mathbb{r}) \qquad \text{future record}$$

$$\mathbb{b} ::= \mathtt{0} \mid \langle c \mapsto c' \rangle^{n \div n} \mid \langle c \mapsto \mathtt{d} \rangle^{n \div n} \mid \langle \mathtt{C!m}(\overline{\mathbb{r}}) \to \mathbb{r}' \rangle^{m \div n} \qquad \text{behavioural type}$$
$$\mid \mathbb{b} + \mathbb{b} \mid \mathbb{b} \parallel \mathbb{b} \mid \langle \mathbb{b} \rangle^{m \div n}$$

Table 3. Future records and behavioural types of `dcABS`

name X that represents a place-holder for a value and can be instantiated by substitutions, or (*iii*) $[cog{:}c, \overline{x}{:}\overline{\mathbb{r}}]$, which defines an object with its cog name c and the values for fields of the object, or (*iv*) $\mathit{fut}(\mathbb{r})$, which is associated to method invocations returning a value with record $\mathbb{r}$. As regards Line 12, since `C` has no field, we record in the environment Γ' of (1) the binding `x1`: $[cog : c_1]$, where c_1 is a fresh cog name.

The statement `moveto d1` corresponds to migrating the current cog (*i.e.* c) to the deployment component `d1`. This is specified by the type

$$c \mapsto \mathtt{d1}.$$

The above ones are the basic deployment informations of our type system. We next discuss the management of method invocations, which is the major difficulty in the design of the type system. In fact, the execution of methods' bodies may change deployment informations and these changes, because invocations are asynchronous, are the main source of imprecision of our analysis. Consider, for example, line 15 of Table 1

```
15  Fut<Bool> f1 = x1!m(x2);
```

and assume that the environment Γ (and Γ') in (1) binds method `m` as follows

$$\Gamma(\mathtt{C.m}) = ([cog : c], [cog : c']) \to \_$$

where

- $[cog : c]$ and $[cog : c']$ are the future records of the receiver and of the argument of the method invocation, respectively,
- $\_$ is the future record of the returned value (it is $\_$ because returned values have primitive type `Bool`).

(This association is defined during the typing of the method body – see below.) The behavioural type of the invocation `x1!m(x2)` is therefore $\mathtt{C!m}([cog : c_1], [cog : c_2]) \to \_$ where $\Gamma(\mathtt{x1}) = [cog : c_1]$ and $\Gamma(\mathtt{x2}) = [cog : c_2]$.

There is a relevant feature that is not expressed by the type $\mathtt{C!m}([cog : c_1], [cog : c_2]) \to \_$. The task corresponding to the invocation `x1!m(x2)` must be assumed to start when the invocation is evaluated and to terminate when the operation `get` on the corresponding future is performed – *cf.* line 17. During this interval, the statements of `x1!m(x2)` may interleave with those of the caller

and those of the other method invocations therein – *cf.* line 16. To have a more precise analysis, we label the type of line 15 with the (logical) time interval in which it has an effect on the computation. Namely we write $\langle \mathbb{b} \rangle^{m \div n}$, where m and n are the starting and the ending interval points, respectively. Our type system increments logical timestamps in correspondence of

1. cog creations,
2. cog migrations,
3. and synchronisation points (`get` operations).

For example, the lines 15–20 of the code in Table 1 have associated timestamps

```
15  Fut<Bool> f1 = x1!m(x2); // timestamp: n
16  Fut<Bool> f2 = x2!m(x3); // timestamp: n
17  Bool b1 = f1.get; // timestamp: n
18  Fut<Bool> f3 = x3!m(x2); // timestamp: n + 1
19  Bool b2 = f2.get; // timestamp: n + 1
20  Bool b3 = f3.get; // timestamp: n + 2
```

As a consequence, the behavioural type of the above code is

$$\langle \texttt{C!m}(\mathbb{r}_1, \mathbb{r}_2) \rightarrow \_ \rangle^{n \div n} \,[]\, \langle \texttt{C!m}(\mathbb{r}_2, \mathbb{r}_3) \rightarrow \_ \rangle^{n \div n+1} \,[]\, \langle \texttt{C!m}(\mathbb{r}_3, \mathbb{r}_2) \rightarrow \_ \rangle^{n+1 \div n+2}$$

where $\mathbb{r}_1$, $\mathbb{r}_2$ and $\mathbb{r}_3$ are the record types of the objects `x1`, `x2`, and `x3`, respectively. As we will see in Section 4, this will impact on the analysis by letting us to consider all the possible computations.

The syntax of behavioural types $\mathbb{b}$ is defined in Table 3. Apart those types that have been already discussed, $\mathbb{b} + \mathbb{b}'$ defines the abstract behaviour of conditionals, $\mathbb{b} \,[]\, \mathbb{b}'$ corresponds to a juxtaposition of behavioural types, and $\langle \mathbb{b} \rangle^{m \div n}$ defines a behavioural type $\mathbb{b}$ to be executed in the interval $m \div n$. It is worth to notice that it is the combination of intervals that models the sequential and the parallel composition: two disjoint intervals specify two subsequent actions, while two overlapping intervals specify two (possibly) parallel actions. This complies with `dcABS` semantics where parallelism is not explicit in the syntax, but it is generated by the (asynchronous) invocations of methods.

We next discuss the association of a method behavioural type to a method declaration. To this aim, let us consider lines 3-5 of the code in Table 1:

```
3   Bool m (C x) {
4     if (@this != @x) moveto @x else moveto d1;
5     return true; }
```

The behaviour of `m` in `C` is given by $(\mathbb{r}, \mathbb{r}')\ \{\mathbb{b}_{\texttt{m}}\} \rightarrow \_$, where $\mathbb{r}$ and $\mathbb{r}'$ are the future records of the receiver of the method and of the argument, respectively, $\mathbb{b}_{\texttt{m}}$ is the type of the body and $\_$ is the future record of the returned boolean value. The records $\mathbb{r}$ and $\mathbb{r}'$ are formal parameters of `m`. Therefore, it is always the case that cog and record names in $\mathbb{r}$ and $\mathbb{r}'$ do occur linearly and *bind* the occurrences of names in $\mathbb{b}_{\texttt{m}}$. It is worth to notice that cog names occurring in $\mathbb{b}_{\texttt{m}}$ may be *not bound*. These *free names* correspond to `new cog` instructions.

In the case of `m` in `C`, its type is:

$$([cog : c], [cog : c'])\{\langle c \mapsto c' \rangle^{1 \div 1} + \langle c \mapsto \texttt{d1} \rangle^{1 \div 1}\} \rightarrow \ \_ \ .$$

The behavioural type for the the main statement of Table 1 is:

$$\begin{array}{l} \langle c_1 \mapsto \mathit{start} \rangle^{1 \div 1} \ [] \ \langle \mathit{start} \mapsto \texttt{d1} \rangle^{2 \div 2} \\ [] \ \langle c_2 \mapsto \mathit{start} \rangle^{3 \div 3} \ [] \ \langle \mathit{start} \mapsto \texttt{d2} \rangle^{4 \div 4} \\ [] \ \langle c_3 \mapsto \mathit{start} \rangle^{5 \div 5} \ [] \ \langle \mathit{start} \mapsto \texttt{d3} \rangle^{6 \div 6} \\ [] \ \langle \texttt{C!m}([cog : c_1], [cog : c_2]) \rightarrow \_ \rangle^{7 \div 7} \\ [] \ \langle \texttt{C!m}([cog : c_2], [cog : c_3]) \rightarrow \_ \rangle^{7 \div 8} \\ [] \ \langle \texttt{C!m}([cog : c_3], [cog : c_2]) \rightarrow \_ \rangle^{8 \div 9}. \end{array}$$

We conclude this section with the typing of the code in Table 2. Method `move` in `D` has type:

$$([cog : c]) \ \{\langle c \mapsto \texttt{d1} \rangle^{1 \div 1}\} \rightarrow \_$$

Method `multi_create` in `D` has type:

$$\begin{array}{l} ([cog : c], \_) \ \{ \\ \quad \texttt{0} \ + \\ \qquad \langle c' \mapsto c \rangle^{1 \div 1} \ [] \ \langle \texttt{D!multi\_create}([cog : c'], \_) \rightarrow \_ \rangle^{2 \div 2} \\ \qquad [] \langle \texttt{D!move}([cog : c']) \rightarrow \_ \rangle^{3 \div 3} \\ \quad \} \ \rightarrow \ \_ \end{array} \tag{2}$$

We notice that the `then`-branch is typed with `0`. In fact, it does not affect the method behaviour since it does not contain any deployment information nor method invocation.

4 Analysis of behavioural types

The analysis of behavioural types defined in Section 3 highlights the trend of cog numbers running in each deployment component over a period of (logical) time. More specifically, behavioural types are used to compute the abstract states of a system that record the deployment of cogs with respect to components. The component load is then obtained by projecting out the number of cogs in a state, which can be visualised by means of a standard graphic plotter program.

A primary item of this programme is the definition of the semantics of behavioural types. To this aim we use *deployment environments* Σ that map cog names to sets of deployment components. For example $[\mathit{start} \mapsto \{\texttt{d0}\}]$ is the *initial* deployment environment. Behavioural types' semantics is defined by means of a transition system where states are triples $\big(\Sigma,\ \mathbb{b},\ n\big)$ and transitions $\big(\Sigma,\ \mathbb{b},\ n\big) \stackrel{m \div m'}{\longrightarrow} \big(\Sigma',\ \mathbb{b}',\ n'\big)$ are defined inductively according to the structure

of $\mathbb{b}$. The basic rules of the transition relation are

(MoveTo-c)
$$\big(\Sigma,\ \langle c \mapsto c'\rangle^{m \div m},\ n\big) \xrightarrow{m \div m} \big(\Sigma[c \mapsto \Sigma(c')],\ \mathtt{0},\ max(m,n)\big)$$

(MoveTo-d)
$$\big(\Sigma,\ \langle c \mapsto \mathtt{d}\rangle^{m \div m},\ n\big) \xrightarrow{m \div m} \big(\Sigma[c \mapsto \{\mathtt{d}\}],\ \mathtt{0},\ max(m,n)\big)$$

(Invk)
$$\frac{\mathtt{C.m} = (\overline{\mathbb{r}})\{\mathbb{b}_{\mathtt{m}}\}\mathbb{r}' \qquad var(\mathbb{b}_{\mathtt{m}}) \setminus var(\overline{\mathbb{r}}, \mathbb{r}') = \overline{c} \qquad \overline{c'} \text{ are fresh} \qquad \mathbb{b}_{\mathtt{m}}[\overline{c'}/\overline{c}][\overline{\mathbb{s}}, \mathbb{s}'/\overline{\mathbb{r}}, \mathbb{r}'] = \mathbb{b}'}{\big(\Sigma, \langle \mathtt{C!m}(\overline{\mathbb{s}}) \to \mathbb{s}'\rangle^{m \div m'},\ n\big) \xrightarrow{m \div m'} \big(\Sigma,\ \langle \mathbb{b}'\rangle^{m \div m'},\ max(m,n)\big)}$$

The rules (MoveTo-c) and (MoveTo-d) update the deployment environment and return a null behavioural type. Rule (Invk) deals with method invocations and, apart from instantiating the formal parameters with the actual ones, it creates fresh cog names that correspond to the `new cog` operations in the method body. The inductive rules (that are omitted in this paper) lift the above transitions to structured behavioural types. In particular, let $m \div n \preceq m' \div n'$ if and only if $n < m'$ ($\preceq$ is a partial order). The rule for $\mathbb{b}_1 \,[]\, \cdots \,[]\, \mathbb{b}_k$ enables a transition $\xrightarrow{m \div n}$ provided $m \div n$ is $\preceq$-*minimal* in the set of transitions of $\mathbb{b}_1, \cdots, \mathbb{b}_k$.

In order to illustrate the operational semantics of behavioural types we discuss the transitions of the type of the main statement in Table 1:

$$\begin{aligned}\mathbb{b}_0 \ = \quad & \langle c_1 \mapsto \mathit{start}\rangle^{1 \div 1} \ [] \ \langle \mathit{start} \mapsto \mathtt{d1}\rangle^{2 \div 2} \\ [] \ & \langle c_2 \mapsto \mathit{start}\rangle^{3 \div 3} \ [] \ \langle \mathit{start} \mapsto \mathtt{d2}\rangle^{4 \div 4} \\ [] \ & \langle c_3 \mapsto \mathit{start}\rangle^{5 \div 5} \ [] \ \langle \mathit{start} \mapsto \mathtt{d3}\rangle^{6 \div 6} \\ [] \ & \langle \mathtt{C!m}([\mathit{cog} : c_1], [\mathit{cog} : c_2]) \to \_\rangle^{7 \div 7} \\ [] \ & \langle \mathtt{C!m}([\mathit{cog} : c_2], [\mathit{cog} : c_3]) \to \_\rangle^{7 \div 8} \\ [] \ & \langle \mathtt{C!m}([\mathit{cog} : c_3], [\mathit{cog} : c_2]) \to \_\rangle^{8 \div 9}.\end{aligned}$$

Let $\Sigma_0 = [\mathit{start} \mapsto \mathtt{d0}]$. According to the semantics of behavioural types, we have

$$\begin{aligned}\big(\Sigma_0\ ,\ \mathbb{b}_0,\ 0\big) &\xrightarrow{1 \div 1} \big(\Sigma_1\ ,\ \mathbb{b}_1,\ 1\big) \xrightarrow{2 \div 2} \big(\Sigma_2\ ,\ \mathbb{b}_2,\ 2\big) \xrightarrow{3 \div 3} \big(\Sigma_3\ ,\ \mathbb{b}_3,\ 3\big) \xrightarrow{4 \div 4} \big(\Sigma_4\ ,\ \mathbb{b}_4,\ 4\big) \\ &\xrightarrow{5 \div 5} \big(\Sigma_5\ ,\ \mathbb{b}_5,\ 5\big) \xrightarrow{6 \div 6} \big(\Sigma_6\ ,\ \mathbb{b}_6,\ 6\big)\end{aligned}$$

where, at each step $1 \le i \le 6$, the type that is evaluated is the one with interval $i \div i$, the deployment environment Σ_6 is $[\mathit{start} \mapsto \{\mathtt{d3}\},\ c_1 \mapsto \{\mathtt{d0}\},\ c_2 \mapsto \{\mathtt{d1}\},\ c_3 \mapsto \{\mathtt{d2}\}]$, and the type $\mathbb{b}_6$ is $\langle \mathtt{C!m}([\mathit{cog} : c_1], [\mathit{cog} : c_2]) \to \_\rangle^{7 \div 7}$ [] $\langle \mathtt{C!m}([\mathit{cog} : c_2], [\mathit{cog} : c_3]) \to \_\rangle^{7 \div 8}$ [] $\langle \mathtt{C!m}([\mathit{cog} : c_3], [\mathit{cog} : c_2]) \to \_\rangle^{8 \div 9}$.

In Figure 1 we have drawn the computations starting at $\big(\Sigma_6,\ \mathbb{b}_6,\ 6\big)$. Here we discuss the rightmost computation. In $\big(\Sigma_6,\ \mathbb{b}_6,\ 6\big)$, the two transitions that are possible are the method invocations with intervals $7 \div 7$ and $7 \div 8$. We perform the one with interval $7 \div 8$ and, by rule (Invk), we get $\big(\Sigma_6,\ \mathbb{b}_8,\ 7\big)$, where $\mathbb{b}_8 = \langle \mathtt{C!m}([\mathit{cog} : c_1], [\mathit{cog} : c_2]) \to \_\rangle^{7 \div 7}$ [] $\langle \langle c_2 \mapsto c_3\rangle^{1 \div 1} + \langle c_2 \mapsto \mathtt{d1}\rangle^{1 \div 1}\rangle^{7 \div 8}$ [] $\langle \mathtt{C!m}([\mathit{cog} : c_3], [\mathit{cog} : c_2]) \to \_\rangle^{8 \div 9}$.

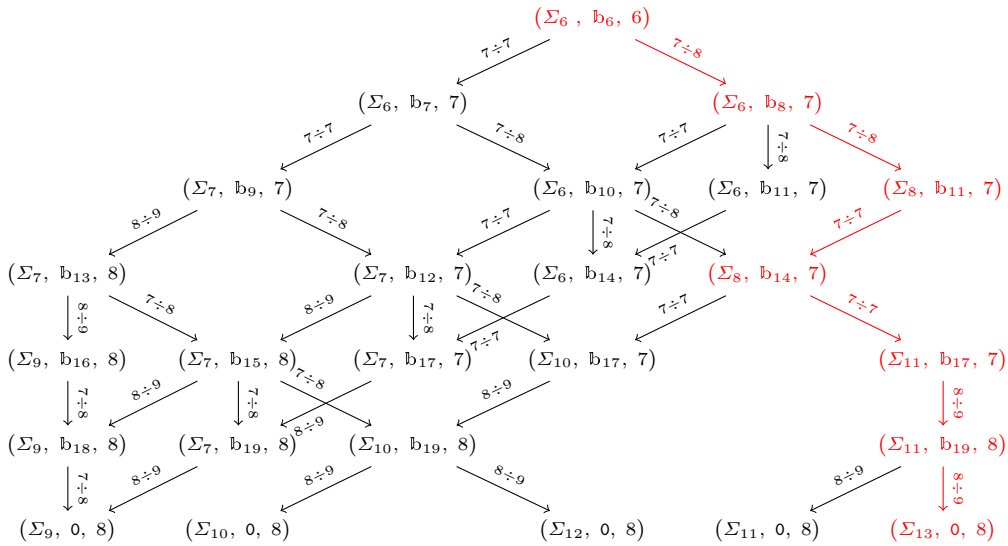


Fig. 1. An example of transition system of behavioural types

At this point there are three options: the method invocation with interval $7 \div 7$ or the evaluation of either $\langle c_2 \mapsto c_3 \rangle^{1 \div 1}$ or $\langle c_2 \mapsto \mathtt{d1} \rangle^{1 \div 1}$, both with interval $7 \div 8$ because underneath a $\langle \cdot \rangle^{7 \div 8}$ context.

By evaluating $\langle c_2 \mapsto c_3 \rangle^{1 \div 1}$, one obtains $\big(\Sigma_8,\ \mathbb{b}_{11},\ 7\big)$, where Σ_8 is $[start \mapsto \{\mathtt{d3}\}, c_1 \mapsto \{\mathtt{d0}\}, c_2 \mapsto \{\mathtt{d2}\}, c_3 \mapsto \{\mathtt{d2}\}]$. and $\mathbb{b}_{11}$ is $\langle \mathtt{C!m}([cog : c_1], [cog : c_2]) \to \_ \rangle^{6 \div 6} \,[]\, \langle \mathtt{C!m}([cog : c_3], [cog : c_2]) \to \_ \rangle^{7 \div 8}$.

In $\big(\Sigma_8,\ \mathbb{b}_{11},\ 7\big)$ only one transition is possible: the method invocation with interval $7 \div 7$. Therefore one has $\big(\Sigma_8,\ \mathbb{b}_{14},\ 7\big)$, where $\mathbb{b}_{14} = \langle \langle c_1 \mapsto c_2 \rangle^{1 \div 1} + \langle c_1 \mapsto \mathtt{d1} \rangle^{1 \div 1} \rangle^{7 \div 7} \,[]\, \langle \mathtt{C!m}([cog : \mathtt{c_3}], [cog : \mathtt{c_2}]) \to \_ \rangle^{8 \div 9}$.

In the state $\big(\Sigma_8,\ \mathbb{b}_{14},\ 7\big)$ the possible transitions are those of the type $\langle \langle c_1 \mapsto c_2 \rangle^{1 \div 1} + \langle c_1 \mapsto \mathtt{d1} \rangle^{1 \div 1} \rangle^{7 \div 7}$. By letting $\langle c_1 \mapsto \mathtt{d1} \rangle^{1 \div 1}$ move, one has $\big(\Sigma_{11},\ \mathbb{b}_{17},\ 7\big)$, where $\Sigma_{11} = [start \mapsto \{\mathtt{d3}\}, c_1 \mapsto \{\mathtt{d1}\}, c_2 \mapsto \{\mathtt{d2}\}, c_3 \mapsto \{\mathtt{d2}\}]$ and $\mathbb{b}_{17} = \langle \mathtt{C!m}([cog : c_3], [cog : c_2]) \to \_ \rangle^{8 \div 9}$. Finally, by performing two transitions labelled $8 \div 9$, one first unfolds the method invocation and then evaluates the corresponding body $\langle \langle c_3 \mapsto c_2 \rangle^{1 \div 1} + \langle c_3 \mapsto \mathtt{d1} \rangle^{1 \div 1} \rangle^{8 \div 9}$. By letting $\langle c_3 \mapsto \mathtt{d1} \rangle^{1 \div 1}$ move, the computation terminates with a deployment environment $\Sigma_{13} = [start \mapsto \{\mathtt{d3}\}, c_1 \mapsto \{\mathtt{d1}\}, c_2 \mapsto \{\mathtt{d2}\}, c_3 \mapsto \{\mathtt{d1}\}]$.

Given a transition system $\mathcal{T}$ as the one illustrated in Figure 1, it is possible to compute the *abstract trace*, *i.e.* the sequence $\Sigma(0) \cdot \Sigma(1) \cdot \Sigma(2) \cdots$ where $\Sigma(i)$ is the deployment environment

$$\Sigma(i) : c \mapsto \cup \{\Sigma(c) \quad | \quad \text{there is } \mathbb{b} \text{ such that } \big(\Sigma,\ \mathbb{b},\ i\big) \in \mathcal{T}\}\ .$$

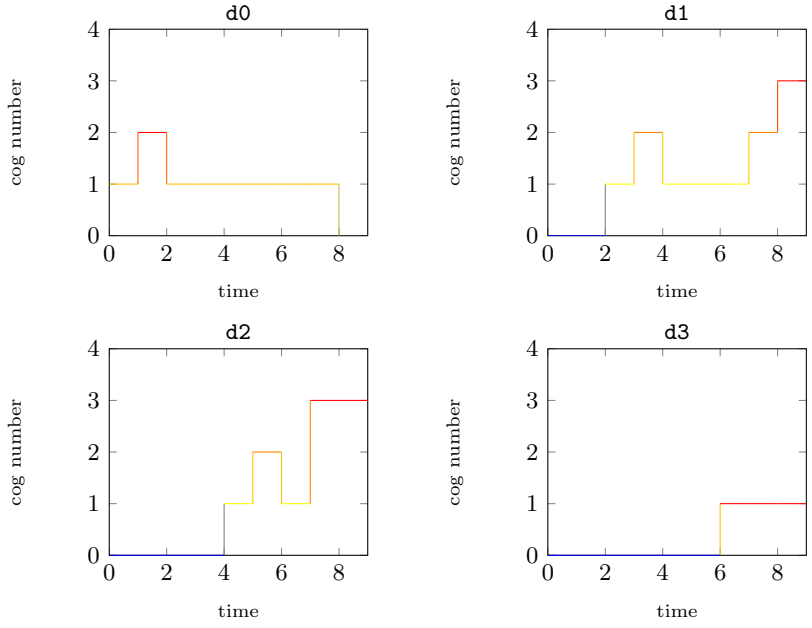
For example, letting the deployment environments of Figure 1 be

$$\begin{aligned}
\Sigma_6 &= [\mathit{start} \mapsto \{\texttt{d3}\},\ c_1 \mapsto \{\texttt{d0}\},\ c_2 \mapsto \{\texttt{d1}\},\ c_3 \mapsto \{\texttt{d2}\}] \\
\Sigma_7 &= [\mathit{start} \mapsto \{\texttt{d3}\},\ c_1 \mapsto \{\texttt{d1}\},\ c_2 \mapsto \{\texttt{d1}\},\ c_3 \mapsto \{\texttt{d2}\}] \\
\Sigma_8 &= [\mathit{start} \mapsto \{\texttt{d3}\},\ c_1 \mapsto \{\texttt{d0}\},\ c_2 \mapsto \{\texttt{d2}\},\ c_3 \mapsto \{\texttt{d2}\}] \\
\Sigma_9 &= [\mathit{start} \mapsto \{\texttt{d3}\},\ c_1 \mapsto \{\texttt{d1}\},\ c_2 \mapsto \{\texttt{d1}\},\ c_3 \mapsto \{\texttt{d1}\}] \\
\Sigma_{10} &= [\mathit{start} \mapsto \{\texttt{d3}\},\ c_1 \mapsto \{\texttt{d1}\},\ c_2 \mapsto \{\texttt{d2}\},\ c_3 \mapsto \{\texttt{d2}\}] \\
\Sigma_{11} &= [\mathit{start} \mapsto \{\texttt{d3}\},\ c_1 \mapsto \{\texttt{d2}\},\ c_2 \mapsto \{\texttt{d2}\},\ c_3 \mapsto \{\texttt{d2}\}] \\
\Sigma_{12} &= [\mathit{start} \mapsto \{\texttt{d3}\},\ c_1 \mapsto \{\texttt{d1}\},\ c_2 \mapsto \{\texttt{d2}\},\ c_3 \mapsto \{\texttt{d1}\}] \\
\Sigma_{13} &= [\mathit{start} \mapsto \{\texttt{d3}\},\ c_1 \mapsto \{\texttt{d2}\},\ c_2 \mapsto \{\texttt{d2}\},\ c_3 \mapsto \{\texttt{d1}\}]
\end{aligned}$$

we can compute the cog trend for each deployment component. Let $\Sigma(i)|_{\texttt{d}} \stackrel{def}{=} \{c \mid \texttt{d} \in \Sigma(i)(c)\}$. Then

$\Sigma(i)\|_{\texttt{d}}$	d0	d1	d2	d3
$\Sigma(0)$	*start*	$\varnothing$	$\varnothing$	$\varnothing$
$\Sigma(1)$	c_1, *start*	$\varnothing$	$\varnothing$	$\varnothing$
$\Sigma(2)$	c_1	*start*	$\varnothing$	$\varnothing$
$\Sigma(3)$	c_1	c_2, *start*	$\varnothing$	$\varnothing$
$\Sigma(4)$	c_1	c_2	*start*	$\varnothing$
$\Sigma(5)$	c_1	c_2	c_3, *start*	$\varnothing$
$\Sigma(6)$	c_1	c_2	c_3	*start*
$\Sigma(7)$	c_1	c_1, c_2	c_1, c_2, c_3	*start*
$\Sigma(8)$	$\varnothing$	c_1, c_2, c_3	c_1, c_2, c_3	*start*

Graphically (note that d0 starts at level 1, since at the beginning it contains the "*start*" cog):



We conclude our overview by discussing the issue of recursive invocation. To this aim, consider the type (2) of the method `multi_create` in Table 2 and the main statement

```
D x = new cog D( ); Fut<Bool> f = x!multi_create(4); Bool b = f.get;
```

whose type is:

$$\mathbb{b}_0^r = \langle c_1^r \mapsto \mathit{start} \rangle^{1 \div 1} [\!] \langle \texttt{D!multi\_create}([\mathit{cog} : c_1^r], \_) \to \_ \rangle^{2 \div 2}$$

Being `d0` and `d1` the two declared deployment components, we obtain the following computation:

$$\begin{array}{l}
\big([\mathit{start} \mapsto \{\texttt{d0}\}]\ ,\ \mathbb{b}_0^r,\ 0\big) \\
\quad \xrightarrow{1 \div 1} \big([\mathit{start} \mapsto \{\texttt{d0}\}, c_1^r \mapsto \{\texttt{d0}\}]\ ,\ \mathbb{b}_1^r,\ 1\big) \\
\quad \xrightarrow{2 \div 2} \quad \xrightarrow{2 \div 2} \big([\mathit{start} \mapsto \{\texttt{d0}\}, c_1^r \mapsto \{\texttt{d0}\}, c_2^r \mapsto \{\texttt{d0}\}]\ ,\ \mathbb{b}_3^r,\ 2\big) \\
\quad \xrightarrow{2 \div 2} \cdots \xrightarrow{2 \div 2} \big([\mathit{start} \mapsto \{\texttt{d0}\}, c_1^r \mapsto \{\texttt{d0}\}, c_2^r \mapsto \{\texttt{d0}\}, \cdots, c_{n-2}^r \mapsto \{\texttt{d0}\}]\ ,\ \mathbb{b}_n^r,\ 2\big) \\
\quad \xrightarrow{2 \div 2} \quad \xrightarrow{2 \div 2} \big([\mathit{start} \mapsto \{\texttt{d0}\}, c_1^r \mapsto \{\texttt{d0}\}, c_2^r \mapsto \{\texttt{d0}\}, \cdots, c_{n-2}^r \mapsto \{\texttt{d1}\}]\ ,\ \mathbb{b}_{n+2}^r,\ 2\big) \\
\quad \xrightarrow{2 \div 2} \cdots \xrightarrow{2 \div 2} \big([\mathit{start} \mapsto \{\texttt{d0}\}, c_1^r \mapsto \{\texttt{d0}\}, c_2^r \mapsto \{\texttt{d1}\}, \cdots, c_{n-2}^r \mapsto \{\texttt{d1}\}]\ ,\ 0,\ 2\big)
\end{array}$$

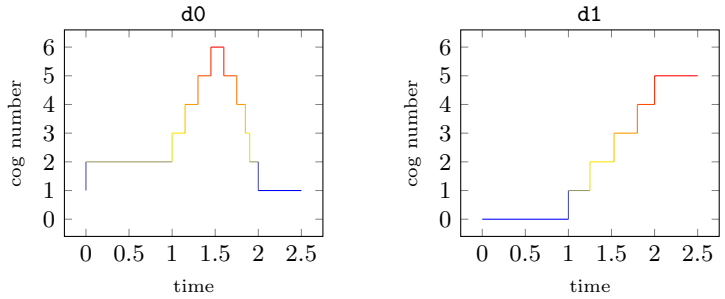
where

$$\begin{array}{lll}
\mathbb{b}_1^r & = & \langle \texttt{D!multi_create}([\mathit{cog} : c_1^r], _) \to _ \rangle^{2 \div 2} \\
\mathbb{b}_2^r & = & \langle \quad \langle c_2^r \mapsto c_1^r \rangle^{1 \div 1} [\!] \langle \texttt{D!multi_create}([\mathit{cog} : c_2^r], _) \to _ \rangle^{2 \div 2} \\
 & & \qquad [\!] \langle \texttt{D!move}([\mathit{cog} : c_2^r]) \to _ \rangle^{3 \div 3} \quad \rangle^{2 \div 2} \\
\mathbb{b}_3^r & = & \langle\ \langle \texttt{D!multi_create}([\mathit{cog} : c_2^r], _) \to _ \rangle^{2 \div 2} [\!] \langle \texttt{D!move}([\mathit{cog} : c_2^r]) \to _ \rangle^{3 \div 3} \\
 & & \rangle^{2 \div 2} \\
\cdots & & \\
\mathbb{b}_n^r & = & \langle\ \langle \langle \cdots \langle \langle \texttt{D!move}([\mathit{cog} : c_{(n-2)}^r]) \to _ \rangle^{3 \div 3} \rangle^{2 \div 2} \\
 & & \quad [\!] \langle \langle \texttt{D!move}([\mathit{cog} : c_{(n-3)}^r]) \to _ \rangle^{3 \div 3} \cdots \rangle^{2 \div 2} \\
 & & \quad [\!] \langle \texttt{D!move}([\mathit{cog} : c_3^r]) \to _ \rangle^{3 \div 3} \rangle^{2 \div 2} [\!] \langle \texttt{D!move}([\mathit{cog} : c_2^r]) \to _ \rangle^{3 \div 3} \\
 & & \rangle^{2 \div 2} \\
\mathbb{b}_{n+2}^r & = & \langle\ \langle \langle \cdots \langle \texttt{D!move}([\mathit{cog} : c_{(n-3)}^r]) \to _ \rangle^{3 \div 3} \cdots \rangle^{2 \div 2} \\
 & & \quad [\!] \langle \texttt{D!move}([\mathit{cog} : c_3^r]) \to _ \rangle^{3 \div 3} \rangle^{2 \div 2} [\!] \langle \texttt{D!move}([\mathit{cog} : c_2^r]) \to _ \rangle^{3 \div 3} \\
 & & \rangle^{2 \div 2}
\end{array}$$

We observe the following two facts: first, every transition, except the initial one, is at logical timestamp 2, because only the outermost interval is observable, while the nested intervals are only relevant to specify the order of events at the same level of nesting; second, in case of recursion the specified behaviour is potentially infinite and parameterised by the number n of transitions, which depends on the number of recursive invocations.

In visualising the results of the analysis, these two aspects pose some questions: the first one may lead us to flatten all the events at timestamp 2 as they happened in parallel, while if observing carefully the computation we notice the events follow a strict sequence; the second one may make it difficult to graphically represent the unbounded behaviour. To address the first point, we don't

simply rely on the label of the transition to recognise the state of the computation, but at each interval the visualiser performs a sort of *zoom in*, so to magnify the nested behaviour. The result is the sequentialised behaviour depicted below. To address the second one, we just approximate the behaviour by letting *at most* n nested recursive invocations. The corresponding graphs are as follows, fixing $n = 8$, (note that `d0` starts at level 1, since at the beginning it contains the "*start*" cog):



In this case, the recursive behaviour corresponds to a pick of deployed cogs in the interval $2 \div 3$. This pick grows according to the value of n. The interesting property we may grasp from the graphs for `d0` is that, the upward pick in the interval $1 \div 2$ corresponds to a downward pick in the same interval of the same length. This is due to the property that, for each increment in that interval, there is a decrement, thus leaving unchanged the number of cogs in `d0` (which is 2). A different behaviour is manifested by the graph of the component `d1`. In this case, there is a growing increment of deployed cogs according to the increasing of n. The rightmost function lets us derive that the deployment component `d1` may become critical as the computation progresses.

5 Related work

Resource analysis has been extensively studied in the literature and several methods have been proposed, ranging from static analyses (data-flow analysis and type systems) to model checking. We discuss in this section a number of related techniques and the differences with the one proposed in this paper.

A well-known technique is the so-called *resource-aware programming* [21] that allows users to monitor the resources consumed by their programs and to express policies for the management of such resources in the programs. Resource-aware programming is also available for mainstream languages, such as Java [4]. Our typing system may integrate resource-aware programming by providing static-time feedbacks about the correctness of the management, such as full-coverage of cases, correctness of the policies, etc.

Other techniques address resource management in embedded systems and mostly use performance analysis on models that are either process algebra [18], or Petri Nets [23], or various types of automata [24]. It is also worth to remind that similar techniques have been defined for web services and business processes [6,

22]. Usually, all these approaches are *invasive* because they oblige programmers to declare the cost of transitions in terms of time or in terms of a single resource. On the contrary, our technique does not assign any commitment to programmers, which may be completely unaware of resources and their management.

In [1] a quantified analysis targets `ABS` programs and returns informations about the different kinds of nodes that compose the system, how many instances of each kind exist, and node interactions. A resource analysis infers upper bounds to the number of concrete instances that the nodes and arcs represent. (The analysis in [1] does not explicitly support deployment components and cog migration; however we believe that this integration is possible.) An important difference of this analysis with respect to our contribution is that our behavioural types are intended to represent a part of SLA that may be validated in a formal way and that support compositional analysis. It is not clear if these correspondence with SLA is also possible for the models of [1].

A type inference technique for resource analysis has been developed in [11,12]. They study the problem of worst-case heap usage in functional and (sequential) object-oriented languages and their tool returns functions on the size of inputs of every method that highlight the heap consumption. On the contrary, our technique returns upper bounds *disregarding* input sizes. However, we think it is possible to extend our types to enable a transition system model that support the expressivity of [12] (our current analysis of behavioural types is preliminary and must be considered as a proof-of-concept). In these regards, we plan to explore the adoption of behavioural types that depends [3] on the input data of conditions in if-statements. We observe, anyway, that the generalisation of the results in [11, 12] to a concurrent setting has not been investigated.

Kobayashi, Suenaga and Wischik develop a technique that is very close to the one in this paper [16]. In particular, they extend pi-calculus with primitives for creating and using resources and verify whether a program conforms with resource usage declarations (that may be also automatically inferred). A difference between their technique and the one in this paper is that here the resource analysis is performed *ex-post* by resorting to abstract transition systems of behavioural types, while in [16] the analysis is done during the type checking(/inference). As discussed in [9], our technique is in principle more powerful than those verifying resource usage during the checking/inference of types.

6 Conclusions

This work is a preliminary theoretical study about the analysis of resource deployments by means of type systems. Our types are lightweight abstract descriptions of behaviours that retain resource informations and admit type inference.

The analysis of behavioural types that has been discussed in Section 4 is very preliminary. In fact, in Example 2, the resource analysis depends on the input value of the method `multi_create`. In these cases, a reasonable output of the analysis is a formula that defines the cog-load of deployment components according to the actual value in input. As discussed in Section 5, we intend to

investigate more convenient behavioural type analyses, possibly by using more expressive types, such as dependent ones [3].

One obvious research direction is to apply our technique for defining an inference system for resource deployment in programming languages, such as `ABS` or `core ABS`, and prototyping it with a tool for displaying the load of deployment components. The programme is similar to the one developed for deadlock analysis [10]. The next step is then the experiment of the prototype on real programs in order to have assessments about its performance and precision.

We also intend to study the range of application of type system techniques when resources are either cloud virtual machines, or CPU, or memory, or bandwidth. The intent is to replace/complement the simulation techniques used in [14, 15] with static analysis techniques based on types.

References

[1] E. Albert, J. Correas, G. Puebla, and G. Román-Díez. Quantified abstractions of distributed systems. In *iFM'13*, volume 7940 of *LNCS*, pages 285–300. Springer-Verlag, 2013.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[3] A. Bove and P. Dybjer. Dependent types at work. In *LerNet ALFA Summer School*, volume 5520 of *Lecture Notes in Computer Science*, pages 57–99. Springer, 2008.

[4] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of OOPSLA*, pages 21–35, 1998.

[5] F. de Boer, D. Clarke, and E. Johnsen. A complete guide to the future. In *Progr. Lang. and Systems*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.

[6] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. S. Rosenblum, and S. Uchitel. Model checking service compositions under resource constraints. In *Proc. 6th of the European Software Engineering Conf. and the Symposium on Foundations of Software Engineering*, pages 225–234. ACM, 2007.

[7] S. Gay and M. Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

[8] E. Giachino, C. A. Grazia, C. Laneve, M. Lienhardt, and P. Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice. In *iFM'13*, volume 7940 of *LNCS*, pages 394–411. Springer-Verlag, 2013.

[9] E. Giachino, N. Kobayashi, and C. Laneve. Deadlock analysis of unbounded process networks. In *Proceedings of Concur'2014*, LNCS. Springer-Verlag, 2014.

[10] E. Giachino, C. Laneve, and M. Lienhardt. A Framework for Deadlock Detection in ABS. *Software and Systems Modeling.*, 2014. To Appear.

[11] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.

[12] M. Hofmann and D. Rodriguez. Automatic type inference for amortised heap-space analysis. In *22nd European Symposium on Programming, ESOP 2013*, volume 7792 of *Lecture Notes in Computer Science*, pages 593–613. Springer, 2013.

[13] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. of FMCO 2010*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.

[14] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In *Proc. of ICFEM'10*, volume 6447 of *Lecture Notes in Computer Science*, pages 646–661. Springer-Verlag, 2010.

[15] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In *Proc. of FoVeOOS'10*, volume 6528 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2011.

[16] N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the pi-calculus. *Logical Methods in Computer Science*, 2(3), 2006.

[17] C. Laneve and L. Padovani. The *must* preorder revisited. In *Proc. CONCUR 2007*, volume 4703 of *LNCS*, pages 212–225. Springer, 2007.

[18] G. Lüttgen and W. Vogler. Bisimulation on speed: A unified approach. *Theoretical Computer Science*, 360(1–3):209–227, 2006.

[19] R. Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[20] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, ii. *Inf. and Comput.*, 100:41–77, 1992.

[21] L. Moreau and C. Queinnec. Resource aware programming. *ACM Trans. Program. Lang. Syst.*, 27(3):441–476, 2005.

[22] M. Netjes, W. M. van der Aalst, and H. A. Reijers. Analysis of resource-constrained processes with Colored Petri Nets. In *Proceedings of the Sixth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2005)*, volume 576 of *DAIMI*. University of Aarhus, 2005.

[23] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice Petri nets. In *Proc. 36th ACM/IEEE Design Automation Conference (DAC'99)*, pages 805–810. ACM, 1999.

[24] A. Vulgarakis and C. C. Seceleanu. Embedded systems resources: Views on modeling and analysis. In *Proc. 32nd IEEE Intl. Computer Software and Applications Conference (COMPSAC'08)*, pages 1321–1328. IEEE Computer Society, 2008.