Project N°:	**FP7-610582**
Project Acronym:	**ENVISAGE**
Project Title:	**Engineering Virtualized Services**
Instrument:	**Collaborative Project**
Scheme:	**Information & Communication Technologies**

Deliverable D2.1
Behavioural Interfaces for Virtualized Services

Date of document: T12

Start date of the project: **1st October 2013** Duration: **36 months**

Organisation name of lead contractor for this deliverable: **BOL**

Final version

STREP Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Behavioural Interfaces for Virtualized Services

This document summarises deliverable D2.1 of project FP7-610582 (Envisage), a Collaborative Project supported by the 7th Framework Programme of the EC within the Information & Communication Technologies scheme. Full information on this project is available online at `http://www.envisage-project.eu`.

This report studies and develops abstract descriptions of components of virtualized systems that are amenable to automatic verification and validation. These abstract descriptions are *behavioral interfaces* for the target language developed in Task T1.1 (`ABS`) and are mostly used to verify the interoperability obligations between the different parties that compose a virtual system.

The following aspects are reported:

1. the decision algorithm for deadlock freedom of behavioural interfaces of virtual systems;
2. the abstract specifications of virtual systems by means of communication histories and definition of their compositions;
3. the analysis of resource deployments in virtual systems by means of type systems.

List of Authors

Crystal Chang Din (TUD)
Elena Giachino (BOL)
Cosimo Laneve (BOL)
Michael Lienhardt (BOL)

Contents

Chapter 1

Introduction

Work package 2 studies and develops abstract descriptions of components of virtualised systems that are amenable to automatic verification and validation (an issue that is considered in Work package 3). According to the Envisage DoW, the Deliverable 2.1

1. studies a notion of behavioural interface for the basic virtualized services as developed in Task T1.1. These interfaces are mostly used to verify the interoperability obligations between the different parties that compose a virtual system;

2. defines the formal assessment of the relationship between the detailed description of a virtualized service as given in Task T1.1 and the behavioural interface as defined in this task;

3. motivates the appropriateness of the behavioural interfaces developed in this task by the needs arising in the initial definition of the Envisage case studies, and possibly refined according to the feedbacks.

The language for virtualized services that is developed in Task T1.1 – `ABS` – is an abstract, executable, object-oriented modelling language with a formal semantics. In `ABS`, method invocations are asynchronous: the caller continues after the invocation and the called code runs on a different task. Tasks are cooperatively scheduled, that is there is a notion of group of objects, called *COG*, and there is at most one active task at each time per COG. The active task explicitly returns the control in order to let other tasks progress. The synchronisation between the caller and the called methods is performed when the result is strictly necessary. Technically, the decoupling of method invocation and the returned value is realised using *future variables* (see [9] and the references in there), which are pointers to values that may be not available yet. Clearly, the access to values of future variables may require waiting for the value to be returned.

As regards the above item 1, the Deliverable 2.1 studies the formalisation of the obligations of components of `ABS` systems that guarantee never-ending waitings for returned values. This property, which turns out to be a consequence of deadlock freedom, is enforced by associating *behavioural interfaces* to method definitions, following the techniques ranging from session types [13] to process contracts [23] and to calculi of processes such as Milner's CCS or pi-calculus [24,25]. These behavioural interfaces are sequences of basic terms recording the method invocations and the synchronisations between calling and called methods.

In Chapters 2 and 3 we have investigated two possible techniques. The one in Chapter 2 defines behavioural interfaces, called *lam*s, that feature recursion and resource creation; therefore their underlying model has infinite states. These interfaces were already studied in the previous European project HATS, where we proposed two analysis techniques [14], which give imprecise, yet sound, answers (in some cases, the techniques may signal false positives). Chapter 2 defines a *decision algorithm* for lams that is based on a fixpoint technique. This algorithm takes a lam and returns a *precise answer*, according to whether the lam manifests a circular dependency (a deadlock) or not. The behavioural interfaces defined in Chapter 3 are based on communication histories. These histories abstractly capture the system state at any point in time [7,8]. Partial correctness properties of a system may thereby be specified by finite initial segments of its communication histories. A history invariant is a predicate over the communication history, which holds for

all finite sequences in the (prefix-closed) set of possible histories, expressing safety properties (such as there is never a blocking call). The chapter defines a framework for compositional reasoning about object systems, establishing an invariant over the global history from invariants over the local histories of each object.

The two techniques described in Chapters 2 and 3 mainly differ in the ways the interfaces are related to the virtualized service as given in Task T1.1 – see **the above item 2**. As regards lams, we have designed an inference system that *automatically extracts behavioural interfaces* from ABS code [18] (this activity that completely redefines the inference system of the EU-funded project HATS [14] lasted during the initial months of Envisage). In this case, a subject reduction theorem guarantees the correctness of the association behavioural interfaces/virtualized services. The inference system will be discussed in Work package 3. As regards communication histories, Chapter 3 defines a proof system for local (class-based) reasoning using Hoare triples, and a rule for composition of object specifications. A soundness and completeness result of the logics with respect to the operational semantics is proved. A crucial property of communication histories and their proof system is that the corresponding Hoare triples can be verified with the KeY theorem prover [4]. A reasoning system developed in KeY is currently being defined in dynamic logic and will be discussed in Work package 3.

Since the behavioural interfaces in Chapters 2 and 3 appear to be related, currently we are investigating the relation between them. The ultimate goal is to bridge the gap between the two techniques by translating the inference system for deadlock analysis developed at BOL into a dynamic logic proof system for KeY and reformulate lams as suitable invariants on communication histories. This work is planned to be done within the scope of Task 3.4, Hybrid Analysis.

As regards the above item 3, all services described in D4.1 by our industrial partners ATB, FRH, and ENG are in general highly concurrent (e.g. crawling with multiple crawler nodes and processes, indexing and serving of data to mobile users). Therefore deadlock handling is absolutely of relevance.

Deliverable D1.1 also introduces a basic notion of *deployment component* and of *computing resource*. The former represents abstract virtual machines in a cloud architecture and the latter one represents any cloud resource that can properly be quantified (for example memory, disk, network, etc). Chapter 4 reports on a preliminary study of a type-based technique for analysing the deployment of resources in ABS. In particular, we design a type system for a dialect of ABS that supports dynamic COG creation and movements between deployment components (which are statically defined). The type of a program expresses the resource deployments over periods of (logical) time. This technique allows the inference of types, in the same style as [18], and the resource load of deployment components can be visualised by means of a standard graphic plotter program. The details of the technique, such as the system for deriving behavioural types automatically and the correctness results, will be reported in the Deliverable D2.2 (Formalization of Service Contracts and SLAs – Initial Report) of the Envisage project.

1.1 List of Papers Comprising Deliverable D2.1

This section lists all the papers that this deliverable comprises, indicates where they were published, and explains how each paper is related to the main text of this deliverable. The deliverable contains either extended abstracts of the papers or the parts that are relevant for the Envisage project. The full papers are made available in the appendix of this deliverable and on the Envisage web site at the url `http://www.envisage-project.eu/` (select "Dissemination"). Direct links are also provided for each paper listed below.

Paper 1: Deadlock analysis of unbounded process networks This paper [15] addresses deadlock detection in concurrent programs that create networks with arbitrary numbers of nodes. This problem is extremely complex and solutions either give imprecise answers or do not scale. To enable the analysis of such programs, the contribution (1) defines an algorithm for detecting deadlocks of a basic model featuring recursion and fresh name generation: the lam programs, and (2) illustrate its relevance by designing a type system for value passing CCS that returns lam programs. As a byproduct of these two techniques, one has an algorithm that is more powerful than previous ones and that can be easily integrated in the current release of

`TyPiCal`, a type-based analyser for pi-calculus. The algorithm (1) has also been integrated in the deadlock analyser of `ABS` [18] and is available online at `http://df4abs.nws.cs.unibo.it`.

The paper was written by Elena Giachino, Naoki Kobayashi and Cosimo Laneve and an extended version was published in the proceedings of CONCUR 2014.

Download the paper at `http://www.cs.unibo.it/~laneve/papers/concur2014.pdf`.

Paper 2: Representation of behavioural interfaces for concurrent objects communicating by asynchronous method calls and futures First-class futures improve the communication efficiency between objects. However, futures are shared entities between objects. In this work [10, 11] we achieve local reasoning inside each class using communication histories that deal with futures and asynchronous method calls. The reasoning system is proved sound and relative complete with respect to the operational semantics.

The paper was written by Crystal Chang Din and Olaf Owe and was published in the Journal of Logical and Algebraic Methods in Programming 2014.

Download the paper at `http://dx.doi.org/10.1016/j.jlamp.2014.03.003`.

Paper 3: Towards the typing of resource deployment This paper [17] is a preliminary study of a type-based technique for analysing the deployments of resources in cloud computing. The type system is targeted to (a dialect of) `ABS` with dynamic resource creations and movements. The technique admits the inference of types and may underlie the optimisation of the costs and consumption of resources.

The paper was written by Elena Giachino and Cosimo Laneve and was published in the proceedings of ISoLA 2014.

Download the paper at `http://www.cs.unibo.it/~laneve/papers/Isola2014.pdf`.

Chapter 2

A decision algorithm for deadlock freedom of behavioural interfaces of virtual systems

2.1 Introduction

Deadlock detection in concurrent programs that create networks with arbitrary numbers of nodes is extremely complex and solutions either give imprecise answers or do not scale. To enable the analysis of such programs, we define an algorithm for detecting deadlocks of a basic model featuring recursion and fresh name generation: the *lam programs*.

As a byproduct of the inference algorithm discussed in [14, 18], we have a deadlock detection algorithm for `ABS` that is more powerful than previous ones and that has been integrated in our analyser available at `http://df4abs.nws.cs.unibo.it`

2.2 A decision algorithm for detecting lam's circularities

Deadlock-freedom of concurrent programs has been largely investigated in the literature [1, 5, 12, 21, 26, 27]. The proposed algorithms automatically detect deadlocks by building graphs of dependencies (a, b) between resources, meaning that the release of a resource referenced by a depends on the release of the resource referenced by b. The absence of cycles in the graphs entails deadlock freedom. When programs have infinite states, in order to ensure termination, current algorithms use finite approximate models that are excerpted from the dependency graphs. The most critical programs are those that create networks with an arbitrary number of nodes.

To illustrate the issue, consider the following pi-calculus-like process that computes the factorial (the notation "." represents the prefix, while "|" represents parallel composition):

```
Fact(n,r,s) =  if n=0 then r?m. s!m
               else  new t in (r?m. t!(m*n)) | Fact(n-1,t,s)
```

Here, `r?m` waits to receive a value for `m` on `r`, and `s!m` sends the value `m` on `s`. The expression **new** `t` **in** `P` creates a fresh communication channel `t` and executes `P`. If the above code is invoked with `r!1 | Fact(n,r,s)`, then there will be a synchronisation between `r!1` and the input `r?m` in the body of `Fact(n,r,s)`. In turn, this may delegate the computation of the factorial to another process in parallel by means of a subsequent synchronisation on a new channel `t`. That is, in order to compute the factorial of `n`, `Fact` builds a network of $n + 1$ nodes, where node `i` takes as input a value `m` and outputs `m*i`. Due to the inability of statically reasoning about unbounded structures, the current analysers usually return false positives when fed with `Fact`. For example, this is the case of `TyPiCal` [21, 22], a tool developed for pi-calculus. In particular, `TyPiCal` fails to recognise that there is no circularity in the dependencies among r, s, and t.

In Chapter A, we develop a technique to enable the deadlock analysis of processes with arbitrary networks of nodes. Instead of reasoning on finite approximations of such processes, we associate them with terms of a

basic recursive model, called *lam* – for *deadLock Analysis Model* –, which collects dependencies and features recursion and dynamic name creation [14, 16]. For example, the lam function corresponding to `Fact` is

$$\texttt{fact}(a_1, a_2, a_3, a_4) = (a_2, a_3) + (\nu\, a_5, a_6)\big((a_2, a_6) \,\&\, \texttt{fact}(a_5, a_6, a_3, a_4)\big)$$

where (a_2, a_3) displays the dependency between the actions `r?m` and `s!m` and (a_2, a_5) the one between `r?m` and `t!(m*n)`, and $(\nu\, a)\mathsf{L}$ defines a new name a whose scope is L. The semantics of `fact` is defined operationally by unfolding the recursive invocations, e.g. by replacing the invocations with the instance of the body where the defined names are replaced by fresh names and the formal parameters are replaced by the actual ones. In particular, the unfolding of $\texttt{fact}(a_1, a_2, a_3, a_4)$ yields the following sequence of abstract states (bound names in the definition of `fact` are replaced by fresh ones in the unfoldings).

$$\begin{aligned}
\texttt{fact}(a_1, a_2, a_3, a_4) &\longrightarrow (a_2, a_3) + \big((a_2, a_6) \,\&\, \texttt{fact}(a_5, a_6, a_3, a_4)\big) \\
&\longrightarrow (a_2, a_3) + (a_2, a_6) \,\&\, (a_6, a_3) + (a_2, a_6) \,\&\, (a_6, a_8) \,\&\, \texttt{fact}(a_7, a_8, a_3, a_4) \\
&\longrightarrow (a_2, a_3) + (a_2, a_6) \,\&\, (a_6, a_3) + (a_2, a_6) \,\&\, (a_6, a_8) \,\&\, (a_8, a_3) \\
&\qquad + (a_2, a_6) \,\&\, (a_6, a_8) \,\&\, (a_8, a_{10}) \,\&\, \texttt{fact}(a_9, a_{10}, a_3, a_4) \\
&\longrightarrow \cdots
\end{aligned}$$

While the model of `fact` is not finite-state, in Chapter A we demonstrate that it is decidable whether the computations of a lam program will ever produce a circular dependency. The algorithm is defined by means of a standard fixpoint technique. In our previous work [14, 16], the decidability was established only for a restricted subset of lams.

Combining the type inference developed for `ABS` in [18], we have an analyser that is powerful enough to detect deadlocks of programs that create networks with arbitrary numbers of processes. It is also worth to notice that the algorithm defined in this chapter has been applied to value passing CCS in [15] and the technique can be easily extended to pi-calculus.

Chapter 3

Representation of Behavioural Interfaces for Concurrent Objects Communicating by Asynchronous Method Calls and Futures

3.1 Introduction

Nowadays many software systems are distributed. However, distributed systems are difficult to analyse especially when we take concurrency, communication and synchronisation mechanisms into account. We present a model which facilitates invariant specification over the locally visible communication history of each distributed component (concurrent object). We model the system by `core ABS`, in which concurrent objects communicates with one another by asynchronous method calls and futures, and present a compositional approach for specification.

3.2 Representation of Behavioural Interfaces using Communication Histories

In Chapter B we present a communication model for concurrent objects communicating by means of asynchronous message passing and futures. We describe the execution of an object by different *communication events* which reflect the *observable* interaction between the object and its environment. The observable behavior of a system is described by communication histories over observable events [6, 19].

3.2.1 Communication Events

Since message passing in ABS is asynchronous, we consider separate events for method invocation, reacting upon a method call, resolving a future, and for fetching the value of a future. Each event is observable to only one object, which is the one that *generates* the event. The events generated by a method call cycle is depicted in Figure 3.1. The object o calls a method m on object o' with input values $\overline{e}$ and where u denotes the future identity. An invocation message is sent from o to o' when the method is invoked. This is reflected by the *invocation event* $\langle o \rightarrow o', u, m, \overline{e}\rangle$ generated by o. An *invocation reaction event* $\langle \twoheadrightarrow o', u, m, \overline{e}\rangle$ is generated by o' once the method starts execution. When the method terminates, the object o' generates the *future event* $\langle \leftarrow\ o', u, e\rangle$. This event reflects that u is resolved with return value e. The *fetching event* $\langle o\ \twoheadleftarrow, u, e\rangle$ is generated by o when o fetches the value of the resolved future. Since future identities may be passed to other objects, e.g, o'', that object may also fetch the future value, reflected by the event $\langle o''\ \twoheadleftarrow, u, e\rangle$, generated by o''. The *object creation event* $\langle o \uparrow o', C, \overline{e}\rangle$ represents object creation, and is generated by o when o creates a fresh object o'.

Definition 3.2.1. (Events) *Let type* Mid *include all method names, and let* Data *be the supertype of*

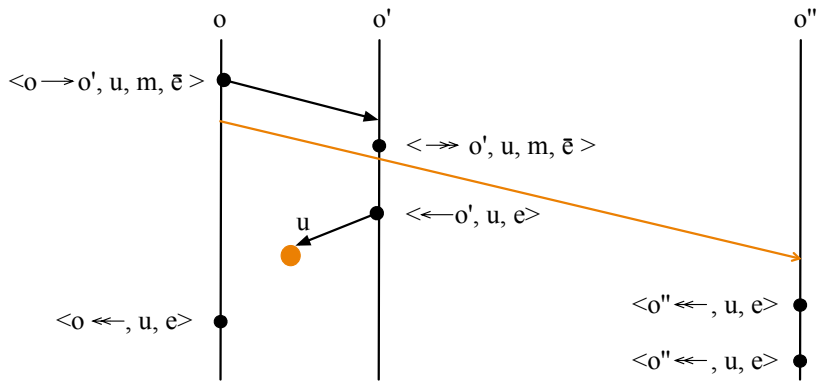


Figure 3.1: A method call cycle: object o calls a method m on object o' with *future* u. The events on the left-hand side are visible to o, those in the middle are visible to o', and the ones on the right-hand side are visible to o''. There is an arbitrary delay between message receiving and reaction. The message sending marked in orange from o to o'' represents that the *future* u is passed from o to o''.

all values occurring as actual parameters, including future identities Fid *and object identities* Oid*. Let* $caller, callee, receiver$: Oid*,* $future$: Fid*,* $method$: Mid*,* $args$: List[Data]*, and* $result$: Data*. Communication events* Ev *include:*

- Invocation events $\langle caller \rightarrow callee, future, method, args\rangle$*, generated by caller.*
- Invocation reaction events $\langle \twoheadrightarrow callee, future, method, args\rangle$*, generated by callee.*
- Future events $\langle \leftarrow\ callee, future, result\rangle$*, generated by callee.*
- Fetching events $\langle receiver\ \twoheadleftarrow, future, result\rangle$*, generated by receiver.*
- Object creation events $\langle caller \uparrow callee, class, args\rangle$*, generated by caller.*

Events may be decomposed by functions. For instance, $\_.result : \mathsf{Ev} \rightarrow \mathsf{Data}$ is well-defined for future and fetching events, e.g., $\langle \leftarrow\ o', u, e\rangle.result = e$.

The execution of a system up to present time may be described by its history of observable events, defined as a sequence. A sequence over some type T is constructed by the empty sequence ε and the right append function $\_\cdot\_ : \mathsf{Seq}[T] \times T \rightarrow \mathsf{Seq}[T]$ (where "_" indicates an argument position). Projection, $\_/\_ : \mathsf{Seq}[T] \times \mathsf{Set}[T] \rightarrow \mathsf{Seq}[T]$ is defined inductively by $\varepsilon/s \triangleq \varepsilon$ and $(a \cdot x)/s \triangleq$ **if** $x \in s$ **then** $(a/s) \cdot x$ **else** a/s **fi**, for $a : \mathsf{Seq}[T]$, $x : T$, and $s : \mathsf{Set}[T]$, restricting a to the elements in s.

For a method invocation with future u, the ordering of events depicted in Figure 3.1 is described by the following regular expression:

$$\langle o \rightarrow o', u, m, \overline{e}\rangle \cdot \langle \twoheadrightarrow o', u, m, \overline{e}\rangle \cdot \langle \leftarrow\ o', u, e\rangle[\cdot\langle\_\ \twoheadleftarrow, u, e\rangle]^*$$

for some fixed o, o', m, $\overline{e}$, e, and where _ denotes an arbitrary value. This implies that the result value may be read several times, each time with the same value, namely that given in the preceding future event.

3.2.2 History-Based Program Specification

The communication history abstractly captures the system state at any point in time [7,8]. Partial correctness properties of a system may thereby be specified by finite initial segments of its communication histories. A *history invariant* is a predicate over the communication history, which holds for all finite sequences in the (prefix-closed) set of possible histories, expressing safety properties [2]. In this section we present a framework for compositional reasoning about object systems, establishing an invariant over the global history from

invariants over the local histories of each object. Since the local object histories are disjoint with our five-event semantics, it is possible to reason locally about each object. In particular, the history updates of the operational semantics affect the local history of the active object only, and can be treated simply as an assignment to the local history. The local history is not effected by the environment, and interference-free reasoning is then possible. Correspondingly, the reasoning framework consists of two parts: A proof system for local (class-based) reasoning, and a rule for composition of object specifications.

3.2.3 Class Invariants

The class invariant must hold after initialization of all class instances and must be maintained by all methods, serving as a contract for the different methods: A method implements its part of the contract by ensuring that the invariant holds upon termination, assuming that it holds when the method starts execution. A class invariant establishes a *relationship between the internal state and the observable behavior of class instances.* The internal state reflects the values of the fields, and the observable behavior is expressed as potential communication histories. A *user-provided invariant* $I_C(\overline{w}, \mathcal{H})$ for a class C is a predicate over the fields $\overline{w}$, the read-only parameters cp and this, in addition to the local history $\mathcal{H}$ which is a sequence of events generated by this.

3.2.4 System Invariants

A history invariant for instances of C is a predicate that only talks about the local history of that object and is satisfied at all times. A history invariant can usually be derived from the class invariant (when prefix-closed). For an instance o of C with actual class parameter values $\overline{e}$, the history invariant $I_{o:C(\overline{e})}(h)$ is defined by hiding the internal state $\overline{w}$ and instantiating `this` and the class parameters $\overline{cp}$:

$$I_{o:C(\overline{e})}(h) \triangleq \exists \overline{w} \,.\, I_C(\overline{w}, h)^{\texttt{this},\overline{cp}}_{o,\overline{e}}$$

but in addition it must be proved that $I_{o:C(\overline{e})}(\mathcal{H})$ holds at all times, possibly weakening the class invariant if needed. In practice this is trivial, when the history invariant is prefix closed (with respect to the history).

We next consider systems with several objects and with an externally created initial object. The initial object may create some objects which again may create other objects and so on. We say that the system is *generated by* the externally created object.

The history invariant $I_S(h)$ for a system S given by an initial object, say $c : C(\overline{e})$, is then given by the conjunction of the history invariants of the initial and generated objects on their respective local histories:

$$I_S(h) \triangleq \langle c \uparrow c, C, \overline{e} \rangle \leq h \wedge wf(h) \bigwedge_{(o:C(\overline{e})) \in new_{ob}(h)} I_{o:C(\overline{e})}(h/o)$$

The externally created object will appear as an initial creation event in the global history, and thus be part of $new_{ob}(h)$, which returns the set of created objects in a history h. The well-formedness property wf(h) serves as a connection between the local histories, relating events with the same future to each other. Note that the system invariant is obtained directly from the history invariants of the dynamically composed objects, without any restrictions on the local reasoning, since the local histories are disjoint. This ensures compositional reasoning.

3.3 Conclusion

In Chapter B we present a compositional reasoning system for distributed, concurrent objects with asynchronous method calls and shared futures. A class invariant defines a relation between the inner state and the observable communication of instances, and can be verified independently for each class. The class invariant can be instantiated for each object of the class, resulting in a history invariant over the observable behavior of the object. Compositional reasoning is ensured as history invariants may be combined to form global system specifications.

Chapter 4

Towards the typing of resource deployment

4.1 Introduction

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 dialect of `core ABS` [20], 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.

4.2 A type based technique for resource deployment of `core ABS`

One of the prominent features of cloud computing is elasticity, namely the property of providing (almost infinite) computing resources 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-grained scale when user's needs change. However, current cloud technologies do not match this fine-grained requirement. For example, the 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 [3].

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

In Chapter C, 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 by discussing in detail a few examples. Then we discuss the type system for analysing the resource deployments. Our technique is based on so-called *behavioural types*

that abstractly describe the behaviours of systems. 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 [13] to process contracts [23] and to calculi of processes such as Milner's CCS or pi-calculus [24, 25]. 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 Chapter C are a simple model that may be displayed by highlighting the resource load of deployment components using existing tools.

The work reported in Chapter C is an overview of our type system for analysing resource deployments. Therefore the style is informal. Problems and (our) solutions are discussed mainly by means of examples. The details of the technique, such as the system for deriving behavioural types automatically and the correctness results, will be reported in the Deliverable D2.2 (Formalization of Service Contracts and SLAs – Initial Report) of the Envisage project.

Bibliography

[1] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28, 2006.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[4] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe program.: preventing data races and deadlocks. In *OOPSLA*, pages 211–230. ACM, 2002.

[6] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems*. Monographs in Computer Science. Springer-Verlag, 2001.

[7] Ole-Johan Dahl. Object-oriented specifications. In *Research directions in object-oriented programming*, pages 561–576. MIT Press, Cambridge, MA, USA, 1987.

[8] Ole-Johan Dahl. *Verifiable Programming*. International Series in Computer Science. Prentice Hall, New York, N.Y., 1992.

[9] Frank de Boer, Dave Clarke, and Einar Johnsen. A complete guide to the future. In *Progr. Lang. and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2007.

[10] Crystal Chang Din, Johan Dovland, and Olaf Owe. Compositional reasoning about shared futures. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods*, volume 7504 of *Lecture Notes in Computer Science*, pages 94–108. Springer-Verlag, 2012.

[11] Crystal Chang Din and Olaf Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *Journal of Logical and Algebraic Methods in Programming*, 2014. Available online.

[12] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *In PLDI 03: Programming Language Design and Implementation*, pages 338–349. ACM Press, 2003.

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

[14] Elena Giachino, Carlo A. Grazia, Cosimo Laneve, Michael Lienhardt, and Peter Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice. In *iFM'13*, volume 7940 of *Lecture Notes in Computer Science*, pages 394–411. Springer-Verlag, 2013.

[15] Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock detection of unbounded process networks. In *Proceedings of CONCUR 2014*, volume 8704 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2014.

[16] Elena Giachino and Cosimo Laneve. Deadlock detection in linear recursive programs. In Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Ina Schaefer, editors, *Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinoro, Italy*, volume 8483 of *Lecture Notes in Computer Science*, pages 26–64. Springer-Verlag, June 2014.

[17] Elena Giachino and Cosimo Laneve. Towards the typing of resource deployment. In Tiziana Margaria and Bernhard Steffen, editors, *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'14)*, volume 8803 of *Lecture Notes in Computer Science*, pages 90–105. Springer-Verlag, 2014. To appear.

[18] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A Framework for Deadlock Detection in ABS. *Software and Systems Modeling*, 2014. To Appear.

[19] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.

[20] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer-Verlag, 2011.

[21] Naoki Kobayashi. A new type system for deadlock-free processes. In *Proc. CONCUR 2006*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer-Verlag, 2006.

[22] Naoki Kobayashi. TyPiCal. at `kb.ecei.tohoku.ac.jp /~koba/typical/`, 2007.

[23] Cosimo Laneve and Luca Padovani. The *must* preorder revisited. In *Proc. CONCUR 2007*, volume 4703 of *Lecture Notes in Computer Science*, pages 212–225. Springer-Verlag, 2007.

[24] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[25] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Inf. and Comput.*, 100:1–77, 1992.

[26] Kohei Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 155–170. Springer-Verlag, 2008.

[27] Vasco Thudichum Vasconcelos, Francisco Martins, and Tiago Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *Proc. PLACES'09*, volume 17 of *EPTCS*, pages 95–109, 2009.

Glossary

ABS Abstract Behavioural Specification language. An executable class-based, concurrent, object-oriented modelling language based on Creol, created for the HATS project.

Behavioural Interface The intended behaviour of programs such as functional behaviour and resource consumption can be expressed in the behavioural interface. Formal specifications of program behaviour is useful for precise documentation, for the generation of test cases and test oracles, for debugging, and for formal program verification.

Communication Histories The communication history h of a system of objects S is a sequence of events, such that each event in h is generated by an object in S.

Contract Abstract specification of a program's behaviour at runtime, used to perform specific analysis on the program, like deadlock detection or resource consumption analysis.

Deployment component Abstraction encoding virtual machine identities: a cog inside a specific deployment component corresponds to a thread executing in a specific virtual machine and using its resources and computation power.

Lam Main data structure of the Deadlock Analysis which stores all the sets of await and get synchronisations between cogs possibly generated by a method's execution.

Observable Behaviour The observable behaviour of an object is the interaction between the object and its environment which can be captured in the communication history over observable events.

Records. Annex structure used in the Deadlock and the Resource Analysis which store where (i.e. in which cogs) are every manipulated objects.

Appendix A

A decision algorithm for deadlock freedom of behavioural interfaces of virtual systems

Deadlock analysis of unbounded process networks

Elena Giachino [1], Naoki Kobayashi [2], and Cosimo Laneve [1]

[1] Dept. of Computer Science and Egineering, University of Bologna – INRIA FOCUS
[2] Dept. of Computer Science, University of Tokyo

Abstract. Deadlock detection in concurrent programs that create networks with arbitrary numbers of nodes is extremely complex and solutions either give imprecise answers or do not scale. To enable the analysis of such programs, (1) we define an algorithm for detecting deadlocks of a basic model featuring recursion and fresh name generation: the *lam programs*, and (2) we design a type system for value passing CCS that returns lam programs. As a byproduct of these two techniques, we have an algorithm that is more powerful than previous ones and that can be easily integrated in the current release of `TyPiCal`, a type-based analyser for pi-calculus.

1 Introduction

Deadlock-freedom of concurrent programs has been largely investigated in the literature [2, 4, 1, 11, 18, 19]. The proposed algorithms automatically detect deadlocks by building graphs of dependencies (a, b) between resources, meaning that the release of a resource referenced by a depends on the release of the resource referenced by b. The absence of cycles in the graphs entails deadlock freedom. When programs have infinite states, in order to ensure termination, current algorithms use finite approximate models that are excerpted from the dependency graphs. The cases that are particularly critical are those of programs that create networks with an arbitrary number of nodes.

To illustrate the issue, consider the following pi-calculus-like process that computes the factorial:

```
Fact(n,r,s) =  if n=0 then r?m. s!m
               else  new t in (r?m. t!(m*n)) | Fact(n-1,t,s)
```

Here, `r?m` waits to receive a value for `m` on `r`, and `s!m` sends the value `m` on `s`. The expression **new** `t in P` creates a fresh communication channel `t` and executes `P`. If the above code is invoked with `r!1 | Fact(n,r,s)`, then there will be a synchronisation between `r!1` and the input `r?m` in the body of `Fact(n,r,s)`. In turn, this may delegate the computation of the factorial to another process in parallel by means of a subsequent synchronisation on a new channel `t`. That is, in order to compute the factorial of `n`, `Fact` builds a network of $n+1$ nodes, where node `i` takes as input a value `m` and outputs `m*i`. Due to the inability of statically reasoning about unbounded structures, the current analysers usually return false positives when fed with `Fact`. For example, this is the case of `TyPiCal` [12, 11], a tool developed for pi-calculus. (In particular, `TyPiCal` fails to recognise that there is no circularity in the dependencies among r, s, and t.)

In this paper we develop a technique to enable the deadlock analysis of processes with arbitrary networks of nodes. Instead of reasoning on finite approximations of such processes, we associate them with terms of a basic recursive model, called *lam* – for *deadLock Analysis Model* –, which collects dependencies and features recursion and dynamic name creation [5, 6]. For example, the lam function corresponding to `Fact` is

$$\texttt{fact}(a_1, a_2, a_3, a_4) = (a_2, a_3) + (\nu\, a_5, a_6)((a_2, a_6) \,\&\, \texttt{fact}(a_5, a_6, a_3, a_4))$$

where (a_2, a_3) displays the dependency between the actions `r?m` and `s!m` and (a_2, a_5) the one between `r?m` and `t!(m*n)`. The function `fact` is defined operationally by unfolding the recursive invocations; see Section 3. The unfolding of $\texttt{fact}(a_1, a_2, a_3, a_4)$ yields the following sequence of abstract states (bound names in the definition of `fact` are replaced by fresh ones in the unfoldings).

$$\begin{aligned}
&\texttt{fact}(a_1, a_2, a_3, a_4) \longrightarrow (a_2, a_3) + ((a_2, a_6) \,\&\, \texttt{fact}(a_5, a_6, a_3, a_4))\\
&\quad \longrightarrow (a_2, a_3) + (a_2, a_6) \,\&\, (a_6, a_3) + (a_2, a_6) \,\&\, (a_6, a_8) \,\&\, \texttt{fact}(a_7, a_8, a_3, a_4)\\
&\quad \longrightarrow (a_2, a_3) + (a_2, a_6) \,\&\, (a_6, a_3) + (a_2, a_6) \,\&\, (a_6, a_8) \,\&\, (a_8, a_3)\\
&\qquad +(a_2, a_6) \,\&\, (a_6, a_8) \,\&\, (a_8, a_{10}) \,\&\, \texttt{fact}(a_9, a_{10}, a_3, a_4)\\
&\quad \longrightarrow \cdots
\end{aligned}$$

While the model of `fact` is not finite-state, in Section 4 we demonstrate that it is decidable whether the computations of a lam program will ever produce a circular dependency. In our previous work [5, 6], the decidability was established only for a restricted subset of lams.

We then define a type system that associates lams to processes. Using the type system, for example, the lam program `fact` can be extracted from the factorial process `Fact`. For the sake of simplicity, we address the (*asynchronous*) *value passing CCS* [15], a simpler calculus than pi-calculus, because it is already adequate to demonstrate the power of our lam-based approach. The syntax, semantics, and examples of value passing CCS are in Section 5; the type system is defined in Section 6. As a byproduct of the above techniques, our system is powerful enough to detect deadlocks of programs that create networks with arbitrary numbers of processes. It is also worth to notice that our system admits type inference and can be easily extended to pi-calculus. We discuss the differences of our techniques with respect to the other ones in the literature in Section 7 where we also deliver some concluding remark.

2 Preliminaries

We use an infinite set $\mathscr{A}$ of (*level*) *names*, ranged over by $a, b, c, \cdots$. A relation on a set A of names, denoted $\mathtt{R}, \mathtt{R}', \cdots$, is an element of $\mathscr{P}(A \times A)$, where $\mathscr{P}(\cdot)$ is the standard powerset operator and $\cdot \times \cdot$ is the cartesian product. Let

– $\mathtt{R}^+$ be the *transitive closure* of $\mathtt{R}$.

– $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\} \Subset \{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\}$ if and only if, for all $\mathtt{R}_i$, there is $\mathtt{R}'_j$ such that $\mathtt{R}_i \subseteq \mathtt{R}'^{+}_j$.

– $(a_0, a_1), \cdots, (a_{n-1}, a_n) \in \{\mathtt{R}_1, \cdots, \mathtt{R}_m\}$ if and only if there is $\mathtt{R}_i$ such that $(a_0, a_1), \cdots, (a_{n-1}, a_n) \in \mathtt{R}_i$.

– $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\} \,\&\, \{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\} \stackrel{def}{=} \{\mathtt{R}_i \cup \mathtt{R}'_j \mid 1 \le i \le m \text{ and } 1 \le j \le n\}$.

We use $\mathscr{R}, \mathscr{R}', \cdots$ to range over $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\}$ and $\{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\}$, which are elements of $\mathscr{P}(\mathscr{P}(A \times A))$.

Definition 1. *A relation* $\mathtt{R}$ ***has a circularity*** *if* $(a, a) \in \mathtt{R}^+$ *for some a. A set of relations* $\mathscr{R}$ ***has a circularity*** *if there is* $\mathtt{R} \in \mathscr{R}$ *that has a circularity.*

For instance $\Big\{\{(a,b),(b,c)\},\ \{(a,b),(c,b),(d,b),(b,c)\},\ \{(e,d),(d,c)\},\ \{(e,d)\}\Big\}$ has a circularity because the second element of the set does.

3 The language of lams

In addition to the set of (*level*) *names*, we will also use *function names*, ranged over by $\mathtt{f}, \mathtt{g}, \mathtt{h}, \cdots$. A sequence of names is denoted by $\widetilde{a}$ and, with an abuse of notation, we also use $\widetilde{a}$ to address the *set of names* in the sequence.

A *lam program* is a pair $(\mathscr{L}, \mathtt{L})$, where $\mathscr{L}$ is a *finite set* of *function definitions* $\mathtt{f}(\widetilde{a}) = \mathtt{L}_\mathtt{f}$, with $\widetilde{a}$ and $\mathtt{L}_\mathtt{f}$ being the *formal parameters* and the *body* of $\mathtt{f}$, and $\mathtt{L}$ is the *main lam*. The syntax of the function bodies and the main lam is

$$\mathtt{L} \quad ::= \quad \mathbb{0} \ \mid\ (a,b) \ \mid\ \mathtt{f}(\widetilde{a}) \ \mid\ \mathtt{L} \,\&\, \mathtt{L} \ \mid\ \mathtt{L} + \mathtt{L} \ \mid\ (\nu\, a)\mathtt{L}$$

The lam $\mathbb{0}$ enforces no dependency, the lam (a, b) enforces the dependency (a, b), while $\mathtt{f}(\widetilde{a})$ represents a function invocation. The composite lam $\mathtt{L} \,\&\, \mathtt{L}'$ enforces the dependencies of $\mathtt{L}$ *and* of $\mathtt{L}'$, while $\mathtt{L} + \mathtt{L}'$ nondeterministically enforces the dependencies of $\mathtt{L}$ *or* of $\mathtt{L}'$, $(\nu\, a)\mathtt{L}$ creates a fresh name a and enforces the dependencies of $\mathtt{L}$ that may use a. Whenever parentheses are omitted, the operation "&" has precedence over "+". We will shorten $\mathtt{L}_1 \,\&\, \cdots \,\&\, \mathtt{L}_n$ into $\&_{i \in 1..n}\mathtt{L}_i$ and $(\nu\, a_1) \cdots (\nu\, a_n)\mathtt{L}$ into $(\nu\, a_1 \cdots a_n)\mathtt{L}$. Function definitions $\mathtt{f}(\widetilde{a}) = \mathtt{L}_\mathtt{f}$ and $(\nu\, a)\mathtt{L}$ are *binders* of $\widetilde{a}$ in $\mathtt{L}_\mathtt{f}$ and of a in $\mathtt{L}$, respectively, and the corresponding occurrences of $\widetilde{a}$ in $\mathtt{L}_\mathtt{f}$ and of a in $\mathtt{L}$ are called *bound*. A name x in $\mathtt{L}$ is *free* if it is not underneath a $(\nu\, a)$ (similarly for function definitions). Let $var(\mathtt{L})$ be the set of free names in $\mathtt{L}$.

In the syntax of $\mathtt{L}$, the operations "&" and "+" are associative, commutative with $\mathbb{0}$ being the identity on &, and definitions and lams are equal up-to alpha renaming of bound names. Namely, if $a \notin var(\mathtt{L})$, the following axioms hold:

$$(\nu\, a)\mathtt{L} = \mathtt{L} \qquad (\nu\, a)\mathtt{L}' \,\&\, \mathtt{L} = (\nu\, a)(\mathtt{L}' \,\&\, \mathtt{L}) \qquad (\nu\, a)\mathtt{L}' + \mathtt{L} = (\nu\, a)(\mathtt{L}' + \mathtt{L})$$

Additionally, when $\mathtt{V}$ ranges over lams that do not contain function invocations, the following axioms hold:

$$\mathtt{V} \,\&\, \mathtt{V} = \mathtt{V} \qquad \mathtt{V} + \mathtt{V} = \mathtt{V} \qquad \mathtt{V} \,\&\, (\mathtt{L}' + \mathtt{L}'') = \mathtt{V} \,\&\, \mathtt{L}' + \mathtt{V} \,\&\, \mathtt{L}'' \tag{1}$$

These axioms permit to rewrite a lam without function invocations as a *collection* (operation +) *of relations* (elements of a relation are gathered by the operation &). Let $\equiv$ be the least congruence containing the above axioms.

Definition 2. *A lam* $\mathtt{V}$ *is in* ***normal form****, denoted* $\mathtt{nf}(\mathtt{V})$*, if* $\mathtt{V} = (\nu\, \widetilde{a})(\mathtt{V}_1 + \cdots + \mathtt{V}_n)$*, where* $\mathtt{V}_1, \cdots, \mathtt{V}_n$ *are dependencies only composed with &.*

Proposition 1. *For every* $\mathtt{V}$*, there is* $\mathtt{nf}(\mathtt{V})$ *such that* $\mathtt{V} \equiv \mathtt{nf}(\mathtt{V})$.

In the rest of the paper, we will never distinguish between equal lams. Moreover, we always assume lam programs $(\mathscr{L}, \mathtt{L})$ to be *well formed*.

Remark 1. The axioms (1) are restricted to terms $\mathtt{V}$ that do not contain function invocations. In fact, $\mathtt{f}(\widetilde{d})\&((a,b)+(b,c)) \neq (\mathtt{f}(\widetilde{d})\&(a,b))+(\mathtt{f}(\widetilde{d})\&(b,c))$ because the evaluation of the two lams (see below) may produce terms with different names.

In the paper, we always assume lam programs $(\mathscr{L}, \mathtt{L})$ to be *well-formed*.

Definition 3. *A lam program* $(\mathscr{L}, \mathtt{L})$ *is* ***well formed*** *if* (1) *function definitions in* $\mathscr{L}$ *have pairwise different function names and all function names occurring in the function bodies and* $\mathtt{L}$ *are defined;* (2) *the arity of function invocations occurring anywhere in the program matches the arity of the corresponding function definition;* (3) *every function definition in* $\mathscr{L}$ *has shape* $\mathtt{f}(\widetilde{a}) = (\nu\,\widetilde{c})\mathtt{L}_{\mathtt{f}}$*, where* $\mathtt{L}_{\mathtt{f}}$ *does not contain any* ν*-binder and* $var(\mathtt{L}_{\mathtt{f}}) \subseteq \widetilde{a} \cup \widetilde{c}$*.*

Operational semantics. Let a *lam context*, noted $\mathfrak{L}[\,]$, be a term derived by the following syntax:

$$\mathfrak{L}[\,] \quad ::= \quad [\,] \qquad | \qquad \mathtt{L} \,\&\, \mathfrak{L}[\,] \qquad | \qquad \mathtt{L} + \mathfrak{L}[\,]$$

As usual $\mathfrak{L}[\mathtt{L}]$ is the lam where the hole of $\mathfrak{L}[\,]$ is replaced by $\mathtt{L}$. According to the syntax, lam contexts have no ν-binder; that is, the hassle of name captures is avoided. The operational semantics of a program $(\mathscr{L}, \mathtt{L})$ is a transition system where *states* are lams, the *transition relation* is the least one satisfying the rule

$$\text{(Red)}\quad \frac{\mathtt{f}(\widetilde{a}) = (\nu\,\widetilde{c})\mathtt{L}_{\mathtt{f}} \in \mathscr{L} \qquad \widetilde{c}' \text{ are fresh} \qquad \mathtt{L}_{\mathtt{f}}[\widetilde{c}'/\widetilde{c}][\widetilde{a}'/\widetilde{a}] = \mathtt{L}'_{\mathtt{f}}}{\mathfrak{L}[\mathtt{f}(\widetilde{a}')] \longrightarrow \mathfrak{L}[\mathtt{L}'_{\mathtt{f}}]}$$

and the initial state is the lam $\mathtt{L}'$ such that $\mathtt{L} \equiv (\nu\,\widetilde{c})\mathtt{L}'$ and $\mathtt{L}'$ does not contain any ν-binder. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

By (Red), a lam $\mathtt{L}$ is evaluated by successively replacing function invocations with the corresponding lam instances. Name creation is handled by replacing bound names of function bodies with fresh names. For example, if $\mathtt{f}(a) = (\nu\,c)((a,c) \,\&\, \mathtt{f}(c))$ and $\mathtt{f}(a')$ occurs in the main lam, then $\mathtt{f}(a')$ is replaced by $(a',c') \,\&\, \mathtt{f}(c')$, where c' is a fresh name.

Let us discuss some examples.

1. $(\{\mathtt{f}(a,b,c) = (a,b) \,\&\, \mathtt{g}(b,c) + (b,c),\ \mathtt{g}(d,e) = (d,e) + (e,d)\},\ \mathtt{f}(a,b,c)\,)$. Then
$$\begin{aligned}\mathtt{f}(a,b,c) &\longrightarrow (a,b) \,\&\, \mathtt{g}(b,c) + (b,c) \longrightarrow (a,b) \,\&\, ((b,c) + (c,b)) + (b,c)\\ &\longrightarrow (a,b) \,\&\, (b,c) + (a,b) \,\&\, (c,b) + (b,c)\end{aligned}$$
The lam in the final state *does not contain function invocations*. This is because the above program is not recursive. Additionally, the evaluation of $\mathtt{f}(a,b,c)$ *has not created names*. This is because the bodies of $\mathtt{f}$ and $\mathtt{g}$ do not contain ν-binders.
2. $(\{\mathtt{f}'(a) = (\nu\,b)(a,b) \,\&\, \mathtt{f}'(b)\}\ ,\ \mathtt{f}'(a_0))$. Then
$$\begin{aligned}\mathtt{f}'(a_0) &\longrightarrow (a_0,a_1) \,\&\, \mathtt{f}'(a_1) \longrightarrow (a_0,a_1) \,\&\, (a_1,a_2) \,\&\, \mathtt{f}'(a_2)\\ &\longrightarrow^n (a_0,a_1) \,\&\, \cdots \,\&\, (a_{n+1},a_{n+2}) \,\&\, \mathtt{f}'(a_{n+2})\end{aligned}$$
In this case, *because of the* $(\nu\,b)$ *binder*, the lam grows in the number of dependencies as the evaluation progresses.

3. $(\{\mathtt{f}''(a) = (\nu\, b)(a,b) + (a,b) \,\&\, \mathtt{f}''(b)\},\ \mathtt{f}''(a_0))$. Then

$$\begin{aligned}\mathtt{f}''(a_0) &\longrightarrow (a_0,a_1) + (a_0,a_1) \,\&\, \mathtt{f}''(a_1)\\ &\longrightarrow (a_0,a_1) + (a_0,a_1) \,\&\, (a_1,a_2) + (a_0,a_1) \,\&\, (a_1,a_2) \,\&\, \mathtt{f}''(a_2)\\ &\longrightarrow^n (a_0,a_1) + \cdots + (a_0,a_1) \,\&\, \cdots \,\&\, (a_{n+1},a_{n+2}) \,\&\, \mathtt{f}''(a_{n+2})\end{aligned}$$

In this case, the lam grows in the number of "+"-terms, which in turn become larger and larger as the evaluation progresses.

Flattening and circularities. Lams represent elements of the set $\mathscr{P}(\mathscr{P}(\mathscr{A} \times \mathscr{A}))$. This property is displayed by the following flattening function.

Let $\mathscr{L}$ be a set of function definitions and let $I(\cdot)$, called *flattening*, be a function on lams that (i) maps function name $\mathtt{f}$ defined in $\mathscr{L}$ to elements of $\mathscr{P}(\mathscr{P}(A \times A))$ and (ii) is defined on lams as follows

$$I(\mathbb{0}) = \{\varnothing\}, \qquad I((a,b)) = \{\{(a,b)\}\}, \qquad I(\mathtt{L} \,\&\, \mathtt{L}') = I(\mathtt{L}) \,\&\, I(\mathtt{L}'),$$
$$I(\mathtt{L} + \mathtt{L}') = I(\mathtt{L}) \cup I(\mathtt{L}'), \qquad I((\nu\, a)\mathtt{L}) = I(\mathtt{L})[{}^{a'}/_{a}] \ \text{ with } a' \text{ fresh},$$
$$I(\mathtt{f}(\widetilde{c})) = I(\mathtt{f})[{}^{\widetilde{c}}/_{\widetilde{a}}] \text{ (where } \widetilde{a} \text{ are the formal parameters of } \mathtt{f}).$$

Note that $I(\mathtt{L})$ is unique up to a renaming of names that do not occur free in $\mathtt{L}$. Let $I^{\perp}$ be the map such that, for every $\mathtt{f}$ defined in $\mathscr{L}$, $I^{\perp}(\mathtt{f}) = \{\varnothing\}$. For example, let $\mathscr{L}$ defines $\mathtt{f}$ and $\mathtt{g}$ and let

$$I(\mathtt{f}) = \{\{(a,b),(b,c)\}\} \qquad I(\mathtt{g}) = \{\{(b,a)\}\}$$
$$\mathtt{L}'' = \mathtt{f}(a,b,c) + (a,b) \,\&\, \mathtt{g}(b,c) \,\&\, \mathtt{f}(d,b,c) + \mathtt{g}(d,e) \,\&\, (d,c) + (e,d).$$

Then

$$I(\mathtt{L}'') = \{\{(a,b),(b,c)\},\ \{(a,b),(c,b),(d,b),(b,c)\},\ \{(e,d),(d,c)\},\ \{(e,d)\}\}$$
$$I^{\perp}(\mathtt{L}'') = \{\varnothing, \{(a,b)\}, \{(d,c)\}, \{(e,d)\}\}\ .$$

Definition 4. *A lam* $\mathtt{L}$ ***has a circularity*** *if* $I^{\perp}(\mathtt{L})$ *has a circularity. A lam program* $(\mathscr{L}, \mathtt{L})$ ***has a circularity*** *if there is* $\mathtt{L} \longrightarrow^{*} \mathtt{L}'$ *and* $\mathtt{L}'$ *has a circularity.*

The property of "having a circularity" is preserved by $\equiv$ while the "absence of circularities" of a composite lam can be carried to its components.

Proposition 2. *1. if* $\mathtt{L} \equiv \mathtt{L}'$ *then* $\mathtt{L}$ *has a circularity if and only if* $\mathtt{L}'$ *has a circularity;*
2. $\mathtt{L} \,\&\, \mathtt{L}'$ *has no circularity implies both* $\mathtt{L}$ *and* $\mathtt{L}'$ *have no circularity (similarly for* $\mathtt{L} + \mathtt{L}'$ *and for* $(\nu\, a)\mathtt{L}$*).*

4 The decision algorithm for detecting circularities

In this section we assume a lam program $(\mathscr{L}, \mathtt{L})$ such that pairwise different function definitions in $\mathscr{L}$ have disjoint formal parameters. Without loss of generality, we assume that $\mathtt{L}$ does not contain any ν-binder.

Fixpoint definition of the interpretation function. The basic item of our algorithm is the computation of lam functions' interpretation. This computation is performed by means of a standard fixpoint technique that is detailed below.

Let A be the set of formal parameters of definitions in $\mathscr{L}$ and let $\varkappa$ be a special name that does not occur in $(\mathscr{L}, \mathtt{L})$. We use the domain $\big(\mathscr{P}(\mathscr{P}(A \cup \{\varkappa\} \times A \cup \{\varkappa\})),\ \subseteq\big)$ which is a *finite* lattice [3].

Definition 5. *Let* $\mathtt{f}_i(\widetilde{a}_i) = (\nu\widetilde{c}_i)\mathtt{L}_i$*, with* $i \in 1..n$*, be the function definitions in* $\mathscr{L}$*. The family of flattening functions* $I^{(k)}_{\mathscr{L}} : \{\mathtt{f}_1, \cdots, \mathtt{f}_n\} \to \mathscr{P}(\mathscr{P}(A \cup \{\varkappa\} \times A \cup \{\varkappa\}))$ *is defined as follows*

$$I^{(0)}_{\mathscr{L}}(\mathtt{f}_i) \ = \{\varnothing\} \qquad I^{(k+1)}_{\mathscr{L}}(\mathtt{f}_i) \ = \{\mathtt{proj}_{\widetilde{a}_i}(R^+) \mid \ R \in I^{(k)}_{\mathscr{L}}(\mathtt{L}_i)\}$$

where $\mathtt{proj}_{\widetilde{a}}(R) \stackrel{def}{=} \{(a,b) \mid \ (a,b) \in R \text{ and } a,b \in \widetilde{a}\} \cup \ \{(\varkappa,\varkappa) \mid \ (c,c) \in R \text{ and } c \notin \widetilde{a}\}$.

We notice that $I^{(0)}_{\mathscr{L}}$ is the function $I^{\perp}$ of the previous section.

Proposition 3. *Let* $\mathtt{f}(\widetilde{a}) = (\nu\widetilde{c})\mathtt{L}_{\mathtt{f}} \ \in \mathscr{L}$*. (i) For every k,* $I^{(k)}_{\mathscr{L}}(\mathtt{f}) \in \mathscr{P}(\mathscr{P}((\widetilde{a} \cup \{\varkappa\}) \times (\widetilde{a} \cup \{\varkappa\})))$*. (ii) For every k,* $I^{(k)}_{\mathscr{L}}(\mathtt{f}) \Subset I^{(k+1)}_{\mathscr{L}}(\mathtt{f})$*.*

Proof. 1 is immediate by definition. 2 follows by a straightforward structural induction on L. □

Since, for every k, $I^{(k)}_{\mathscr{L}}(\mathtt{f}_i)$ ranges over a finite lattice, by the fixpoint theory [3], there exists m such that $I^{(m)}_{\mathscr{L}}$ is a fixpoint, namely $I^{(m)}_{\mathscr{L}} \approx I^{(m+1)}_{\mathscr{L}}$ where $\approx$ is the equivalence relation induced by $\Subset$. In the following, we let $I_{\mathscr{L}}$, called the *interpretation function* (of a lam), be the least fixpoint $I^{(m)}_{\mathscr{L}}$.

Example 1. For example, let $\mathscr{L}$ be the factorial function in Section 1. Then

$$I^{(0)}_{\mathscr{L}}(\mathtt{fact}) \ = \{\varnothing\} \qquad I^{(1)}_{\mathscr{L}}(\mathtt{fact}) \ = \{\{(a_2,a_3)\},\varnothing\} \qquad I^{(2)}_{\mathscr{L}}(\mathtt{fact}) \ = \{\{(a_2,a_3)\},\varnothing\}$$

That is, in this case, $I_{\mathscr{L}} = I^{(1)}_{\mathscr{L}}$. □

Lemma 1. *Let* $\mathtt{f}(\widetilde{a}) = (\nu\widetilde{c})\mathtt{L}_{\mathtt{f}} \ \in \mathscr{L}$ *and* $b', b'' \in \widetilde{b}$ *and* $\widetilde{c'}$ *be disjoint from* $\widetilde{b}, \widetilde{a}$*. Then* $(b', b'') \in I^{(k+1)}_{\mathscr{L}}(\mathtt{f}(\widetilde{b}))$ *if and only if there is* $\mathtt{R} \in I^{(k)}_{\mathscr{L}}(\mathtt{L}_{\mathtt{f}}[\widetilde{c'}/\widetilde{c}][\widetilde{b}/\widetilde{a}])$ *such that* $(b', b'') \in \mathtt{R}^+$*. (In particular this statement holds when* $I^{(k+1)}_{\mathscr{L}} = I^{(k)}_{\mathscr{L}} = I_{\mathscr{L}}$*.)*

Lam programs and circularities. Below we use *multiple lam contexts*, that is lam contexts with several holes, written $\mathfrak{L}[\ ]\cdots[\ ]$, with the standard meaning. For example, if $\mathfrak{L}[\ ][\ ] = [\ ] \,\&\, (a,b) + [\ ]$ then $\mathfrak{L}[\mathtt{f}(b,c)][(a,c)] = \mathtt{f}(b,c) \,\&\, (a,b) + (a,c)$.

Lemma 2. *Let* $(\{\mathtt{f}_1(\widetilde{a}_1) = (\nu\widetilde{c}_1)\mathtt{L}_1, \cdots, \mathtt{f}_n(\widetilde{a}_n) = (\nu\widetilde{c}_n)\mathtt{L}_n\},\ \mathtt{L})$ *be a lam program and let*

$$\mathfrak{L}[\mathtt{f}_{i_1}(\widetilde{a'_1})]\cdots[\mathtt{f}_{i_m}(\widetilde{a'_m})] \longrightarrow^m \mathfrak{L}[\mathtt{L}_{i_1}[\widetilde{c'_1}/\widetilde{c_{i_1}}][\widetilde{a'_1}/\widetilde{a_{i_1}}]]\cdots[\mathtt{L}_{i_m}[\widetilde{c'_m}/\widetilde{c_{i_m}}][\widetilde{a'_m}/\widetilde{a_{i_m}}]]$$

where $\mathfrak{L}[\cdot]\cdots[\cdot]$ *is a multiple context without function invocations.*

Then, the following two properties are equivalent:

1. $I^{(k+1)}_{\mathscr{L}}(\mathfrak{L}[\mathtt{f}_{i_1}(\widetilde{a'}_1)]\cdots[\mathtt{f}_{i_m}(\widetilde{a'}_m)])$ *has a circularity,*
2. $I^{(k)}_{\mathscr{L}}(\mathfrak{L}[\mathtt{L}_{i_1}[\widetilde{c'}_1/\widetilde{c}_{i_1}][\widetilde{a'}_1/\widetilde{a}_{i_1}]]\cdots[\mathtt{L}_{i_m}[\widetilde{c'}_m/\widetilde{c}_{i_m}][\widetilde{a'}_m/\widetilde{a}_{i_m}]])$ *has a circularity.*

Proof. Let $\mathtt{L}' = \mathfrak{L}[\mathtt{f}_{i_1}(\widetilde{a'}_1)]\cdots[\mathtt{f}_{i_m}(\widetilde{a'}_m)]$ and $\mathtt{L}'' = \mathfrak{L}[\mathtt{L}_{i_1}[\widetilde{c'}_1/\widetilde{c}_{i_1}][\widetilde{a'}_1/\widetilde{a}_{i_1}]]\cdots[\mathtt{L}_{i_m}[\widetilde{c'}_m/\widetilde{c}_{i_m}][\widetilde{a'}_m/\widetilde{a}_{i_m}]]$. For the implication 2⇒ 1, there are two subcases:

a) $I^{(k)}_{\mathscr{L}}(\mathtt{L}'')$ has a circularity consisting *only of* names in $\widetilde{c'}_i$. Namely there are $c'_0, c'_1, \cdots, c'_h \in \widetilde{c'}_j$ such that
$$(c'_0, c'_1), (c'_1, c'_2), \cdots, (c'_h, c'_0) \in I^{(k)}_{\mathscr{L}}(\mathtt{L}'')\ .$$
Since names $\widetilde{c'}_i$ are fresh, then, by definition of $I_{\mathscr{L}}(\cdot)$, the circularity must occur in $I^{(k)}_{\mathscr{L}}(\mathtt{L}_{i_j}[\widetilde{c'}_i/\widetilde{c}_{i_j}])$, and conversely. In turn, this is possible if and only if $I^{(k)}_{\mathscr{L}}(\mathtt{L}_{i_j})$ has a circularity consisting of names in $\widetilde{c}_{i_j}$ (because $[\widetilde{c'}_j/\widetilde{c}_{i_j}]$ is a bijective renaming). This means, by definition of $I^{(k+1)}_{\mathscr{L}}(\cdot)$, that $(\varkappa,\varkappa) \in I^{(k+1)}_{\mathscr{L}}(\mathtt{f}_{i_j})$ and, in turn, $(\varkappa,\varkappa) \in I^{(k+1)}_{\mathscr{L}}(\mathtt{L}')$.

b) $I^{(k)}_{\mathscr{L}}(\mathtt{L}'')$ has a circularity, let it be $(b_0, b_1), \cdots (b_h, b_0)$, that also contains names not in $\widetilde{c'}_1, \cdots, \widetilde{c'}_m$. Without loss of generality, let
$$(b_0, b_1), \cdots, (b_{h'-1}, b_{h'}) \in I^{(k)}_{\mathscr{L}}(\mathtt{L}_{i_j}[\widetilde{c'}_j/\widetilde{c}_{i_j}][\widetilde{a'}_j/\widetilde{a}_{i_j}]) \tag{2}$$
while the other pairs of the circularity come from the context $\mathfrak{L}[\,]\cdots[\,]$. The general case follows by iterating the following argument. Then, by Lemma 1, (2) is possible if and only if $(b_0, b_{h'}) \in I^{(k+1)}_{\mathscr{L}}(\mathtt{f}_{i_j}(\widetilde{a'}_j))$. This last statement gives $(b_0, b_{h'}), (b_{h'+1}, b_{h'+2}), \cdots (b_h, b_0) \in I^{(k+1)}_{\mathscr{L}}(\mathtt{L}')$.

For the converse, we consider two cases.

c) Case $(\varkappa,\varkappa) \in I^{(k+1)}_{\mathscr{L}}(\mathtt{L}')$. If $(\varkappa,\varkappa)$ comes from the context $\mathfrak{L}$, $(\varkappa,\varkappa) \in I^{(k)}_{\mathscr{L}}(\mathtt{L}'')$ follows immediately. If $(\varkappa,\varkappa)$ comes from $I^{(k+1)}_{\mathscr{L}}(\mathtt{f}_{i_j}(\widetilde{a'}_j))$, then by the definition of $I^{(k+1)}_{\mathscr{L}}$, $I^{(k)}_{\mathscr{L}}(\mathtt{L}_{i_j})$ also has a circularity, hence also $I^{(k)}_{\mathscr{L}}(\mathtt{L}'')$.

d) Otherwise, $I^{(k+1)}_{\mathscr{L}}(\mathtt{L}')$ has a circularity on names other than $\varkappa$. By the definition of $I^{(k+1)}_{\mathscr{L}}(\cdot)$, there exists $R_j \in I^{(k+1)}_{\mathscr{L}}(\mathtt{f}_{i_j}(\widetilde{a'}_j))$ for each $j \in \{1, \ldots, m\}$ such that
$$I^{(k+1)}_{\mathscr{L}}(\mathfrak{L}[R_1]\cdots[R_m])$$
has a circularity. (Here, we have identified $R \subseteq A \times A$ with the lam expression $\&_{(a_1,a_2)\in R}(a_1, a_2)$.) Because $\mathfrak{L}$ does not contain function invocations, we have
$$I^{(k+1)}_{\mathscr{L}}(\mathfrak{L}[R_1]\cdots[R_m]) = I^{(k)}_{\mathscr{L}}(\mathfrak{L}[R_1]\cdots[R_m]).$$
By the definition of $I^{(k+1)}_{\mathscr{L}}$, there exists $R'_j \in I^{(k)}_{\mathscr{L}}(\mathtt{L}_{i_j}[\widetilde{c'}_j/\widetilde{c}_{i_j}][\widetilde{a'}_j/\widetilde{a}_{i_j}])$ such that $R_j \subseteq R'^{+}_j$. Therefore, $I^{(k)}_{\mathscr{L}}(\mathfrak{L}[R'_1]\cdots[R'_m])$ has a circularity, hence also $I^{(k)}_{\mathscr{L}}(\mathtt{L}'')$. □

Lemma 3. *Let* $(\mathscr{L}, \mathtt{L})$ *be a lam program and* $\mathfrak{L}[\mathtt{f}(\widetilde{a'})] \longrightarrow \mathfrak{L}[\mathtt{L}'[\widetilde{c'}/\widetilde{c}][\widetilde{a'}/\widetilde{a}]]$. *The following two properties are equivalent:*

1. $I_{\mathscr{L}}(\mathfrak{L}[\mathtt{f}(\widetilde{a})])$ *has a circularity,*

2. $I_{\mathscr{L}}(\mathfrak{L}[\mathtt{L}'[\widetilde{c'}/\widetilde{c}][\widetilde{a'}/\widetilde{a}]])$ *has a circularity.*

Proof. The proof is similar to the one of Lemma 2: we consider the k such that $I^{(k)}_{\mathscr{L}} = I$ and we reason on $I^{(k+1)}$ and $I^{(k)}$. □

Theorem 1. *A lam program* $(\mathscr{L}, \mathtt{L})$ *has a circularity if and only if* $I_{\mathscr{L}}(\mathtt{L})$ *has a circularity.*

Proof. (*If* direction) By definition, $(\mathscr{L}, \mathtt{L})$ has a circularity if there is $\mathtt{L} \longrightarrow^* \mathtt{L}'$ such that $I^{\perp}(\mathtt{L}')$ has a circularity. By induction on the length of $\mathtt{L} \longrightarrow^* \mathtt{L}'$. When the length is 0 then $I^{\perp}(\mathtt{L}')$ has a circularity implies $I_{\mathscr{L}}(\mathtt{L})$ has a circularity (by $I^{\perp}(\mathtt{L}') = I^{(0)}_{\mathscr{L}}(\mathtt{L}')$ and Proposition 3(2)). Assume $\mathtt{L} \longrightarrow^* \mathtt{L}'$ be equal to $\mathtt{L} \longrightarrow \mathtt{L}'' \longrightarrow^* \mathtt{L}'$. By inductive hypothesis, we assume that the theorem holds on the computation $\mathtt{L}'' \longrightarrow^* \mathtt{L}'$. Then, by Lemma 3, if $I_{\mathscr{L}}(\mathtt{L}'')$ has a circularity then $I_{\mathscr{L}}(\mathtt{L})$ has a circularity. Therefore the theorem.

(*Only-if* direction) We demonstrate that, if $I_{\mathscr{L}}(\mathtt{L})$ has a circularity then there is $\mathtt{L} \longrightarrow^* \mathtt{L}'$ such that $I^{\perp}(\mathtt{L}')$ has a circularity.

Let m be the least natural number such that $I_{\mathscr{L}} = I^{(m)}_{\mathscr{L}}$. Let $\mathtt{L} = \mathfrak{L}[\mathtt{f}_{i_1}(\widetilde{a'_1})] \cdots [\mathtt{f}_{i_n}(\widetilde{a'_n})]$ such that $\mathfrak{L}[\ ] \cdots [\ ]$ does not contain function invocations. Then

$$\mathtt{L} \longrightarrow^n \mathfrak{L}[\mathtt{L}_{i_1}[\widetilde{c'_1}/\widetilde{c}_{i_1}][\widetilde{a'_1}/\widetilde{a}_{i_1}]] \cdots [\mathtt{L}_{i_n}[\widetilde{c'_n}/\widetilde{c}_{i_n}][\widetilde{a'_n}/\widetilde{a}_{i_n}]] = \mathtt{L}''$$

where $\widetilde{c'_1}, \cdots, \widetilde{c'_n}$ are fresh. Additionally, by Lemma 2, $I^{(m-1)}_{\mathscr{L}}(\mathtt{L}'')$ has a circularity because $I^{(m)}_{\mathscr{L}}(\mathtt{L}')$ has a circularity. Now, we reapply the same argument to $\mathtt{L}''$ since $I^{(m-1)}_{\mathscr{L}}(\mathtt{L}'')$ has a circularity. After m-steps we get a lam $\mathtt{L}'$ such that $I^{(0)}_{\mathscr{L}}(\mathtt{L}') = I^{\perp}(\mathtt{L}')$ has a circularity. □

For example, let $\mathscr{L}$ be the factorial function in Section 1 and let $\mathtt{L} = (a_3, a_2) \,\&\, \mathtt{fact}(a_1, a_2, a_3.a_4)$. From Example 1, we have $I_{\mathscr{L}}(\mathtt{fact}) = \{\{(a_2, a_3)\}, \emptyset\}$. Since $I_{\mathscr{L}}(\mathtt{L})$ has a circularity, by Theorem 1, there is $\mathtt{L} \longrightarrow^* \mathtt{L}'$ such that $I^{\perp}(\mathtt{L}')$ has a circularity. In fact it displays a circularity after the first transition:

$$\mathtt{L} \longrightarrow (a_3, a_2) \,\&\, ((a_2, a_3) + ((a_2, a_5) \,\&\, \mathtt{fact}(a_5, a_6, a_3, a_4))) \ .$$

5 Value-passing CCS

In the present and next sections, we apply the foregoing theory of lams to refine Kobayashi's type system for deadlock-freedom of concurrent programs [11]. In his type system, the deadlock-freedom is guaranteed by a combination of *usages*, which are a kind of behavioral types capturing channel-wise communication behaviors, and *capability/obligation levels*, which are natural numbers capturing inter-channel dependencies (like "a message is output on x only if a message is received along y"). By replacing numbers with (lam) level names, we can achieve a more precise analysis of deadlock-freedom because of the algorithm in Section 4. The original type system in [11] is for the pi-calculus [16], but for the sake of simplicity, we consider a variant of the value-passing CCS [15], which is sufficient for demonstrating the power of our lam-based approach.

P (processes) ::= $\mathbb{0} \mid x!e \mid x?y.P \mid (P \mid Q) \mid$ **if** e **then** P **else** $Q \mid (\nu\widetilde{a}; x : \mathtt{T})P \mid A(\widetilde{a}; \widetilde{e})$
e (expressions) ::= $x \mid v \mid e_1$ **op** e_2
$\mathtt{T}$ (types) ::= **int** $\mid U$
U (usages) ::= $\mathbf{0} \mid !^{a_1}_{a_2} \mid ?^{a_1}_{a_2}.U \mid (U_1|U_2) \mid \alpha \mid \mu\alpha.U$

Fig. 1. The Syntax of value-passing CCS

Our *value-passing CCS* uses several disjoint countable sets of names: in addition to level names, there are *integer and channel names*, ranged over by $x, y, z, \cdots$, *process names*, ranged over by A, B, $\cdots$, and *usage names*, ranged over by $\alpha, \beta, \cdots$. A *value-passing CCS program* is a pair $(\mathscr{D}, P)$, where $\mathscr{D}$ is a *finite set* of *process name definitions* $A(\widetilde{a}; \widetilde{x}) = P_A$, with $\widetilde{a}; \widetilde{x}$ and P_A respectively being the *formal parameters* and the *body* of A, and P is the *main process*.

The syntax of processes P_A and P is shown in Figure 1. A process can be the inert process $\mathbb{0}$, a message $x!e$ sent on a name x that carries (the value of) an expression e, an input $x?y.P$ that consumes a message $x!v$ and behaves like $P[^v/_y]$, a parallel composition of processes $P \mid Q$, a conditional **if** e **then** P **else** Q that evaluates e and behaves either like P or like Q depending on whether the value is $\neq 0$ (*true*) or $= 0$ (*false*), a restriction $(\nu\widetilde{a}; x : \mathtt{T})P$ that behaves like P except that communications on x with the external environment are prohibited, an invocation $A(\widetilde{a}; \widetilde{e})$ of the process corresponding to A.

An expression e can be a name x, an integer value v, or a generic binary operation on integers v **op** v', where **op** ranges over a set including the usual operators like $+$, $\leq$, etc. Integer expressions without names (*constant expressions*) may be evaluated to an integer value (the definition of the evaluation of constant expressions is omitted). Let $[\![e]\!]$ be the evaluation of a constant expression e ($[\![e]\!]$ is undefined when the integer expression e contains integer names). Let also $[\![x]\!] = x$ when x is a non-integer name.

We defer the explanation of the meaning of types $\mathtt{T}$ (and usages U) until Section 6. It is just for the sake of simplicity that processes are annotated with types and level names. They do not affect the operational semantics of processes, and can be automatically inferred by using an inference algorithm similar to those in [11, 10].

Similarly to lams, $A(\widetilde{a}; \widetilde{x}) = P_A$ and $(\nu\widetilde{a}; x : \mathtt{T})P$ are *binders* of $\widetilde{a}; \widetilde{x}$ in P_A and of $\widetilde{a}, x$ in P, respectively. We use the standard notions of alpha-equivalence, free and bound names of processes and, with an abuse of notation, we let $var(P)$ be the free names in P. In process name definitions $A(\widetilde{a}; \widetilde{x}) = P_A$, we always assume that $var(P_A) \subseteq \widetilde{a}, \widetilde{x}$.

Definition 6. *The* ***structural equivalence*** $\equiv$ *on processes is the least congruence containing alpha-conversion of bound names, commutativity and associativity of* $\mid$ *with identity* $\mathbb{0}$*, and closed under the following rule:*

$$((\nu\widetilde{a}; x : T)P) \mid Q \equiv (\nu\widetilde{a}; x : T)(P \mid Q) \qquad \widetilde{a}, x \notin var(Q) \ .$$

The ***operational semantics*** *of a program* $(\mathscr{D}, P)$ *is a transition system where the states are processes, the initial state is* P*, and the* ***transition relation*** $\rightarrow_{\mathscr{D}}$ *is the least one*

closed under the following rules:

$$\text{(R-Com)}\ \frac{[\![e]\!] = v}{x!e \mid x?y.P \to_{\mathscr{D}} P[^{v}/_{y}]} \qquad \text{(R-Par)}\ \frac{P \to_{\mathscr{D}} P'}{P \mid Q \to_{\mathscr{D}} P' \mid Q} \qquad \text{(R-New)}\ \frac{P \to_{\mathscr{D}} Q}{(\nu\widetilde{a}; x : T)P \to_{\mathscr{D}} (\nu\widetilde{a}; x : T)Q}$$

$$\text{(R-IfT)}\ \frac{[\![e]\!] \neq 0}{\textbf{if } e \textbf{ then } P \textbf{ else } Q \to_{\mathscr{D}} P} \qquad \text{(R-IfF)}\ \frac{[\![e]\!] = 0}{\textbf{if } e \textbf{ then } P \textbf{ else } Q \to_{\mathscr{D}} Q} \qquad \text{(R-Call)}\ \frac{[\![\widetilde{e}]\!] = \widetilde{v} \quad A(\widetilde{a}; \widetilde{x}) = P \in \mathscr{D}}{A(\widetilde{a}'; \widetilde{e}) \to_{\mathscr{D}} P[^{\widetilde{a}'}/_{\widetilde{a}}][^{\widetilde{v}}/_{\widetilde{x}}]}$$

$$\text{(R-Cong)}\ \frac{P \equiv P' \quad P' \to_{\mathscr{D}} Q' \quad Q' \equiv Q}{P \to_{\mathscr{D}} Q}$$

We often omit the subscript of $\to_{\mathscr{D}}$ when it is clear from the context. We write $\to^*$ for the reflexive and transitive closure of $\to$.

The deadlock-freedom of a process P, which is the basic property that we will verify, means that P does not get stuck into a state where there is a message or an input. The formal definition is below.

Definition 7 (deadlock-freedom). *A program $(\mathscr{D}, P)$ is **deadlock-free** if the following condition holds: whenever $P \to^* P'$ and either (i) $P' \equiv (\nu\widetilde{a}_1; x_1 : T_1) \cdots (\nu\widetilde{a}_k; x_k : T_k)(x!v \mid Q)$, or (ii) $P' \equiv (\nu\widetilde{a}_1; x_1 : T_1) \cdots (\nu\widetilde{a}_k; x_k : T_k)(x?y.Q_1 \mid Q_2)$, then there exists R such that $P' \to R$.*

Example 2 (The dining philosophers). Consider the program consisting of the process definitions

$$\begin{array}{l} \mathit{Phils}(a_1, a_2, a_3, a_4; n : \textbf{int}, \mathit{fork}_1 : U_1, \mathit{fork}_2 : U_2) = \\ \quad \textbf{if } n = 1 \textbf{ then } \mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1, \mathit{fork}_2) \textbf{ else} \\ \quad\quad (\nu\, a_5, a_6; \mathit{fork}_3 : U_3 \mid U_3 \mid !^{a_5}_{a_6})(\quad \mathit{Phils}(a_1, a_2, a_5, a_6; n-1, \mathit{fork}_1, \mathit{fork}_3) \\ \quad\quad\quad\quad \mid \quad \mathit{Phil}(a_5, a_6, a_3, a_4; \mathit{fork}_3, \mathit{fork}_2) \quad \mid \quad \mathit{fork}_3!1 \quad) \end{array}$$

$$\begin{array}{l} \mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1 : U_1, \mathit{fork}_2 : U_2) = \\ \quad \mathit{fork}_1?x_1.\mathit{fork}_2?x_2.(\, \mathit{fork}_1!x_1 \mid \mathit{fork}_2!x_2 \mid \mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1, \mathit{fork}_2)\,) \end{array}$$

and of the main process P:

$$\begin{array}{l} (\nu\, a_1, a_2; \mathit{fork}_1 : U_1 \mid U_1 \mid !^{a_1}_{a_2})(\nu\, a_3, a_4; \mathit{fork}_2 : U_2 \mid U_2 \mid !^{a_1}_{a_2}) \\ (\, \mathit{Phils}(a_1, a_2, a_3, a_4; m, \mathit{fork}_1, \mathit{fork}_2) \mid \mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1, \mathit{fork}_2) \mid \mathit{fork}_1!1 \mid \mathit{fork}_2!1\,) \end{array}$$

Here, $U_1 = \mu\alpha.?^{a_2}_{a_1}.(!^{a_1}_{a_2} \mid \alpha)$, $U_2 = \mu\alpha.?^{a_4}_{a_3}.(!^{a_3}_{a_4} \mid \alpha)$, and $U_3 = \mu\alpha.?^{a_6}_{a_5}.(!^{a_5}_{a_6} \mid \alpha)$, but please ignore the types for the moment. Every philosopher $\mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1, \mathit{fork}_2)$ grabs the two forks fork_1 and fork_2 in this order, releases the forks, and repeats the same behavior. The main process creates a ring consisting of $m + 1$ philosophers, where only one of the philosophers grabs the forks in the opposite order to avoid deadlock. This program is indeed deadlock-free in our definition. On the other hand, if we replace $\mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1, \mathit{fork}_2)$ with $\mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_2, \mathit{fork}_1)$ in the main process, then the resulting process is not deadlock-free. □

The dining philosophers example is a paradigmatic case of the power of the analysis described in the next section. This example cannot be type-checked in Kobayashi's previous type system [11]: see Remark 2 in Section 6.

6 The deadlock freedom analysis of value-passing CCS

We now explain the syntax of types in Figure 1. A type is either **int** or a usage. The former is used to type *integer* names; the latter is used to type *channel* names [11, 9]. A usage describes how a channel can be used for input and output. The usage $\mathbf{0}$ describes a channel that cannot be used, $!^{a_1}_{a_2}$ describes a channel that is used for output, $?^{a_1}_{a_2}.U$ describes a channel that is first used for input and then used according to U, and $U \mid U'$ describes a channel that is used according to U and U', possibly in parallel. For example, in process $x!2 \mid x?z.y!z$, y has the usage $!^{a_1}_{a_2}$ (please, ignore the subscript and superscript for the moment), and x has the usage $!^{a_3}_{a_4} \mid ?^{a_5}_{a_6}.\mathbf{0}$. The usage $\mu\alpha.U$ describes a channel that is used recursively according to $U[^{\mu\alpha.U}/_{\alpha}]$. The operation $\mu\alpha.-$ is a binder and we use the standard notions of alpha-equivalence, free and bound usage names. For example, $\mu\alpha.!^{a_1}_{a_2}.\alpha$ describes a channel that can be sequentially used for output an arbitrary number of times; $\mu\alpha.?^{a_1}_{a_2}.!^{a_3}_{a_4}.\alpha$ describes a channel that should be used for input and output alternately. We often omit a trailing $\mathbf{0}$ and just write $?^{a_1}_{a_1}$ for $?^{a_1}_{a_1}.\mathbf{0}$.

The superscripts and subscripts of ? and ! are level names of lams (recall Section 3), and are used to control the causal dependencies between communications [11]. The superscript, called an *obligation level*, describes the degree of the obligation to use the channel for the specified operation. The subscript, called a *capability level*, describes the degree of the capability to use the channel for the specified operation (and successfully find a partner of the communication).

In order to detect deadlocks we consider the following two conditions:

1. If a process has an obligation of level a, then it can exercise only capabilities of level a' *less than* a before fulfilling the obligation. This corresponds to a dependency (a', a). For example, if x has type $?^{a_1}_{a_2}$ and y has type $!^{a_3}_{a_4}$, then the process $x?u.y!u$ has lam (a_2, a_3).

2. The whole usage of each channel must be consistent, in the sense that if there is a capability of level a to perform an input (respectively, a message), there must be a corresponding obligation of level a to perform a corresponding message (respectively, input). For example, the usage $!^{a_1}_{a_2} \mid ?^{a_2}_{a_1}$ is consistent, but neither $!^{a_1}_{a_2} \mid ?^{a_1}_{a_2}$ nor $!^{a_1}_{a_2}$ is.

To see how the constraints above guide our deadlock analysis, consider the (deadlocked) process: $x?u.y!u \mid y?u.x!u$. Because of condition 2 above, the usage of x and y must be of the form $?^{a_1}_{a_2} \mid !^{a_2}_{a_1}$ and $?^{a_3}_{a_4} \mid !^{a_4}_{a_3}$ respectively. Due to 1, we derive (a_2, a_4) for $x?u.y!u$, and (a_4, a_2) for $y?u.x!u$. Hence the process is deadlocked because the lam $(a_2, a_4) \,\&\, (a_4, a_2)$ has a circularity. On the other hand, for the process $x?u.y!u \mid y?u.\mathbf{0} \mid x!u$, we derive the lam (a_2, a_4), which has no circularity. Indeed, this last process is not deadlocked. While we use lams to detect deadlocks, Kobayashi [11] used natural numbers for obligation/-capability levels.

As explained above, usages describe the channel-wise behavior of a process, and they form a tiny process calculus. The usage reduction relation $U \rightsquigarrow U'$ defined below means that if a channel of usage U is used for a communication, the channel may be used according to U' afterwards.

Definition 8. *Let = be the least congruence on usages containing alpha-conversion of bound names, commutativity and associativity of* $\mid$ *with identity* $\mathbf{0}$*, and closed under the*

following rule:

$$\text{(UC-Mu)}\quad \mu\alpha.U \;=\; U[^{\mu\alpha.U}/_{\alpha}]$$

The reduction relation $U \rightsquigarrow U'$ is the least relation closed under the rules:

$$\text{(UR-Com)}\quad !^{a_1}_{a_2} \mid ?^{a_3}_{a_4}.U \rightsquigarrow U \qquad \text{(UR-Par)}\quad \frac{U_1 \rightsquigarrow U_1'}{U_1 \mid U_2 \rightsquigarrow U_1' \mid U_2} \qquad \text{(UR-Cong)}\quad \frac{U_1 = U_1' \quad U_1' \rightsquigarrow U_2' \quad U_2' = U_2}{U_1 \rightsquigarrow U_2}$$

As usual, we let $\rightsquigarrow^*$ be the reflexive and transitive closure of $\rightsquigarrow$.

The following relation $rel(U)$ guarantees the condition 2 on capabilities and obligations above, that each capability must be accompanied by a corresponding obligation. This must hold during the whole computation, hence the definition below. The predicate $rel(U)$ is computable because it may be reduced to Petri Nets reachability (see [10] for the details about the encoding).

Definition 9. *U is **reliable**, written $rel(U)$, when the following conditions hold:*

1. *whenever $U \rightsquigarrow^* U'$ and $U' = !^{a_1}_{a_2} \mid U_1$, there are U_2 and U_3 such that $U_1 = ?^{a_2}_{a_3}.U_2 \mid U_3$ for some a_3; and*
2. *whenever $U \rightsquigarrow^* U'$ and $U' = ?^{a_1}_{a_2}.U_1 \mid U_2$, there is U_3 such that $U_2 = !^{a_2}_{a_3} \mid U_3$ for some a_3.*

The following type system uses *type environments*, ranged over $\Gamma, \Gamma', \cdots$, that map integer and channel names to types and process names to sequences $[\widetilde{a}; \widetilde{T}]$. When $x \notin dom(\Gamma)$, we write $\Gamma, x : T$ for the environment such that $(\Gamma, x : T)(x) = T$ and $(\Gamma, x : T)(y) = \Gamma(y)$, otherwise. The operation $\Gamma_1 \mid \Gamma_2$ is defined by:

$$(\Gamma_1 \mid \Gamma_2)(x) = \begin{cases} \Gamma_1(x) & \text{if } x \in dom(\Gamma_1) \text{ and } x \notin dom(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in dom(\Gamma_2) \text{ and } x \notin dom(\Gamma_1) \\ [\widetilde{a}; \widetilde{T}] & \text{if } \Gamma_1(x) = \Gamma_2(x) = [\widetilde{a}; \widetilde{T}] \\ \textbf{int} & \text{if } \Gamma_1(x) = \Gamma_2(x) = \textbf{int} \\ U_1 \mid U_2 & \text{if } \Gamma_1(x) = U_1 \text{ and } \Gamma_2(x) = U_2 \end{cases}$$

The map $\Gamma_1 \mid \Gamma_2$ is undefined if, for some x, $(\Gamma_1 \mid \Gamma_2)(x)$ does not match any of the cases. Let $var(\Gamma) = \{a \mid \text{there is } x : \Gamma(x) = U \text{ and } a \in var(U)\}$.

There are three kinds of type judgments:

$\Gamma \vdash e : T$ – the expression e has type T in Γ;
$\Gamma \vdash P : \mathtt{L}$ – the process P has lam $\mathtt{L}$ in Γ;
$\Gamma \vdash (\mathscr{D}, P) : (\mathscr{L}, \mathtt{L})$ – the program $(\mathscr{D}, P)$ has lam program $(\mathscr{L}, \mathtt{L})$ in Γ.

As usual, $\Gamma \vdash e : T$ means that e evaluates to a value of type T under an environment that respects the type environment Γ. The judgment $\Gamma \vdash P : \mathtt{L}$ means that P uses each channel x according to $\Gamma(x)$, with the causal dependency as described by $\mathtt{L}$. For example, $x{:}?^{a_1}_{a_2}, y{:}!^{a_3}_{a_4} \vdash x?u.y!u : (a_2, a_3)$ should hold.

The typing rules of value-passing CCS are defined in Figure 2, where we use the predicate $noact(\Gamma)$ and the function $ob(U)$ defined as follows:

Processes:

$$\text{(T-Zero)}\ \frac{noact(\Gamma)}{\Gamma \vdash \mathbb{0} : \mathbb{0}} \qquad \text{(T-Out)}\ \frac{\Gamma \vdash e : \mathbf{int}}{\Gamma, x{:}!^{a_1}_{a_2} \vdash x!e : \mathbb{0}} \qquad \text{(T-In)}\ \frac{\Gamma, x : U, y : \mathbf{int} \vdash P : \mathtt{L}}{\Gamma, x{:}?^{a_1}_{a_2}.U \vdash x?y.P : \mathtt{L} \,\&\, (\&_{a \in ob(\Gamma)}(a_2, a))}$$

$$\text{(T-Par)}\ \frac{\Gamma \vdash P : \mathtt{L} \qquad \Gamma' \vdash P' : \mathtt{L}'}{\Gamma \mid \Gamma' \vdash P \mid P' : \mathtt{L} \,\&\, \mathtt{L}'} \qquad \text{(T-New)}\ \frac{\Gamma, x : U \vdash P : \mathtt{L} \qquad rel(U) \qquad \widetilde{a} \cap var(\Gamma) = \emptyset}{\Gamma \vdash (\nu \widetilde{a}; x : U)P : (\nu \widetilde{a})\mathtt{L}}$$

$$\text{(T-If)}\ \frac{\Gamma \vdash e : \mathbf{int} \qquad \Gamma' \vdash P : \mathtt{L} \qquad \Gamma' \vdash P' : \mathtt{L}'}{\Gamma \mid \Gamma' \vdash \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ P' : \mathtt{L} + \mathtt{L}'} \qquad \text{(T-Call)}\ \frac{\Gamma(A) = [\widetilde{a}; \widetilde{\mathtt{T}}] \qquad |\widetilde{a}| = |\widetilde{a}'| \qquad \Gamma \vdash \widetilde{e} : \widetilde{\mathtt{T}}}{\Gamma \vdash A(\widetilde{a}'; \widetilde{e}) : \mathtt{f}_A(\widetilde{a}')}$$

Expressions:

$$\text{(T-Int)}\ \frac{noact(\Gamma)}{\Gamma \vdash n : \mathbf{int}} \qquad \text{(T-Var)}\ \frac{noact(\Gamma)}{\Gamma, x : \mathtt{T} \vdash x : \mathtt{T}} \qquad \text{(T-Op)}\ \frac{\Gamma \vdash e : \mathbf{int}\ \Gamma \vdash e' : \mathbf{int}}{\Gamma \vdash e\ \mathbf{op}\ e' : \mathbf{int}} \qquad \text{(T-Seq)}\ \frac{(\Gamma_i \vdash e_i : T_i)^{i \in 1..n}}{\Gamma_1 \mid \cdots \mid \Gamma_n \vdash e_1, \ldots, e_n : T_1, \ldots, T_n}$$

Programs:

$$\text{(T-Prog)}\ \frac{\begin{array}{c} \mathscr{D} = \bigcup_{i \in 1..n}\{A_i(\widetilde{a}_i; \widetilde{x}_i : \widetilde{\mathtt{T}}_i) = P_i\} \qquad \Gamma = (A_i : [\widetilde{a}_i; \widetilde{\mathtt{T}}_i])^{i \in 1..n} \\ (\Gamma, \widetilde{x}_i : \widetilde{\mathtt{T}}_i \vdash P_i : \mathtt{L}_i)^{i \in 1..n} \qquad \Gamma' \vdash P : \mathtt{L} \qquad \mathscr{L} = \bigcup_{i \in 1..n}\{\mathtt{f}_{A_i}(\widetilde{a}_i) = \mathtt{L}_i\} \end{array}}{\Gamma \mid \Gamma' \vdash (\mathscr{D}, P) : (\mathscr{L}, \mathtt{L})}$$

Fig. 2. The type system of value-passing CCS (we assume a function name $\mathtt{f}_A$ for every process name A)

$noact(\Gamma) = true$ if and only if, for every channel name $x \in dom(\Gamma)$, $\Gamma(x) = \mathbf{0}$;
$ob(\Gamma) = \bigcup_{x \in dom(\Gamma), \Gamma(x) = U} ob(U)$ where

$$ob(\mathbf{0}) = \emptyset \qquad ob(!^{a_1}_{a_2}) = \{a_1\} \qquad ob(?^{a_1}_{a_2}.U) = \{a_1\}$$
$$ob(U \mid U') = ob(U) \cup ob(U') \qquad ob(\mu\alpha.U) = ob(U[\mathbf{0}/_\alpha])$$

The predicate $noact(\Gamma)$ is used for controlling weakening (as in linear type systems). For example, if we did not require $noact(\Gamma)$ in rule T-Zero, then we would obtain $x{:}?^{a_1}_{a_2}.\mathbf{0} \vdash \mathbb{0} : \mathbb{0}$. Then, by using T-In and T-Out, we would obtain: $x{:}?^{a_1}_{a_2}.\mathbf{0} \mid !^{a_2}_{a_1} \vdash \mathbb{0} \mid x!1 : \mathbb{0}$, and wrongly conclude that the output on x does not get stuck. It is worth to notice that, in the typing rules, we identify usages up to =.

A few key rules are discussed. Rule (T-In) is the unique one that introduces dependency pairs. In particular, the process $x?u.P$ will be typed with a lam that contains pairs (a_2, a), where a_2 is the capability of x and a is the obligation of every channel in P (because they are all causally dependent from x). Rule (T-Out) just records in the type environment that x is used for output. Rule (T-Par) types a parallel composition of processes by collecting the environments – operation "$|$" – (like in other linear type systems [13, 9]) and the lams of the components. Rule (T-Call) types a process name invocation in terms of a (lam) function invocation and constrains the sequences of level names in the two invocations to have equal lengths ($|\widetilde{a}| = |\widetilde{a}'|$) and the types of expressions to match with the types in the process declaration.

Example 3. We illustrate the type system in Figure 2 by typing two simple processes:

$$P = (\nu\, a_1, a_2; x{:}?^{a_1}_{a_2} \mid !^{a_2}_{a_1})(\nu\, a_3, a_4; y{:}?^{a_3}_{a_4} \mid !^{a_4}_{a_3})(x?z.y!z \mid y?z.x!z)$$
$$Q = (\nu\, a_1, a_2; x{:}?^{a_1}_{a_2} \mid !^{a_2}_{a_1})(\nu\, a_3, a_4; y{:}?^{a_3}_{a_4} \mid !^{a_4}_{a_3})(x?z.y!z \mid y?z.\mathbf{0} \mid x!1)$$

The proof tree of P is

$$\dfrac{\dfrac{\dfrac{y{:}!^{a_4}_{a_3},\ z : \mathbf{int} \vdash y!z : \mathbf{0}}{x{:}?^{a_1}_{a_2},\ y{:}!^{a_4}_{a_3} \vdash x?z.y!z : (a_2, a_4)} \qquad \dfrac{x{:}!^{a_2}_{a_1},\ z : \mathbf{int} \vdash x!z : \mathbf{0}}{x{:}!^{a_2}_{a_1},\ y{:}?^{a_3}_{a_4} \vdash y?z.x!z : (a_4, a_2)}}{x{:}?^{a_1}_{a_2} \mid !^{a_2}_{a_1},\ y{:}?^{a_3}_{a_4} \mid !^{a_4}_{a_3} \vdash x?z.y!z \mid y?z.x!z : (a_2, a_4) \,\&\, (a_4, a_2)}}{\varnothing \vdash P : (\nu\, a_1, a_2)(\nu\, a_3, a_4)((a_2, a_4) \,\&\, (a_4, a_2))}$$

and we notice that the lam in the conclusion has a circularity (in fact, P is deadlocked). The typing of Q is

$$\dfrac{\dfrac{\dfrac{\dfrac{z : \mathbf{int} \vdash z : \mathbf{int}}{y{:}!^{a_4}_{a_3},\ z : \mathbf{int} \vdash y!z : \mathbf{0}}}{x{:}?^{a_1}_{a_2},\ y{:}!^{a_4}_{a_3} \vdash x?z.y!z : (a_2, a_4)} \qquad \dfrac{y : \mathbf{0},\ z : \mathbf{int} \vdash \mathbf{0} : \mathbf{0}}{y{:}?^{a_3}_{a_4} \vdash y?z.\mathbf{0} : \mathbf{0}} \qquad \dfrac{\varnothing \vdash 1 : \mathbf{int}}{x{:}!^{a_2}_{a_1} \vdash x!1 : \mathbf{0}}}{x{:}?^{a_1}_{a_2} \mid !^{a_2}_{a_1},\ y{:}?^{a_3}_{a_4} \mid !^{a_4}_{a_3} \vdash x?z.y!z \mid y?z.\mathbf{0} \mid x!1 : (a_2, a_4)}}{\varnothing \vdash Q : (\nu\, a_1, a_2)(\nu\, a_3, a_4)(a_2, a_4)}$$

The lam in the conclusion has no circularity. In fact, Q is not deadlocked. □

Example 3 also spots one difference between the type system in [11] and the one in Figure 2. Here the inter-channel dependencies check is performed *ex-post* by resorting to the lam algorithm in Section 4; in [11] this check is done *during* the type checking(/inference) and, for this reason, the process P is not typable in previous Kobayashi's type systems. In this case, the two analysers both recognize that P is deadlocked; Example 4 below discusses a case where the precision is different.

The following theorem states the soundness of our type system.

Theorem 2. *Let $\Gamma \vdash (\mathscr{D}, P) : (\mathscr{L}, \mathtt{L})$ such that noact(Γ). If $(\mathscr{L}, \mathtt{L})$ has no circularity then $(\mathscr{D}, P)$ is deadlock-free.*

The following examples highlight the difference of the expressive power of the system in Figure 2 and the type system in [11].

Example 4. Let $(\mathscr{D}, P)$ be the dining philosopher program in Example 2 and U_1 and U_2 be the usages defined therein. We have $\Gamma \vdash (\mathscr{D}, P) : (\mathscr{L}, \mathtt{L})$ where

$$\begin{aligned}
\Gamma &= \mathit{Phils} : [a_1, a_2, a_3, a_4; \mathbf{int}, U_1, U_2], \mathit{Phil} : [a_1, a_2, a_3, a_4; U_1, U_2] \\
\mathscr{L} &= \{\ \mathtt{f}_{\mathit{Phils}}(a_1, a_2, a_3, a_4) = \mathtt{f}_{\mathit{Phil}}(a_1, a_2, a_3, a_4) \\
&\qquad\qquad +(\nu a_5, a_6)(\mathtt{f}_{\mathit{Phils}}(a_1, a_2, a_5, a_6) \,\&\, \mathtt{f}_{\mathit{Phil}}(a_5, a_6, a_3, a_4)), \\
&\qquad \mathtt{f}_{\mathit{Phil}}(a_1, a_2, a_3, a_4) = (a_1, a_4) \,\&\, (a_3, a_1) \,\&\, (a_3, a_2) \,\&\, \mathtt{f}_{\mathit{Phil}}(a_1, a_2, a_3, a_4)\ \} \\
\mathtt{L} &= (\nu a_1, a_2, a_3, a_4)(\mathtt{f}_{\mathit{Phils}}(a_1, a_2, a_3, a_4) \,\&\, \mathtt{f}_{\mathit{Phil}}(a_1, a_2, a_3, a_4))
\end{aligned}$$

For example, let

$$\begin{aligned}
P_1 &= \mathit{fork}_1?x_1.\mathit{fork}_2?x_2.(\,\mathit{fork}_1!x_1 \mid \mathit{fork}_2!x_2 \mid \mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1, \mathit{fork}_2)\,) \\
P_2 &= \mathit{fork}_2?x_2.(\,\mathit{fork}_1!x_1 \mid \mathit{fork}_2!x_2 \mid \mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1, \mathit{fork}_2)\,) \\
P_3 &= \mathit{fork}_1!x_1 \mid \mathit{fork}_2!x_2 \mid \mathit{Phil}(a_1, a_2, a_3, a_4; \mathit{fork}_1, \mathit{fork}_2)
\end{aligned}$$

Then the body P_1 of *Phil* is typed as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma_2, fork_1{:}!^{a_1}_{a_2} \vdash fork_1!x_1 : \mathbb{0} \qquad \Gamma_2, fork_2{:}!^{a_3}_{a_4} \vdash fork_2!x_2 : \mathbb{0} \quad \Gamma_2, fork_1 : U_1, fork_2 : U_2 \vdash Phil(a_1,a_2,a_3,a_4; fork_1, fork_2) : \mathtt{f}_{Phil}(a_1,a_2,a_3,a_4)}{\Gamma_2, fork_1{:}!^{a_1}_{a_2} \mid U_1, fork_2{:}!^{a_3}_{a_4} \mid U_2 \vdash P_3 : \mathtt{f}_{Phil}(a_1,a_2,a_3,a_4)}}{\Gamma_1, fork_1{:}!^{a_1}_{a_2} \mid U_1, fork_2 : U_2 \vdash P_2 : (a_3,a_1) \,\&\, (a_3,a_2) \,\&\, \mathtt{f}_{Phil}(a_1,a_2,a_3,a_4)}}{\Gamma, fork_1 : U_1, fork_2 : U_2 \vdash P_1 : (a_1,a_4) \,\&\, (a_3,a_1) \,\&\, (a_3,a_2) \,\&\, \mathtt{f}_{Phil}(a_1,a_2,a_3,a_4)}$$

where $\Gamma_1 = \Gamma, x_1 : \mathbf{int}$, $\Gamma_2 = \Gamma, x_2 : \mathbf{int}$, $U_1 = \mu\alpha.?^{a_2}_{a_1}.(!^{a_1}_{a_2} \mid \alpha)$ and $U_2 = \mu\alpha.?^{a_4}_{a_3}.(!^{a_3}_{a_4} \mid \alpha)$. Because $(\mathscr{L}, \mathtt{L})$ has no circularity, by Theorem 2, we can conclude that $(\mathscr{D}, P)$ is deadlock-free. □

Remark 2. The dining philosopher program cannot be typed in Kobayashi's type system [11]. That is because his type system assigns obligation/capability levels to each input/output *statically*. Thus only a fixed number of levels (represented as natural numbers) can be used to type a process in his type system. Since the above process can create a network consisting of an arbitrary number of dining philosophers, we need an unbounded number of levels to type the process. (Kobayashi [11] introduced a heuristic to partially mitigate the restriction on the number of levels being fixed, but the heuristic does not work here.) A variant of the dining philosopher example has been discussed in [8]. Since the variant is designed so that a finite number of levels are sufficient, it is typed both in [11] and in our new type system.

Similarly to the dining philosopher program, the system in [11] returns a false positive for the process `Fact` in Section 1, while it is deadlock-free according to our new system. We detail the arguments in the next example.

Example 5. Process `Fact` of Section 1 is written in the value passing CCS as follows.

$$\begin{array}{l}
\mathtt{Fact}(a_1,a_2,a_3,a_4; n : \mathbf{int}, r{:}?^{a_1}_{a_2}, s{:}!^{a_3}_{a_4}) = \\
\quad \mathbf{if}\ n = 0\ \mathbf{then}\ r?n.s!n\ \mathbf{else} \\
\qquad (\nu\, a_5, a_6; t{:}?^{a_5}_{a_6} \mid !^{a_6}_{a_5})(r?n.t!(m \times n) \mid \mathtt{Fact}(a_5,a_6,a_3,a_4; n-1, t, s))
\end{array}$$

Let $\Gamma = \mathtt{Fact} : [a_1,a_2,a_3,a_4; \mathbf{int}, ?^{a_1}_{a_2}, !^{a_3}_{a_4}]$ and P be the body of the definition above. Then we have $\Gamma, n : \mathbf{int}, r{:}?^{a_1}_{a_2}, s{:}!^{a_3}_{a_4} \vdash P : \mathtt{L}$ for $\mathtt{L} = (a_2,a_3) + (\nu\, a_5,a_6)((a_2,a_6) \,\&\, \mathtt{f}_{\mathtt{Fact}}(a_5,a_6,a_3,a_4))$. Thus, we have: $\Gamma \vdash (\mathscr{D}, P'){:}(\mathscr{L}, \mathtt{L}')$ for:

$$\begin{array}{l}
P' = (\nu\, a_1,a_2; r{:}?^{a_1}_{a_2} \mid !^{a_2}_{a_1})(\nu\, a_3,a_4; s{:}?^{a_4}_{a_3} \mid !^{a_3}_{a_4})(r!1 \mid \mathtt{Fact}(a_1,a_2,a_3,a_4; m,r,s) \mid s?x.\mathbf{0}) \\
\mathscr{L} = \{\mathtt{f}_{\mathtt{Fact}}(a_1,a_2,a_3,a_4) = \mathtt{L}\} \\
\mathtt{L}' = (\nu\, a_1,a_2,a_3,a_4)(\mathbb{0} \,\&\, \mathtt{f}_{\mathtt{Fact}}(a_1,a_2,a_3,a_4) \,\&\, \mathbb{0})
\end{array}$$

where m is an integer constant. Since $(\mathscr{L}, \mathtt{L}')$ does not have a circularity, we can conclude that $(\mathscr{D}, P')$ is deadlock-free.

6.1 Proof of Theorem 2

Let $\Gamma \rightsquigarrow \Gamma'$ if, for some x, $\Gamma = \Gamma'', x : U$ and $\Gamma' = \Gamma'', x : U'$ with $U \rightsquigarrow U'$. As usual, let $\rightsquigarrow^*$ be the transitive closure of $\rightsquigarrow$.

Theorem 2 follows from the following lemmas.

Lemma 4 (type preservation). *Let $\Gamma \vdash (\mathscr{D}, P) : (\mathscr{L}, \mathtt{L})$ and $P \to_{\mathscr{D}} Q$. Then there exist $\mathtt{L}'$ and Γ' such that $\Gamma \rightsquigarrow^* \Gamma'$ and $\Gamma' \vdash (\mathscr{D}, Q) : (\mathscr{L}, \mathtt{L}')$ and $I_{\mathscr{L}}(\mathtt{L}') \Subset I_{\mathscr{L}}(\mathtt{L})$.*

Proof. This follows by induction on the derivation of $P \to_{\mathscr{D}} Q$, with case analysis on the last rule used. The only non-trivial cases are R-Com and R-Call.

– Case R-Com: In this case, we have

$$\begin{array}{l} P = x!e \mid x?y.P' \qquad Q = P'[^{v}/_{y}] \qquad [\![e]\!] = v \\ \Gamma = \Gamma_0 \mid \Gamma_1 \qquad \mathscr{D} = \bigcup_{i\in 1..n}\{A_i(\widetilde{a_i}; \widetilde{x_i} : \widetilde{\mathtt{T}_i}) = P_i\} \\ (\Gamma_0, \widetilde{x_i} : \widetilde{\mathtt{T}_i} \vdash P_i : \mathtt{L}_i)^{i\in 1..n} \qquad \Gamma_1 \vdash P : \mathtt{L} \end{array}$$

By the conditions $\Gamma_1 \vdash P : \mathtt{L}$ and $P = x!e \mid x?y.P'$, we have

$$\begin{array}{l} \Gamma_1 = (\Gamma_2 \mid \Gamma_3), x{:}!^{a_1}_{a_2} \mid ?^{a_3}_{a_4}.U \\ \Gamma_2 \vdash e : \mathbf{int} \qquad \Gamma_3, x : U, y : \mathbf{int} \vdash P' : \mathtt{L}' \\ \mathtt{L} = \mathbb{0} \,\&\, (\mathtt{L}' \,\&\, (\&_{a\in ob(\Gamma_3)}(a_2, a))) \end{array}$$

By the condition $\Gamma_2 \vdash e : \mathbf{int}$ and $[\![e]\!] = v$, we have $\Gamma_2 \vdash v : \mathbf{int}$. By using the standard substitution lemma, we obtain $\Gamma_2 \mid \Gamma_3, x : U \vdash P'[^{v}/_{y}] : \mathtt{L}'$. Let $\Gamma' = \Gamma_0 \mid ((\Gamma_2 \mid \Gamma_3), x : U)$. Then we have the required result.

– Case R-Call: In this case, we have

$$\begin{array}{l} P = A_i(\widetilde{a'}; \widetilde{e}) \qquad Q = P_i[^{\widetilde{a'}}/_{\widetilde{a}}][^{\widetilde{v}}/_{\widetilde{x}}] \qquad [\![\widetilde{e}]\!] = \widetilde{v} \\ \Gamma = \Gamma_0 \mid \Gamma_1 \qquad \mathscr{D} = \bigcup_{i\in 1..n}\{A_i(\widetilde{a_i}; \widetilde{x_i} : \widetilde{\mathtt{T}_i}) = P_i\} \\ (\Gamma_0, \widetilde{x_i} : \widetilde{\mathtt{T}_i} \vdash P_i : \mathtt{L}_i)^{i\in 1..n} \qquad \Gamma_1 \vdash \widetilde{e} : \widetilde{\mathtt{T}_i} \qquad \mathtt{L} = \mathtt{f}_{A_i}(\widetilde{a'}) \end{array}$$

By the conditions $\Gamma_1 \vdash \widetilde{e} : \widetilde{\mathtt{T}_i}$ and $[\![\widetilde{e}]\!] = \widetilde{v}$, we have $\Gamma_1 \vdash \widetilde{v} : \widetilde{\mathtt{T}_i}$. Thus, by applying the standard substitution lemma to $\Gamma_0, \widetilde{x_i} : \widetilde{\mathtt{T}_i} \vdash P_i : \mathtt{L}_i$, we obtain $\Gamma_0 \mid \Gamma_1 \vdash Q : \mathtt{L}_i[^{\widetilde{a'}}/_{\widetilde{a_i}}]$. We have the required result for $\Gamma' = \Gamma_1$ and $\mathtt{L}' = \mathtt{L}_i[^{\widetilde{a'}}/_{\widetilde{a_i}}]$. □

Lemma 5. *Let $\Gamma \vdash (\mathscr{D}, P) : (\mathscr{L}, \mathtt{L})$ such that $I_{\mathscr{L}}(\mathtt{L})$ has no circularity and noact(Γ). If*

1. *either $P \equiv (\nu\widetilde{a}_1; x_1 : U_1) \cdots (\nu\widetilde{a}_k; x_k : U_k)(x!v \mid Q)$*
2. *or $P \equiv (\nu\widetilde{a}_1; x_1 : U_1) \cdots (\nu\widetilde{a}_k; x_k : U_k)(x?y.P' \mid Q)$*

then there exists R such that $P \to_{\mathscr{D}} R$.

Proof. If Q contains conditionals or process calls at the top level, then the required property immediately follows. Thus, we can assume that Q is a parallel composition of inputs and messages. That is

$$Q \equiv y_1!e_1 \mid \cdots \mid y_m!e_m \mid z_1?w_1.Q_1 \mid \cdots \mid z_n?w_n.Q_n$$

where $\{x, y_1, \cdots, y_m, z_1, \cdots, z_n\} \subseteq \{x_1, \cdots, x_k\}$ because *noact*(Γ).

We demonstrate the case 1 of the statement, the other case is similar, hence omitted.

Since $\Gamma \vdash (\mathscr{D}, P) : (\mathscr{L}, \mathtt{L})$ then there is Γ' such that $\Gamma' \vdash (\nu\widetilde{a}_1; x_1 : U_1) \cdots (\nu\widetilde{a}_k; x_k : U_k)(x!e \mid Q) : \mathtt{L}$. By applying k-times rule (T-New), we are reduced to $\Gamma', x_1 : U_1, \cdots x_k : U_k \vdash x!e \mid Q : \mathtt{L}$. Let $x = x_1$. We notice that $n \geq 1$ because, by *rel*(U_1), $U_1 = !^{a}_{a''} \mid ?^{a'}_{a}.U'_1 \mid U''_1$ and by (T-Par) and definition of $\Gamma \mid \Gamma'$, $Q \equiv z?w.Q' \mid Q''$ for some z, w, Q', Q''.

By the typing rules, $\Gamma', x_1 : U_1, \cdots x_k : U_k \vdash x!e \mid Q : \mathtt{L}$ is possible provided, for every $1 \leq i \leq n$:

$$\Gamma'_i, z_i{:}?^{a'_i}_{a_i} \vdash z_i?w_i.Q_i : \mathtt{L}_i \,\&\, (\&_{a \in ob(\Gamma'_i)}(a_i, a))$$

where $\Gamma'_i = \Gamma', x_1 : U^i_1, \cdots x_k : U^i_k$ and $\Gamma', x_1 : U_1, \cdots x_k : U_k = \Gamma'_1, z_1{:}?^{a'_1}_{a_1} \mid \cdots \mid \Gamma'_n, z_n{:}?^{a'_n}_{a_n}$ and $\mathtt{L} = \&_{i \in 1..n}(\mathtt{L}_i \,\&\, (\&_{a \in ob(\Gamma'_i)}(a_i, a)))$.

We notice that $\mathtt{L} \equiv (\&_{i \in 1..n}\mathtt{L}_i) \,\&\, \mathtt{L}'$, with $\mathtt{L}' = (\&_{i \in 1..n, a \in ob(\Gamma'_i)}(a_i, a))$. Since $I^{\perp}(\mathtt{L})$ has no circularity then, by Proposition 2, $I^{\perp}(\&_{i \in 1..n}\mathtt{L}_i) \,\&\, I^{\perp}(\mathtt{L}')$ has no circularity and, in turn, $I^{\perp}(\mathtt{L}') = \{\mathtt{R}_{\mathtt{L}'}\}$ and $\mathtt{R}_{\mathtt{L}'}$, where $\mathtt{R}_{\mathtt{L}'} = \bigcup_{i \in 1..n}\{(a_i, a) \mid a \in ob(\Gamma'_i)\}$, have no circularity.

Let a_j, with $j \in 1..n$, be a *minimal level* of $\mathtt{R}_{\mathtt{L}'}$, namely:

- there is no a' such that $(a', a_j) \in \mathtt{R}_{\mathtt{L}'}$.

Because $\mathtt{R}_{\mathtt{L}'}$ has no circularity, a_j does exist and, without loss of generality, let $z_j = x_j$. By rule (T-In), $U_j = ?^{a'_j}_{a_j}.U'_j \mid U''_j$. By $rel(U_j)$ we derive $U''_j = !^{a_j}_{a''_j} \mid U'''_j$, for some a''_j and U'''_j. By (T-Par) and the fact that a_j is minimal in $\mathtt{R}_{\mathtt{L}'}$, we immediately derive that there exists $1 \leq i \leq m$ such that $y_i = x_j$, thus we have $P \rightarrow_{\mathscr{D}} R$ for some R, as required. □

Type inference An *untyped* value-passing CCS program is a program where restrictions are $(\nu\, x)P$, process invocations are $A(\widetilde{e})$ and process definitions are $A(\widetilde{x}) = P$. Given an untyped value-passing CCS program $(\mathscr{D}, P)$, with $var(P) = \emptyset$, there is an inference algorithm to decide whether there exists a program $(\mathscr{D}', P')$ that coincides with $(\mathscr{D}, P)$, except for the type annotations, and such that $\Gamma \vdash (\mathscr{D}', P') : (\mathscr{L}, \mathtt{L})$. The algorithm is almost the same as that of the type system in [10] and, therefore, we do not re-describe it here. The only extra work compared with the previous algorithm is the lam program extraction, which is done using the rules in Figure 2. Finally, it suffices to analyze the extracted lams by using the fixpoint technique in Section 4.

Synchronous value passing CCS and pi calculus The type system above can be easily extended to the pi-calculus, where channel names can be passed around through other channels. To that end, we extend the syntax of types as follows.

$$\mathtt{T} ::= \mathbf{int} \mid \mathsf{ch}(\mathtt{T}, U).$$

The type $\mathsf{ch}(\mathtt{T}, U)$ describes a channel that is used according to the usage U, and $\mathtt{T}$ is the type of values passed along the channel. Only a slight change of the typing rules is sufficient, as summarized below.

(T-Out')
$$\frac{\Gamma \vdash e : \mathtt{T}}{\Gamma, x : \mathsf{ch}(\mathtt{T}, !^{a_1}_{a_2}) \vdash x!e : \&_{a \in ob(\Gamma)}(a_2, a)}$$

(T-In')
$$\frac{\Gamma, x : \mathsf{ch}(\mathtt{T}, U), y : \mathtt{T} \vdash P : \mathtt{L}}{\Gamma, x : \mathsf{ch}(\mathtt{T}, ?^{a_1}_{a_2}.U) \vdash x?y.P : \mathtt{L} \,\&\, (\&_{a \in ob(\Gamma)}(a_2, a))}$$

In particular, (T-Out') introduces dependencies between an output channel and the values sent along the channel. We notice that, in case of *synchronous value passing CCS* (as well as pi-calculus), where messages have continuations, rule (T-Out') also introduces dependency pairs between the capability of the channel and the obligations in the continuation.

7 Related Work and Conclusions

In this paper we have designed a new deadlock detection technique for the value-passing CCS (and for the pi-calculus) that enables the analysis of networks with arbitrary numbers of nodes. Our technique relies on a decidability result of a basic model featuring recursion and fresh name generation: the *lam programs*. This model has been introduced and studied in [5, 6] for detecting deadlock of an object-oriented programming language [7], but the decidability was known only for a subset of lams where only linear recursion is allowed [6], and only approximate algorithms have been given for the full lam model.

The application of the lam model to deadlock-freedom of the value-passing CCS (and pi-calculus) is also new, and the resulting deadlock-freedom analysis significantly improves the previous deadlock-freedom analysis [11], as demonstrated through the dining philosopher example. In particular, Kobayashi's type system provides a mechanism for dealing with a limited form of unbounded dependency chains, but the mechanism is rather ad hoc and fragile with respect to a syntactic change. For example, while

```
Fib(n,r) =  if n<2 then r?n else new s in new t in
                  (Fib!(n-1,s) | s?x.(Fib!(n-2,t)|t?y.r!(x+y))
```

is typable, the variation obtained by swapping new s in and new t in is untypable. Neither Fact nor the dining philosopher example are typable in [11]. More recently, in [17], Padovani has introduced another type system for deadlock-freedom, which has a better support than Kobayashi's one for reasoning about unbounded dependency chains, by using a form of polymorphism on levels. However, since the levels in his type system are also integers, neither the Fact example nor the dining philosopher example are typable. In addition, Padovani's type system cannot deal with non-linear channels, like the fork channels in the dining philosopher example. That said, our type system does not subsume Padovani's one, as our system does not support recursive types.

Like other type-based analyses, our method cannot reason about value-dependent behaviors. For example, consider the following process:

$$(\textbf{if } b \textbf{ then } x?z.y!z \textbf{ else } y!1 \mid x?z.) \mid (\textbf{if } b \textbf{ then } x!1 \mid y?z. \textbf{ else } y?z.x!z).$$

It is deadlock-free, but our type system would extract the lam expression: $((a_x, a_y) + \mathbb{0}) \,\&\, (\mathbb{0} + (a_y, a_x))$ (where a_x and a_y are the capability levels of the inputs on x and y respectively), detecting a (false) circular dependency.

The integration of `TyPiCal` with the deadlock detection technique of this paper is left for future work. We expect that we can extend our analysis to cover lock-freedom [8, 17], too. To that end, we can require that a lam is not only circularity-free but is also *well founded*, and/or combine the deadlock-freedom analysis with the termination analysis, following the technique in [14].

Acknowledgments This work was partially supported by JSPS Kakenhi 23220001 and by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *TOPLAS*, 28, 2006.

[2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe program.: preventing data races and deadlocks. In *OOPSLA*, pages 211–230. ACM, 2002.

[3] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[4] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349. ACM, 2003.

[5] E. Giachino and C. Laneve. A beginner's guide to the deadLock Analysis Model. In *TGC'2012*, volume 8191 of *LNCS*, pages 49–63. Springer-Verlag, 2013.

[6] E. Giachino and C. Laneve. Deadlock detection in linear recursive programs. In *Proceedings of SFM-14:ESM*, volume 8483 of *LNCS*, pages 26–64. Springer-Verlag, 2014.

[7] E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in ABS. *Software and System Modeling*, To appear, 2014.

[8] N. Kobayashi. A type system for lock-free processes. *Information and Computation*, 177:122–159, 2002.

[9] N. Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *LNCS*, pages 439–453. Springer, 2003.

[10] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.

[11] N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

[12] N. Kobayashi. `TyPiCal`: Type-based static analyzer for the Pi-Calculus. At `kb.ecei.tohoku.ac.jp/~koba/typical/`, 2007.

[13] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[14] N. Kobayashi and D. Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM TOPLAS*, 32(5), 2010.

[15] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

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

[17] L. Padovani. Deadlock and Lock Freedom in the Linear π-Calculus. In *CSL-LICS'14*, 2014.

[18] K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *APLAS*, volume 5356 of *LNCS*, pages 155–170. Springer, 2008.

[19] V. T. Vasconcelos, F. Martins, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *PLACES*, volume 17 of *EPTCS*, pages 95–109, 2009.

Appendix B

Representation of Behavioural Interfaces for Concurrent Objects communicating by Asynchronous Method Calls and Futures

Journal of Logical and Algebraic Methods in Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

A sound and complete reasoning system for asynchronous communication with shared futures ☆

Crystal Chang Din, Olaf Owe *

Dept. of Informatics – Univ. of Oslo, P.O. Box 1080 Blindern, N-0316 Oslo, Norway

ARTICLE INFO

Article history:
Received 1 March 2013
Received in revised form 20 February 2014
Accepted 24 March 2014
Available online xxxx

Keywords:
Distributed systems
Compositional reasoning
Hoare Logic
Concurrent objects
Operational semantics
Communication history

ABSTRACT

Distributed and concurrent object-oriented systems are difficult to analyze due to the complexity of their concurrency, communication, and synchronization mechanisms. We consider the setting of concurrent objects communicating by *asynchronous method calls*. The *future mechanism* extends the traditional method call communication model by facilitating sharing of references to futures. By assigning method call result values to futures, third party objects may pick up these values. This may reduce the time spent waiting for replies in a distributed environment. However, futures add a level of complexity to program analysis, as the program semantics becomes more involved.

This paper presents a Hoare style reasoning system for distributed objects based on a general concurrency and communication model focusing on asynchronous method calls and futures. The model facilitates invariant specifications over the locally visible communication history of each object. Compositional reasoning is supported, and each object may be specified and verified independently of its environment. The presented reasoning system is proven sound and (relatively) complete with respect to the given operational semantics.

© 2014 Elsevier Inc. All rights reserved.

1. Introduction

Distributed systems play an essential role in society today. For example, distributed systems form the basis for critical infrastructure in different domains such as finance, medicine, aeronautics, telephony, and Internet services. It is of great importance that such systems work properly. However, quality assurance of distributed systems is non-trivial since they depend on unpredictable factors, such as different processing speeds of independent components. It is highly challenging to test such distributed systems after deployment under different relevant conditions. These challenges motivate frameworks combining precise modeling and analysis with suitable tool support. In particular, *compositional verification systems* allow the different components to be analyzed independently from their surrounding components. Thereby, it is possible to deal with systems consisting of many components.

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [1]. However, method-based communication between concurrent units may cause busy-waiting, as in the case of remote and synchronous method invocation, e.g., Java RMI [2]. Concurrent objects communicating by *asynchronous method calls*, which allows the caller to continue with its own activity without blocking while waiting for the reply, combine object-orientation

☆ This work was done in the context of the EU project FP7-610582 *Envisage: Engineering Virtualized Services* (http://www.envisage-project.eu) and FP7-ICT-2013-X *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations* (http://www.upscale-project.eu).

* Corresponding author.
E-mail addresses: crystald@ifi.uio.no (C.C. Din), olaf@ifi.uio.no (O. Owe).

http://dx.doi.org/10.1016/j.jlamp.2014.03.003
2352-2208/© 2014 Elsevier Inc. All rights reserved.

and distribution in a natural manner, and therefore appear as a promising paradigm for distributed systems [3]. Moreover, the notion of *futures* [4–7] improves this setting by providing a decoupling of the process invoking a method and the process reading the returned value. By sharing *future identities*, the caller enables other objects to wait for method results.

ABS is a high-level imperative object-oriented modeling language, based on the concurrency and synchronization model of *Creol* [8]. It supports futures and concurrent objects with an asynchronous communication model suitable for loosely coupled objects in a distributed setting. In *ABS*, each concurrent object encapsulates its own state and processor, and internal interference is avoided as at most one process is executing. The concurrent object model of *ABS* without futures supports compositionality because there is *no direct access* to the internal state variables of other objects, and a method call leads to a new process on the called object. With futures, compositionality is more challenging.

In this paper, we consider the general communication model of *ABS* focusing on the future mechanism. A compositional reasoning system for *ABS* with futures has been presented in [9] based on local communication histories. We here present a revised and simplified version of this system and show that it is sound with respect to an operational semantics which incorporates a notion of global communication history. We also show a completeness result.

The execution of a distributed system can be represented by its *communication history* or *trace*; i.e., the sequence of observable communication events between system components [10,11]. At any point in time the communication history abstractly captures the system state [12,13]. In fact, traces are used in the semantics for full abstraction results (e.g., [14,15]). The *local history* of an object reflects the communication visible to that object, i.e., between the object and its surroundings. A system may be specified by the finite initial segments of its communication histories, and a *history invariant* is a predicate which holds for all finite sequences in the set of possible histories, expressing safety properties [16].

In our reasoning system, we formalize object communication by an operational semantics based on five kinds of communication events, capturing shared (first class) futures, where each event is visible to only one object. Consequently, the local histories of two different objects share no common events. For each object, a history invariant can be derived from the class invariant by hiding the local state of the object. Modularity is achieved since history invariants can be established independently for each object, without interference, and composed at need. This results in behavioral specifications of dynamic system in an open environment. Such specifications allow objects to be specified independently of their internal implementation details, such as the internal state variables. In order to derive a global specification of a system composed of several components, one may compose the specification of different components. Global specifications can then be provided by describing the observable communication history between each component and its environment.

The main contribution of this paper is the presentation of a set of Hoare rules and a proof of soundness and relative completeness with respect to a revised operational semantics including a global communication history. The operational semantics is implemented in Maude by rewriting rules and can be exploited as an executable interpreter for the language, such that execution traces can be automatically generated while simulating programs. In earlier work [17], a similar proof system is derived from a standard sequential language by means of a syntactic encoding. However, soundness with respect to the operational semantics was not considered. A challenge of the current work is that the presence of the global history and shared futures complicate compositional reasoning and also the soundness and completeness proof. We therefore focus the current work on the core communication model with futures and consider process suspension mechanism, but ignore other aspects such as inheritance. The work is relevant for the more general setting of concurrent objects with asynchronous methods and futures, and it can easily be extended to the full *ABS* setting.

An *ABS* reasoning system is currently being implemented within the KeY framework at Technische Universität Darmstadt. The tool support from KeY for (semi-)automatic verification is valuable for verifying *ABS* programs. A publisher–subscriber example will be used here to illustrate the language and the reasoning system.

Paper overview. Section 2 introduces and explains the core language syntax, Section 3 formalizes the observable behavior in the distributed systems, Section 4, presents the operational semantics, and Section 5 defines the proof system for local reasoning within classes and finally considers object composition. A publisher–subscriber example is presented in Section 2 and the corresponding proofs are shown in Section 6. Section 7 defines and proves soundness and relative completeness for our reasoning system. Section 8 shows how to extend the language with non-blocking queries on futures. Section 9 discusses the relevance of choices made in the considered language and formalization, and briefly discusses how some other approaches may affect the reasoning system. Section 10 discusses related and future work, and Section 11 concludes the paper.

2. A core language with shared futures

We consider concurrent objects interacting through method calls. Class instances are concurrent, encapsulating their own state and processor. Each method invoked on the object leads to a new process, and at most one process is executing on an object at a time. Object communication is *asynchronous*, as there is no explicit transfer of control between the caller and the callee. In this setting a *future* represents a placeholder for the return value of a method call. Each future has a unique identity which is *generated* when the method is invoked, and a futures may be seen as a shared entity of information accessible by any object that knows its identity. A future is *resolved* upon method termination, by placing the return value of the method in the future. Thus, unlike the traditional method call mechanism, the callee does not send the return value directly back to the caller. However, the caller may keep a *reference* to the future, allowing the caller to *fetch* the future value

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

In	::=	**interface** I [**extends** I^+]$^?$ $\{S^*\}$	interface declaration
Cl	::=	**class** C([$T\,cp$]*) [**implements** I^+] $\{$[$T\,w$ [:= e]$^?$]* [s]$^?$ $M^*\}$	class definition
M	::=	$S\ B$	method definition
S	::=	$T\ m$([$T\,x$]*)	method signature
B	::=	$\{$[**var** [$T\,x$ [:= e]$^?$]*;]$^?$ [s;]$^?$ **put** $e\}$	method blocks
T	::=	I | Int | Bool | String | Void | Fut$\langle T\rangle$	types
v	::=	x | w	variables (local or field)
e	::=	**null** | this | v | cp | $f(\bar{e})$	pure expressions
s	::=	$v := e$ | fr := $v!m(\bar{e})$ | v := **get** e | v := **new** $C(\bar{e})$ | **skip** | **if** e **then** s [**else** s]$^?$ **fi** | s; s	statements

Fig. 1. Core language syntax, with C class name, cp formal class parameter, m method name, w fields, x method parameter or local variable, and where fr is a future variable. We let [ ]*, [ ]$^+$ and [ ]$^?$ denote repeated, repeated at least once and optional parts, respectively, and $\bar{e}$ is a (possibly empty) expression list. Expressions e and functions f are side-effect free.

once resolved. References to futures may be shared between objects, e.g., by passing them as parameters. After resolving a future reference, this means that third party objects may fetch the future value. Thus, the future value may be fetched several times, possibly by different objects. In this manner, shared futures provide an efficient way to distribute method call results to a number of objects.

For the purposes of this paper, we consider a core object-oriented language with futures, presented in Fig. 1. It includes basic statements for first-class futures, inspired by *ABS* [18]. Methods are organized in classes in a standard manner. A class C takes a list of formal parameters $\overline{cp}$, and defines fields $\overline{w}$, optional initialization statements $\bar{s}$ and methods $\overline{M}$. There is read-only access to class parameters $\overline{cp}$, method parameters $\bar{x}$, and implicit variables, such as this, referring to the current object, and the implicit method parameter future, referring to the future of the call. A method definition has the form $m(\bar{x})\{$**var** $\bar{y}$; s; **put** $e\}$, when ignoring type information, where $\bar{x}$ is the list of formal parameters, $\bar{y}$ is an optional list of *method-local variables*, s is a sequence of statements, and the value of e is put in the future of the call upon termination (i.e., "resolving the future").

A future variable fr is declared by Fut$\langle T\rangle fr$, indicating that fr may refer to futures which may contain values of type T. The call statement $fr := x!m(\bar{e})$ invokes the method m on object x with input values $\bar{e}$. The identity of the generated future is assigned to fr, and the calling process continues execution without waiting for fr to become resolved. The query statement $v :=$ **get** fr is used to fetch the value of a future. The statement blocks until fr is resolved, and then assigns the value contained in fr to v.

To avoid blocking, *ABS* provides statements for process control, including a statement **await** fr?, which releases the current process as long as fr is not yet resolved. This gives rise to more efficient programming with futures. In Section 8 we show how process release statements, including a releasing query statement, can be added as an extension of the present work. However, we first focus on a core language for futures, with a simple semantics, avoiding specialized features such as process control.

The core language contains additionally statements for assignment, **skip**, conditionals, and sequential composition. Object variables are typed by interfaces, and we assume that call and query statements are well-typed. If x refers to an object where m is defined with input types $\bar{S}$ and return type T, the following code is well-typed when $\bar{e}$ is of type $\bar{S}$: Fut$\langle T\rangle fr$; $T\ v$; $fr := x!m(\bar{e})$; $v :=$ **get** fr, which represents a traditional synchronous and blocking method call. This call is abbreviated by the notation

$$v := x.m(\bar{e})$$

Note that the call $v :=$ this$.m(\bar{e})$ will block, and thus a construct for local calls, say $v := m(\bar{e})$, would be useful. The core language ignores language features that are orthogonal to shared futures, such as inheritance. We refer to [19] for a treatment of this.

2.1. Publisher–subscriber example

To illustrate the language, and in particular the usage of shared futures, we consider an implementation of a version of the publisher–subscriber example, in which clients may subscribe to a service, while the service object is responsible for generating news and distributing each news update to the subscribing clients. To avoid bottlenecks when publishing events, the service delegates publishing to a chain of *proxy* objects, where each proxy object handles a bounded number of clients, as illustrated in Fig. 2. The implementation of the classes *Service* and *Proxy* can be found in Fig. 3. We use this implementation later in the paper to illustrate our reasoning techniques: We will define class invariants and illustrate the proof system by verification of these invariants.

The example takes advantage of the future concept by letting the service object delegate publishing of news updates to the proxies without waiting for the result of the news update. This is done by the sequence $fut := prod!detectNews();\ proxy!publish(fut)$. Thus the service object is not blocking by waiting for news updates. Furthermore, the calls on *add* are blocking; however, this is harmless since the implementation of *add* may not deadlock and terminates efficiently. The other calls in the example are not blocking nor involving shared futures.

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

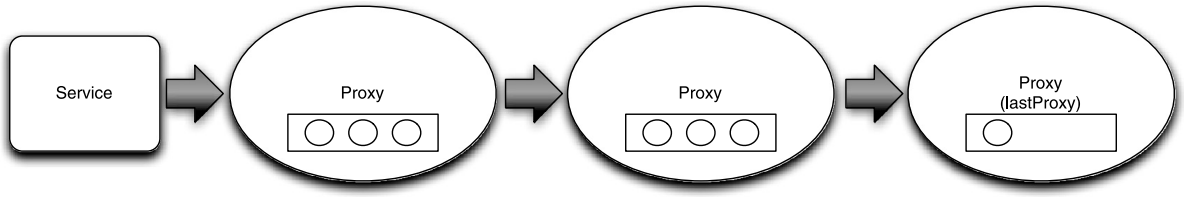


Fig. 2. A publisher–subscriber example with proxies handling at most three clients each.

For convenience, we introduce the following notation for uni- and multicast to be used in the example. A call $fr := x!m(\bar{e})$ is abbreviated to $x!m(\bar{e})$ when the future (and result) is not needed. For this kind of simple *message passing* we also allow x to be a collection of objects (a list in the example), with the effect that the invocation is sent to all objects in the collection, thereby allowing us to program *multi-casting* (without explicit futures). A multicast to an empty collection has no effect. For simplicity, we here omit the (redundant) return statement of Void methods.

3. Observable behavior

In this section we describe a communication model for concurrent objects communicating by means of asynchronous message passing and futures. The model is defined in terms of the *observable* communication between objects in the system. We consider how the execution of an object may be described by different *communication events* which reflect the observable interaction between the object and its environment. The observable behavior of a system is described by communication histories over observable events [10,11].

3.1. Communication events

Since message passing is asynchronous, we consider separate events for method invocation, reacting upon a method call, resolving a future, and for fetching the value of a future. Each event is observable to only one object, which is the one that *generates* the event. The events generated by a method call cycle is depicted in Fig. 4. The object o calls a method m on object o' with input values $\bar{e}$ and where u denotes the future identity. An invocation message is sent from o to o' when the method is invoked. This is reflected by the *invocation event* $\langle o \to o', u, m, \bar{e}\rangle$ generated by o. An *invocation reaction event*

```
interface ServiceI{
    Void subscribe(ClientI cl);
    Void produce()}
interface ProxyI{
    ProxyI add(ClientI cl);
    Void publish(Fut<News> fut)}
interface ClientI{
    Void signal(News ns)}
interface ProducerI{
    News detectNews()}

class Service(Int limit) implements ServiceI{
    ProducerI prod; ProxyI proxy; ProxyI lastProxy;
    {prod := new Producer(); proxy := new Proxy(limit,this); lastProxy := proxy; this!produce()}

    Void subscribe(ClientI cl){lastProxy := lastProxy.add(cl)}

    Void produce(){var Fut<News> fut; fut := prod!detectNews(); proxy!publish(fut)}}

class Proxy(Int limit, ServiceI s) implements ProxyI{
    List<ClientI> myClients := Nil; ProxyI nextProxy;

    ProxyI add(ClientI cl){
        var ProxyI lastProxy := this;
        if length(myClients) < limit then myClients := appendright(myClients, cl)
        else if nextProxy = null then nextProxy := new Proxy(limit,s) fi;
        lastProxy := nextProxy.add(cl) fi; put lastProxy}

    Void publish(Fut<News> fut){
        var News ns = None;
        ns := get fut; myClients!signal(ns);
        if nextProxy = null then s!produce() else nextProxy!publish(fut) fi}}
```

Fig. 3. Implementation of the publisher–subscriber example. Notice that *proxy!publish*(...) is a unicast and *myClients!signal*(...) is a multicast. We have used *ABS* syntax for lists. See Appendix A for a full implementation including data type definitions (including *News*) and implementation of the *Producer* and *Client* classes.

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

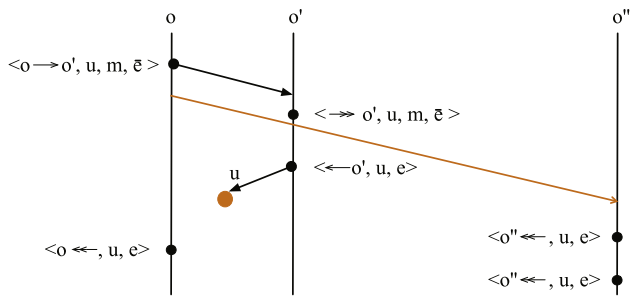


Fig. 4. A method call cycle: object o calls a method m on object o' with *future* u. The events on the left-hand side are visible to o, those in the middle are visible to o', and the ones on the right-hand side are visible to o''. There is an arbitrary delay between message receiving and reaction. The message sending from o to o'' represents that the *future* u is passed from o to o''.

$\langle \twoheadrightarrow o', u, m, \bar{e}\rangle$ is generated by o' once the method starts execution. When the method terminates, the object o' generates the *future event* $\langle \leftarrow o', u, e\rangle$. This event reflects that u is resolved with return value e. The *fetching event* $\langle o \twoheadleftarrow, u, e\rangle$ is generated by o when o fetches the value of the resolved future. Since future identities may be passed to other objects, e.g., o'', that object may also fetch the future value, reflected by the event $\langle o'' \twoheadleftarrow, u, e\rangle$, generated by o''. The *object creation event* $\langle o \uparrow o', C, \bar{e}\rangle$ represents object creation, and is generated by o when o creates a fresh object o'.

Definition 1 *(Events).* Let type Mid include all method names, and let Data be the supertype of all values occurring as actual parameters, including future identities Fid and object identities Oid. Let *caller*, *callee*, *receiver* : Oid, *future* : Fid, *method* : Mid, *args* : List[Data], and *result* : Data. Communication events Ev include:

- *Invocation events* $\langle caller \rightarrow callee, future, method, args\rangle$, generated by *caller*.
- *Invocation reaction events* $\langle \twoheadrightarrow callee, future, method, args\rangle$, generated by *callee*.
- *Future events* $\langle \leftarrow callee, future, result\rangle$, generated by *callee*.
- *Fetching events* $\langle receiver \twoheadleftarrow, future, result\rangle$, generated by *receiver*.
- *Object creation events* $\langle caller \uparrow callee, class, args\rangle$, generated by *caller*.

Events may be decomposed by functions. For instance, $\_.result : \mathsf{Ev} \rightarrow \mathsf{Data}$ is well-defined for future and fetching events, e.g., $\langle \leftarrow o', u, e\rangle.result = e$.

For a method invocation with future u, the ordering of events depicted in Fig. 4 is described by the following regular expression (using $\cdot$ for sequential composition of events)

$$\langle o \rightarrow o', u, m, \bar{e}\rangle \cdot \langle \twoheadrightarrow o', u, m, \bar{e}\rangle \cdot \langle \leftarrow o', u, e\rangle[\cdot\langle \_ \twoheadleftarrow, u, e\rangle]^*$$

for some fixed o, o', m, $\bar{e}$, e, and where _ denotes an arbitrary value. This implies that the result value may be read several times, each time with the same value, namely that given in the preceding future event.

3.2. Communication histories

The execution of a system up to present time may be described by its history of observable events, defined as a sequence. A sequence over some type T is constructed by the empty sequence ε and the right append function $\_\cdot\_ : \mathsf{Seq}[T] \times T \rightarrow \mathsf{Seq}[T]$ (where "_" indicates an argument position). The choice of constructors gives rise to generate inductive function definitions, in the style of [13]. Projection, $\_/\_ : \mathsf{Seq}[T] \times \mathsf{Set}[T] \rightarrow \mathsf{Seq}[T]$ is defined inductively by $\varepsilon/s \triangleq \varepsilon$ and $(a \cdot x)/s \triangleq$ **if** $x \in s$ **then** $(a/s) \cdot x$ **else** a/s **fi**, for $a : \mathsf{Seq}[T]$, $x : T$, and $s : \mathsf{Set}[T]$, restricting a to the elements in s. For sequences a and b, let a **ew** x denote that x is the last element of a, and $a \leqslant b$ denote that a is a prefix (initial sequence) of b, while $a \sqsubseteq b$ denotes that a is a subsequence of b (not necessarily a tight one). Let $[x_1, x_2, \ldots, x_i]$ denote the sequence of $x_1, x_2, \ldots, x_i$ for $i > 0$. For instance, $[x_1, x_2] \leqslant [x_1, x_2, x_3, x_4, x_5]$ and $[x_2, x_4] \sqsubseteq [x_1, x_2, x_3, x_4, x_5]$. Functions for event decomposition are lifted to sequences in the standard way, ignoring events for which the decomposition is not defined, e.g., $\_.result : \mathsf{Seq}[\mathsf{Ev}] \rightarrow \mathsf{Seq}[\mathsf{Data}]$. A *communication history* for a set S of objects is defined as a sequence of events generated by the objects in S. We say that a history is *global* if S includes all objects in the system.

Definition 2 *(Communication histories).* The communication history h of a system of objects S is a sequence of type Seq[Ev], such that each event in h is generated by an object in S.

We observe that the *local history* of a single object o is achieved by restricting S to the single object, i.e., the history contains only elements generated by o. For a history h, we let h/o abbreviate the projection of h to the events generated by o. Since each event is generated by only one object, it follows that the local histories of two different objects are disjoint.

Definition 3 *(Local histories).* For a global history h and an object o, the projection h/o is the local history of o.

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

4. Operational semantics

We define the operational semantics by a transition system, presented in the style of SOS. A system state g is here captured as a mapping. In our case a system state (configuration) consists of objects, messages, and futures, each identified by unique identifiers. A mapping is seen as a set of bindings from identifiers to units: the bindings $id \mapsto \mathbf{ob}(state, code)$, $id \mapsto \mathbf{msg}(callee, method, args)$, and $id \mapsto \mathbf{fut}(value)$, represent bindings of an object identifier to an object, a future identifier to an invocation message and a future identifier to a future containing a value, respectively. Here *code* denotes a sequence of statements followed by a **put** statement, or is *empty*. The special constant null may not bind to an object. A *substate* is given by the corresponding submapping. The local state of an object is given by two mappings, one for attributes a and one for local variables l, mapping variable names to values. We assume a given interpretation of data types and associated functions.

Notation. We use the following mapping notation. A mapping is seen as a set of bindings written $[z_i \mapsto value_i]$ for a set of disjoint identifiers z_i, the domain. Map look-up is written $M[z]$ where M is a mapping and z an identifier. The notation is lifted to expressions, letting $M[e]$ mean the expression e evaluated in the state given by M. Map composition is written $M + M'$ where bindings in M' override those in M for the same identifier. A map update, written $M[z \mapsto d]$, is the map M updated by binding z to the value d, i.e., $M[z \mapsto d]$ is the same as $M + [z \mapsto d]$. For an expression e, the notation $M[z := e]$ abbreviates $M[z \mapsto M[e]]$.

The state of an object is given by a twin mapping, written $(a|l)$, where a is the state of the field variables (including this, nextFut, nextObj, class parameters $\overline{cp}$, and the local history $\mathcal{H}$) and l is the state of the parameters and local variables (including the implicit parameter future). Look-up $(a|l)[z]$ is simply given by $(a + l)[z]$. The notation $(a|l)[v := e]$ abbreviates **if** v *in* l **then** $(a\,|\,l[v \mapsto (a|l)[e]])$ **else** $(a[v \mapsto (a|l)[e]]\,|\,l)$, where *in* is used for testing domain membership. We extract the local state and code of an object o from the global state g by $g[o].State$ and $g[o].Code$, respectively.

Definition 4 *(Configuration mappings).* A configuration of type Config is a mapping from object identities to objects of the form $\mathbf{ob}(\delta, \bar{s})$, from future identities to values of the form

$$\mathbf{msg}(o, m, \bar{d}) \text{ or } \mathbf{fut}(d)$$

and possibly from the global history identifier H to a sequence of events h. The state δ of an object has the form of a twin mapping $(a|l)$. Let OB be the type of object bindings $o \mapsto \mathbf{ob}((a|l), \bar{s})$ such that $o \in$ Oid, $o \neq null$, and this $\mapsto o \in a$, FUT the type of future bindings $u \mapsto \mathbf{fut}(d)$ for $u \in$ Fid, MSG the type of message bindings $u \mapsto \mathbf{msg}(o, m, \bar{d})$ for $u \in$ Fid, and HIST the type of history bindings $H \mapsto h$ for $h \in$ Seq[Ev].

Thus in a global state we talk about objects, messages, futures, and possibly a representation of the global history, each with unique identities. An object identity o is mapped to the corresponding object, given as a pair of the state δ and the code $\bar{s}$ to be executed, written $\mathbf{ob}(\delta, \bar{s})$. A future identity u is mapped to a message containing the callee, the name of the called method and the actual parameters, written $\mathbf{msg}(o, m, \bar{d})$, or to the contained value, written $\mathbf{fut}(d)$. A class C could be bound to the attributes of the class and the set of methods, say $[C \mapsto \mathbf{class}(att, ms)]$. The method definitions in a class is of the form, $(m, \bar{p}, l, \bar{s})$ where m is the method name, $\bar{p}$ is the list of formal parameters, l contains the local variables (including default values), and $\bar{s}$ is the code. However, as classes here represent static information they are ignored in the operational semantics. The code of an object is simply a list of statements to be executed.

Generation of fresh object and future identifiers is modeled by the initial algebra given by the constructors:

- *initO* taking an object identity (the generating object) and returning an (initial) object identity,
- *initF* taking an object identity (the generating object) and returning an (initial) future identity,
- $next_{Fid}$ taking a future identity (say the last generated one) and returning a future identity,
- $next_{Oid}$ taking an object identity (say the last generated one) and returning an object identity.

When no ambiguity arises, we omit the index on the *next* function. A generator term, say $next(next(next(initF(o))))$, represents a unique future identity. Equality over generator terms is given by the syntactic equality, thus local uniqueness implies global uniqueness since the generating object is encoded in the terms. The operational semantics uses an attribute nextFut, initialized to *initF*(this), such that a fresh future identity is generated by *next*(nextFut). Similarly, the attribute nextObj is initialized to *initO*(this), such that a fresh object identity is generated by *next*(nextObj).

4.1. Operational rules

For our purpose, a configuration is a set of units such as concurrent objects, messages, futures, and possibly a representation of the global history. The units are uniquely identified and the configuration is formalized as a mapping. We use blank-space as the configuration constructor, allowing associativity, commutativity, identity (ACI) pattern matching. We later extend system configurations with an explicit representation of the global history ($H \mapsto h$). The history is included to define the interleaving semantics upon which we derive our history-based reasoning formalism.

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

skip:
$$o \mapsto \mathbf{ob}(\delta, \texttt{skip}; \bar{s}) \xrightarrow{empty} o \mapsto \mathbf{ob}(\delta, \bar{s})$$

assign :
$$o \mapsto \mathbf{ob}(\delta, v := e; \bar{s}) \xrightarrow{empty} o \mapsto \mathbf{ob}(\delta[v := e], \bar{s})$$

call :
$$o \mapsto \mathbf{ob}(\delta, fr := v!m(\bar{e}); \bar{s}) \xrightarrow{\langle o \,\rightarrow\, \delta[v], \delta[\mathsf{nextFut}], m, \delta[\bar{e}]\rangle} \begin{array}{l} o \mapsto \mathbf{ob}(\delta[fr := \mathsf{nextFut}, \mathsf{nextFut} := next(\mathsf{nextFut})], \bar{s}) \\ \delta[\mathsf{nextFut}] \mapsto \mathbf{msg}(\delta[v], m, \delta[\bar{e}]) \end{array}$$

start :
$$\begin{array}{l} u \mapsto \mathbf{msg}(o, m, \bar{d}) \\ o \mapsto \mathbf{ob}((a|l'), empty) \end{array} \xrightarrow{\langle \twoheadrightarrow\, o, u, m, \bar{d}\rangle} \begin{array}{l} o \mapsto \mathbf{ob}((a|(l[\bar{p} \mapsto \bar{d}, \mathsf{future} \mapsto u])), \bar{s}) \\ \textbf{where } m \text{ is statically bound to } (m, \bar{p}, l, \bar{s}) \end{array}$$

return :
$$o \mapsto \mathbf{ob}(\delta, \texttt{put}\ e) \xrightarrow{\langle \leftarrow\, o, \delta[\mathsf{future}], \delta[e]\rangle} \begin{array}{l} o \mapsto \mathbf{ob}(\delta, empty) \\ \delta[\mathsf{future}] \mapsto \mathbf{fut}(\delta[e]) \end{array}$$

query :
$$\begin{array}{l} u \mapsto \mathbf{fut}(d) \\ o \mapsto \mathbf{ob}(\delta, v := \texttt{get}\ \ e; \bar{s}) \end{array} \xrightarrow{\langle o \,\twoheadleftarrow, u\ , d\rangle} \begin{array}{l} u \mapsto \mathbf{fut}(d) \\ o \mapsto \mathbf{ob}(\delta[v := d], \bar{s}) \end{array} \quad \texttt{if}\ \ \delta[e] = u$$

new :
$$o \mapsto \mathbf{ob}(\delta, v := \texttt{new}\ C(\bar{e}); \bar{s}) \xrightarrow{\langle o \,\uparrow\, \delta[\mathsf{nextObj}], C, \delta[\bar{e}]\rangle} \begin{array}{l} o \mapsto \mathbf{ob}(\delta[v := \mathsf{nextObj}, \mathsf{nextObj} := next(\mathsf{nextObj})], \bar{s}) \\ \delta[\mathsf{nextObj}] \mapsto \mathbf{ob}(\delta_{init}, init) \end{array}$$

Fig. 5. Operational rules reflecting small-step semantics. Variables are denoted by single characters (the uniform naming convention is left implicit). An object state δ has the form $(a|l)$. Method names are assumed to be unique for each class, and indexed with the class name during type analysis. Thus in the operational semantics we may assume that method names are unique in the system. Consequently, method binding can be done at static time. In Rule start, we assume that m is bound to a method with local state l (including default values), parameters $\bar{p}$, and code $\bar{s}$. Note that parameters and the implicit parameter future, which are read-only, are added to the local state in Rule start. In Rule new, δ_{init} denotes the initial state (including the binding this $\mapsto \delta[\mathsf{nextObj}]$, $\overline{cp} \mapsto \delta[\bar{e}]$), and default/initial values for the fields; and *init* denotes the initialization statements of class C.

For disjoint *substates* g_1 and g_2 (i.e. mappings with disjoint domains) we let $g_1||g_2$ denote the composition, and let $||$ be an ACI operator. Since we deal with distributed systems communicating asynchronously, g_1 will involve exactly one object, o, plus possibly messages and futures. The context rule

$$\frac{g_1 \xrightarrow{\alpha} g_1'}{g_1||g_2 \xrightarrow{\alpha} g_1'||g_2}$$

allows us to derive system transitions for composed systems, and the rule for sequential composition allows us to deal with sequences of statements inside each object.

The operational rules are summarized in Fig. 5. Objects are concurrent in the sense that their executions are interleaved, and in each object statements are executed sequentially. Method invocation is captured by the rule call. The generated future identity is locally unique, and also globally unique since the identity is given by a generator term embedding the parent object. The future identity generated by this rule is first bound to an invocation message, which is to be consumed by rule start. And a future unit is generated upon method completion reusing the same future identifier. When there is no active code in an object, denoted *empty*, a method call is selected for execution by rule start. The invocation message is removed from the configuration by this rule, and the future identity of the call is assigned to the implicit parameter future. Method execution is completed by rule return, and a future value is fetched by rule query. A future unit appears in the configuration when resolved by rule return, which means that a query statement blocks until the future is resolved. Remark that rule query does not remove the future unit from the configuration, which allows several objects to fetch the value of the same future. Object creation is captured by the rule new. The generated object identity is locally unique, and also globally unique since the identity is given by a generator term embedding the parent object. The object identity generated by this rule is then bound to the generated object. The given language fragment may be extended with constructs for inter object reentrance, process control and suspension, e.g., by using the *ABS* approach of [17].

4.2. Augmenting the operational semantics with a history

We have above formulated an operational semantics, representing interleaving semantics, by transitions of the form $g_1 \xrightarrow{\alpha} g_2$, which expresses a transition from the system (sub)state g_1 to the system (sub)state g_2 labeled by the event α (possibly empty). The set of sequences of events for all possible executions corresponds to the trace set.

The given operational semantics does not explicitly include a history. The history of a state is implicitly given as the sequence of events that has occurred in the execution leading to this state, initially being empty. We may include the history H explicitly by transforming each rule $g_1 \xrightarrow{\alpha} g_2$ to

$$
\begin{array}{ll}
\text{call}: & \text{new}: \\
\quad o \mapsto \mathbf{ob}(\delta, fr := v!m(\bar{e}); \bar{s}), & \quad o \mapsto \mathbf{ob}(\delta, v := \texttt{new}\ C(\bar{e}); \bar{s}) \\
\quad H \mapsto h & \quad H \mapsto h \\
\longrightarrow\ o \mapsto \mathbf{ob}(\delta[fr := u], \bar{s}) & \longrightarrow\ o \mapsto \mathbf{ob}(\delta[v := o'], \bar{s}) \\
\quad u \mapsto \mathbf{msg}(\delta[v], m, \delta[\bar{e}]) & \quad o' \mapsto \mathbf{ob}(\delta_{init}, init) \\
\quad H \mapsto h \cdot \langle o \to \delta[v], u, m, \delta[\bar{e}]\rangle & \quad H \mapsto h \cdot \langle o \uparrow o', C, \delta[\bar{e}]\rangle \\
\texttt{if}\ \ u \notin id(h) & \texttt{if}\ \ o' \notin id(h)
\end{array}
$$

Fig. 6. Abstract rule for call and object generation. As earlier δ_{init} and *init* denote the initial state (including the binding $\mathsf{this} \mapsto o'$ and binding of the actual class parameters $\delta[\bar{e}]$) and the initialization statements of class C, respectively.

$$g_1 + [H \mapsto h] \longrightarrow g_2 + [H \mapsto h \cdot \alpha]$$

where h is (the value of) the history of the prestate and $h \cdot \alpha$ the history of the poststate, letting $h \cdot \alpha$ denote h when α is empty. In this way the special identifier H maps to the current history in a global state. Note that the event α may be omitted from the transition symbol in this version of the semantics, since it is redundant. Since we deal with predicates referring to the history, we will below use this history-explicit semantics. Furthermore, we have implemented the history-explicit semantics in Maude, which then allows run-time testing of properties over histories.

Assignments to the history is not possible with the given operation rule for assignments. It would require a specialized assignment rule. However, a program may not update the history by explicit assignments, since the history is hidden from the programmer.

4.2.1. An abstract semantics

The operational semantics above is executable and in particular includes an algorithm for generation of future identities and object identities. Local uniqueness will here imply global uniqueness. This is due to the rules call and new. By redefining these two rules we may give a more abstract semantics, based on non-determinism in generation of fresh identities. The abstract semantics uses history information, and we here therefore consider rules with explicit representation of the history. The abstract semantics is given in Fig. 6, in which the function *id* is overloaded, i.e., either Seq[Ev] → Set[Id] or List[Data] → Set[Id], where Id is the supertype of Oid and Fid. Namely, *id* extracts all the object/future identities occurring in a history and in the expression list $\bar{e}$, as follows:

$$
\begin{array}{llll}
id(\varepsilon) & \triangleq \{\mathsf{null}\} & id(h \cdot \gamma) & \triangleq id(h) \cup id(\gamma) \\
id(\langle o \to o', u, m, \bar{e}\rangle) & \triangleq \{o, o', u\} \cup id(\bar{e}) & id(\langle \twoheadrightarrow o', u, m, \bar{e}\rangle) & \triangleq \{o', u\} \cup id(\bar{e}) \\
id(\langle \leftarrow\ o', u, e\rangle) & \triangleq \{o', u\} \cup id(e) & id(\langle o\ \twoheadleftarrow, u, e\rangle) & \triangleq \{o, u\} \cup id(e) \\
id(\langle o \uparrow o', C, \bar{e}\rangle) & \triangleq \{o, o'\} \cup id(\bar{e}) & &
\end{array}
$$

where $\gamma : \mathsf{Ev}$. Remark that null is always included in $id(h)$. For a global history h, the projection $id(h)/\mathsf{Fid}$ returns all future identities in h, and $id(h/o)/\mathsf{Fid}$ returns the futures generated by o or received as parameters.

Note that the special variables nextFut and nextObj are not needed, nor are the semantical functions initF/initO and next for generating identities. Global uniqueness of object and future identities is here an explicit condition. Clearly the old version of the call and new rules is a specialization of the abstract semantics. Thus a history obtainable with the executable semantics is also a possible history of the abstract semantics.

4.3. Semantic properties

We provide a notion of global and local wellformedness for histories corresponding to the abstract operational semantics, where the constructive approach to making fresh identities is abstracted away. (For soundness we could also use a version of wellformedness corresponding to the executable operational semantics.)

Definition 5 *(Globally wellformed histories).* Let $h : \mathsf{Seq[Ev]}$ be a non-empty history of a global object system S. The wellformedness predicate $wf : \mathsf{Seq[Ev]} \to \mathsf{Bool}$ is defined by:

$$
\begin{array}{ll}
wf(\varepsilon) & \triangleq true \\
wf(\langle o \uparrow o, C, \bar{e}\rangle) & \triangleq o \neq \mathsf{null}\ \wedge id(\bar{e}) = \emptyset \\
wf(h \cdot \langle o \to o', u, m, \bar{e}\rangle) & \triangleq wf(h) \wedge o \in oid(new_{ob}(h)) \wedge (\{o'\} \cup id(\bar{e})) \subseteq id(h/o) \wedge u\ \notin id(h) \\
wf(h \cdot \langle \twoheadrightarrow o, u, m, \bar{e}\rangle) & \triangleq wf(h) \wedge o \in oid(new_{ob}(h)) \wedge\ h/u\ \ \mathbf{ew}\ \langle \_ \to o, u, m, \bar{e}\rangle \\
wf(h \cdot \langle \leftarrow\ o, u, e\rangle) & \triangleq wf(h) \wedge o \in oid(new_{ob}(h)) \wedge id(e) \subseteq id(h/o) \wedge h/u\ \mathbf{ew}\ \langle \twoheadrightarrow o, u, \_, \_\rangle \\
wf(h \cdot \langle o\ \twoheadleftarrow, u, e\rangle) & \triangleq wf(h) \wedge o \in oid(new_{ob}(h)) \wedge u\ \in id(h/o) \wedge (h/u).result\ \mathbf{ew}\ e \\
wf(h \cdot \langle o \uparrow o', C, \bar{e}\rangle) & \triangleq wf(h) \wedge o \in oid(new_{ob}(h)) \wedge id(\bar{e}) \subseteq id(h/o) \wedge o' \notin id(h)
\end{array}
$$

where non-interesting arguments are identified by _ in projections, and h/u abbreviates the projection of the history h to the future u, i.e. the subsequence of events γ such that $\gamma.future = u$. Similarly, $h.result$ is the sequence of result values extracted

from the subsequence of h of events which has a result, i.e., future and fetching events. As defined below, $new_{ob}(h/o)$ extracts the objects *created by* o.

Wellformedness expresses that generated object and future identities are fresh ($\notin id(h)$ clauses), and otherwise no locally new identities are created (subset of $id(h/o)$ clauses); that active objects have been generated ($\in oid(new_{ob}(h)$ clauses); and that the ordering of method call cycles given in Fig. 4 is respected (**ew** clauses). Note that the first event in a global history must be an object generation event, representing an externally generated initial object. The initial object has no accessible creator, and we use the convention that it is created by itself. The function $new_{ob} : \mathsf{Seq[Ev]} \to \mathsf{Set[Obj} \times \mathsf{Cls} \times \mathsf{List[Data]]}$ returns the set of created objects (each given by its object identity, associated class and class parameters) in a history:

$$\begin{array}{lcl} new_{ob}(\varepsilon) & \triangleq & \emptyset \\ new_{ob}(h \cdot \langle o \uparrow o', C, \bar{e}\rangle) & \triangleq & new_{ob}(h) \cup \{o' : C(\bar{e})\} \\ new_{ob}(h \cdot \textbf{\textit{others}}) & \triangleq & new_{ob}(h) \end{array}$$

where ***others*** matches all other events. The function $oid : \mathsf{Set[Obj} \times \mathsf{Cls} \times \mathsf{List[Data]]} \to \mathsf{Set[Obj]}$ extracts object identities o from a set of elements of the form $\{o : C(\bar{e})\}$, like from the output of function new_{ob}.

For the core language considered, method bodies are executed sequentially, and it is possible to strengthen the notion of wellformedness to reflect this, i.e., ensuring

$$0 \leq \#(h/o/\{\twoheadrightarrow\}) - \#(h/o/\{\leftarrow\}) \leq 1$$

where # denotes sequence length. However, this would exclude addition of mechanisms for process release or non-blocking query of futures. Since we will consider an addition such statements in Section 8, and since this property can be expressed by a local invariant, we do not let our notion of wellformedness reflect sequentially ordered method executions.

It follows directly from Definition 5 that a wellformed global history is monotone in the sense

$$h \leq h' \Rightarrow wf(h') \Rightarrow wf(h)$$

and that it satisfies the communication order pictured in Fig. 4, i.e.,

$$\forall u\ .\exists o, o', m, \bar{e}, e\,.\quad h/u\ \leqslant [\langle o \to o', u, m, \bar{e}\rangle \cdot \langle \twoheadrightarrow o', u, m, \bar{e}\rangle \cdot \langle \leftarrow\ o', u, e\rangle[\cdot\langle\_\ \twoheadleftarrow, u, e\rangle]^*]$$

We can prove that the operational semantics guarantees wellformedness:

Lemma 1. *The global history h of a global object system S obtained by the given abstract operational semantics, is wellformed, $wf(h)$.*

This lemma can be proved by induction over the number of rule applications, proving the inductive property

$$\begin{array}{l} \quad wf(g[H]) \\ \wedge\ Futures(g[H]) = g/(\mathsf{MSG}\ \cup \mathsf{FUT}) \\ \wedge\ \forall o\,.\,id(g[o].State) \subseteq id(g[H]/o) \end{array}$$

for any reachable configuration g, where $g/(\mathsf{MSG} \cup \mathsf{FUT})$ is the submapping of g consisting of bindings to messages and futures, and where $Futures(h)$ is the sequence of bindings to messages and futures calculated from the history. The second conjunct is needed to prove the event ordering information (given by **ew** of the wellformedness predicate). The third conjunct says that future/object identities found in the state of a given object have been observed in the local history. This is needed to prove that future identities appearing as *future* in the *future/fetching events*, object identities appearing as *callee* in the *invocation events*, and future/object identities appearing as *args* or *result* in the *invocation/future/new events* have been observed in the local history, given that our underlying programming language does not offer any functions producing (new) future or object identities. Note that freshness of identities and non-nullness of the objects generating the events follow by the definition of configuration.

Definition 6 *(Locally wellformed histories).* Local wellformedness of a local history h for an object o, denoted $wf_o(h)$, is defined by

$$wf_o(h) \triangleq \exists h'\,.\,wf(h') \wedge h = h'/o$$

Here h ranges over local histories and h' over global histories.

It follows that global wellformedness implies local wellformedness of the local history of an object o, i.e.,

$$wf(h) \Rightarrow wf_o(h/o)$$

Thus local wellformedness may be assumed in a given class. Moreover, it follows that local wellformedness reduces to

$$
\begin{array}{lll}
wf_o(\varepsilon) & \triangleq & true \\
wf_o(h \cdot \langle o \rightarrow o', u, m, \bar{e}\rangle) & \triangleq & wf_o(h) \wedge o \neq \mathsf{null} \wedge (\{o'\} \cup id(\bar{e})) \subseteq id(h) \wedge u \notin id(h) \\
wf_o(h \cdot \langle \twoheadrightarrow o, u, m, \bar{e}\rangle) & \triangleq & wf_o(h) \wedge o \neq \mathsf{null} \wedge h/u \leq \langle o \rightarrow o, u, m, \bar{e}\rangle \\
wf_o(h \cdot \langle \leftarrow o, u, e\rangle) & \triangleq & wf_o(h) \wedge o \neq \mathsf{null} \wedge id(e) \subseteq id(h) \wedge h/u \ \mathbf{ew} \ \langle \twoheadrightarrow o, u, \_, \_\rangle \\
wf_o(h \cdot \langle o \twoheadleftarrow, u, e\rangle) & \triangleq & wf_o(h) \wedge o \neq \mathsf{null} \wedge u \in id(h) \wedge (h/u \leq [\langle o \rightarrow \_, u, \_, \_\rangle] \vee \\
 & & (h/u).result \ \mathbf{ew} \ e) \wedge h/u/\{\rightarrow o, \twoheadrightarrow, \leftarrow\} \neq \varepsilon \Rightarrow \langle \leftarrow o, u, e\rangle \in h \\
wf_o(h \cdot \langle o \uparrow o', C, \bar{e}\rangle) & \triangleq & wf_o(h) \wedge o \neq \mathsf{null} \wedge id(\bar{e}) \subseteq id(h) \wedge o' \notin id(h)
\end{array}
$$

where h ranges over histories local to o. The clause $h/u/\{\rightarrow o, \twoheadrightarrow, \leftarrow\} \neq \varepsilon$ expresses that the object o is the callee of the call with future u. Note that the first event may or may not be a creation event, possibly an external creation represented by a self creation.

5. Program verification

5.1. Local reasoning

Local assertions express conditions on a state of a given object and the local history $\mathcal{H}$. Thus in a class C assertions may refer to the fields of C, as well as the class parameters cp, and this, which are constant. Inside a method the assertions may refer to the formal parameters (including future) and local variables of that method. Assertions may not refer to the run-time variables nextFut and nextObj used in the executable operational rules, nor the corresponding semantical functions for generating fresh names (*initF*, *initO* and *next*). For convenience, we let WF abbreviate $wf_{this}(\mathcal{H})$ since this is the only wellformedness predicate one may talk about inside a given class.

The reasoning rules for the core language are defined in Fig. 7. The triple $\{P\}\,\bar{s}\,\{Q\}$ expresses that if P holds prior to execution of the statement list $\bar{s}$ then Q holds after execution, provided it terminates. We write $\vdash \{P\}\,\bar{s}\,\{Q\}$ if $\{P\}\,\bar{s}\,\{Q\}$ is derivable by the rules. If the statement list $\bar{s}$ is empty we write $\vdash \{P\}\ \{Q\}$. Notice that the Rule imp together with sequential composition comp allows us to conclude $\{P\}\,\bar{s}\,\{Q\}$ from $\{P'\}\,\bar{s}\,\{Q'\}$ when $P \wedge WF \Rightarrow P'$ and $Q' \wedge WF \Rightarrow Q$. Standard rules for if-statements and adaptation are not included here. An adaptation rule would allow reuse of verification of triples, for instance when strengthening an invariant.

The rules for comp, skip, and assign, are standard. The imp rule is specialized to empty statement expressing that executions give wellformed states. The other rules deal with futures or object generation, and involve effects on the local history. The effects of a method call cycle is reflected by the rules call, method, return, and query, each introducing a call cycle event, an *invocation*, *invocation reaction*, *future*, and *fetching* event, respectively. Rule new extends the history of the creating object with an *object creation* event. The universal quantifiers in the precondition of call, query, and new reflect non-determinism of the introduced logical variables in the local reasoning. The use of $\bar{y}'$ in rule body reflects that $\bar{y}$ in P and Q does not refer to the local variables, $\bar{y}$.

As an example we consider reasoning about a blocking self call, i.e., $v := \mathsf{this}.m(\bar{e})$, which is an abbreviation of $\mathsf{Fut}\langle T\rangle fr;\ fr := \mathsf{this}!m(\bar{e});\ v := \mathbf{get}\ fr$. We should be able to verify

$$\{true\}\ v := \mathsf{this}.m(\bar{e})\ \{false\} \tag{1}$$

since this call blocks and will never terminate, as there is no reentrance in the core language. By the imp rule it suffices to prove

$$\{true\}\ fr := \mathsf{this}\ !m(\bar{e});\ v := \mathbf{get}\ fr\ \{WF \Rightarrow false\}$$

By the rule for query and call, this reduces to proving the precondition

$$\forall fr, v\,.\, WF^{\mathcal{H}}_{\mathcal{H}\cdot\langle \mathsf{this} \rightarrow \mathsf{this}, fr, m, \bar{e}\rangle \cdot \langle \mathsf{this} \twoheadleftarrow, fr, v\rangle} \Rightarrow false$$

By definition of local wellformedness this can be reduced to $false \Rightarrow false$, which is $true$, since $\mathcal{H}/fr/\{\rightarrow \mathsf{this}, \twoheadrightarrow, \leftarrow\} \neq \varepsilon$ but $\langle \leftarrow \mathsf{this}, fr, e\rangle \in \mathcal{H}$ does not hold. Thus we have verified the triple (1). This example demonstrates the strength of the reasoning system, but it also indicates a weakness of the core language. In order to better support self calls, one could consider reentrance or release mechanisms. In Section 8 we consider the latter and extend the language with constructs for non-blocking and terminating self calls.

Note that one might split the method rule into two rules, block and method', as follows:

$$\text{block}\quad \frac{\{P^{\bar{y}}_{\bar{y}'} \wedge \bar{y} = default\}\ \bar{s}\ \{Q^{\bar{y}}_{\bar{y}'}\}}{\{P\}\ \{\mathbf{var}\ \bar{y};\ \bar{s}\}\ \{Q\}}$$

$$\text{method'}\quad \frac{\{P\}\ body\ \{Q\}}{\{P^{\mathcal{H}}_{\mathcal{H}\cdot\langle \twoheadrightarrow \mathsf{this}, \mathsf{future}, m, \bar{x}\rangle}\}\ (m(\bar{x})\ body)\ \{Q\}}$$

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

imp $$\frac{P \wedge WF \Rightarrow Q}{\{P\}\ \{Q\}}$$

comp $$\frac{\{P\}\,\overline{s_1}\,\{R\} \quad \{R\}\,\overline{s_2}\,\{Q\}}{\{P\}\,\overline{s_1};\overline{s_2}\,\{Q\}}$$

skip $$\{Q\}\ \mathbf{skip}\ \{Q\}$$

assign $$\{Q^{v}_{e}\}\ v := e\ \{Q\}$$

call $$\{\forall fr'\,.\,Q^{fr,\mathcal{H}}_{fr',\mathcal{H}\ \cdot\langle \mathsf{this}\ \rightarrow o, fr', m, \bar{e}\rangle}\}\ fr := o!m(\bar{e})\ \{Q\}$$

method $$\frac{\{P^{\bar{y}}_{\bar{y}'} \wedge \bar{y} = default\}\ \bar{s}\ \{Q^{\bar{y}}_{\bar{y}'}\}}{\{P^{\mathcal{H}}_{\mathcal{H}\ \cdot\langle \twoheadrightarrow \mathsf{this,future}, m, \bar{x}\rangle}\}\ (m(\bar{x})\ \{\mathbf{var}\ \bar{y};\bar{s}\})\ \{Q\}}$$

return $$\{Q^{\mathcal{H}}_{\mathcal{H}\ \cdot\langle \leftarrow\ \mathsf{this,future}, e\rangle}\}\ \mathbf{put}\ e\ \{Q\}$$

query $$\{\forall v'\,.\,Q^{v,\mathcal{H}}_{v',\mathcal{H}\ \cdot\langle \mathsf{this} \twoheadleftarrow, e, v'\rangle}\}\ v := \mathbf{get}\ e\ \{Q\}$$

new $$\{\forall v'\,.\,Q^{v,\mathcal{H}}_{v',\mathcal{H}\ \cdot\langle \mathsf{this}\uparrow v', C, \bar{e}\rangle}\}\ v := \mathbf{new}\ C(\bar{e})\ \{Q\}$$

Fig. 7. Hoare style rules for the core language. Primed variables represent fresh logical variables, and WF is an abbreviation for $wf_{this}(\mathcal{H})$.

Here block represents classical reasoning about blocks and method' represents the simple extension from blocks to annotated method declarations.

By applying the rule imp and sequential composition comp we may also derive $\{P\}\,\bar{s}\,\{Q\}$ from $\{P\}\,\bar{s}\,\{WF \Rightarrow Q\}$. Implication the other direction is trivial, since Q is stronger than $WF \Rightarrow Q$: We may derive $\{P\}\,\bar{s}\,\{WF \Rightarrow Q\}$ from $\{P\}\,\bar{s}\,\{Q\}$ by the imp rule. Accordingly, we have the following result:

Lemma 2 *(Equivalence of Hoare triples).*

$$\vdash \{P\}\,\bar{s}\,\{WF \Rightarrow Q\} \Leftrightarrow \vdash \{P\}\,\bar{s}\,\{Q\}$$

5.2. Invariant reasoning

In interactive and non-terminating systems, it is difficult to specify and reason compositionally about object behavior in terms of pre- and post-conditions of the defined methods. Instead, pre- and post-conditions to method definitions are in our setting used to establish a so-called *class invariant*, i.e., a local assertion that holds after initialization of the class, and is maintained by all methods. The class invariant serves as a *contract* for the class: A method implements its part of the contract by ensuring that the invariant holds upon termination, assuming that the invariant holds initially. To facilitate compositional and component-based reasoning about programs, the class invariant is used to establish a *relationship between the internal state and the observable behavior of the class instance*. The internal state is given by the values of the class fields, whereas the observable behavior is expressed as a set of potential communication histories. By hiding the internal state, class invariants form a suitable basis for compositional reasoning about object systems. Assumptions to the environment may be reflected by invariants on the form of implications.

A *user-provided invariant* $I_C(\overline{w}, \mathcal{H})$ for a class C is a predicate over the fields $\overline{w}$ and the local history $\mathcal{H}$, as well as the formal class parameters $\overline{cp}$ and this, which are constant (read-only) variables.

5.3. Compositional reasoning

A history invariant for instances of C is a predicate that only talks about the local history of that object and is satisfied at all times (in contrast to class invariants that may be temporarily violated inside a method). A history invariant can usually be derived from the class invariant (when prefix-closed). For an instance o of C with actual class parameter values $\bar{e}$, the history invariant $I_{o:C(\bar{e})}(h)$ is defined by hiding the internal state $\overline{w}$ and instantiating this and the class parameters $\overline{cp}$:

$$I_{o:C(\bar{e})}(h) \triangleq \exists \overline{w}\ .\ I_C(\overline{w}, h)^{\mathsf{this},\overline{cp}}_{o,\bar{e}}$$

but in addition it must be proved that $I_{o:C(\bar{e})}(\mathcal{H})$ holds at all times, i.e., is maintained by each statement in the class C, possibly weakening the class invariant if needed. In practice this is trivial, when the history invariant is prefix closed (with respect to the history) [20].

We next consider systems with several objects and with an externally created initial object. The initial object may create some objects which again may create other objects and so on. We say that the system is *generated by* the externally created

object. (We might generalize to several externally generated objects, using a reserved name to represent the environment of the program.)

The history invariant $I_S(h)$ for a system S given by an initial object, say $c : C(\bar{e})$, is then given by the conjunction of the history invariants of the initial and generated objects on their respective local histories:

$$I_S(h) \triangleq \langle c \uparrow c, C, \bar{e}\rangle \leq h \wedge wf(h) \bigwedge_{(o:C(\bar{e}))\in new_{ob}(h)} I_{o:C(\bar{e})}(h/o)$$

The externally created object will appear as an initial creation event in the global history, and thus be part of $new_{ob}(h)$. The wellformedness property serves as a connection between the local histories, relating events with the same future to each other. Note that the system invariant is obtained directly from the history invariants of the dynamically composed objects, without any restrictions on the local reasoning, since the local histories are disjoint. This ensures compositional reasoning. The composition rule is similar to [17]. Soundness of the composition rule is considered in [20].

6. Specification and verification of the publisher–subscriber example

In this example we consider object systems based on the classes found in Fig. 3. Different executions may lead to different global histories for this system, depending on the interleaving of the different object activities. However, we may state some general properties, like the following one: For every *signal* invocation from a proxy py to a client c with news ns, the client must have *subscribed* to a service v, which must have issued a *publish* invocation with a future u generated by a *detectNews* invocation, and then the proxy py must have received news ns from that future. This expresses that when clients get news it is only from services they have subscribed to, and the news is resulting from actions of the server. This global property can be formalized as follows:

$$\begin{aligned} H \ \mathbf{ew}\ \langle py \rightarrow c, u_0, signal, ns\rangle \Longrightarrow \exists v, u\,.\,(&\langle \twoheadrightarrow v, \_, subscribe, c\rangle, \langle py \ \twoheadleftarrow, u, ns\rangle) \sqsubseteq H \wedge \\ &(\langle v \rightarrow \_, u, detectNews, \varepsilon\rangle, \langle v \rightarrow \_, \_, publish, u\rangle) \sqsubseteq H \end{aligned}$$

where non-interesting arguments are identified by _ rather than existentially quantified variables, for better readability.

We may derive this property within the proof system using the following class invariants, which focus on the order of the local events (while ignoring fields):

$$\begin{aligned} I_{Service(limit)}(\mathcal{H}) \ \triangleq\ & (\exists c\,.\,\langle \mathsf{this} \rightarrow \_, \_, add, c\rangle \sqsubseteq \mathcal{H} \\ & \quad \Longrightarrow (\langle \twoheadrightarrow \mathsf{this}, \_, subscribe, c\rangle, \langle \mathsf{this} \rightarrow \_, \_, add, c\rangle) \sqsubseteq \mathcal{H}) \wedge \\ & (\exists u\,.\,\langle \mathsf{this} \rightarrow \_, \_, publish, u\rangle \sqsubseteq \mathcal{H} \\ & \quad \Longrightarrow (\langle \mathsf{this} \rightarrow \_, u, detectNews, \varepsilon\rangle, \langle \mathsf{this} \rightarrow \_, \_, publish, u\rangle) \sqsubseteq \mathcal{H}) \end{aligned}$$

$$\begin{aligned} I_{Proxy(limit,s)}(\mathcal{H}) \ \triangleq\ & (\exists u\,.\,\langle \mathsf{this} \rightarrow \_, \_, publish, u\rangle \sqsubseteq \mathcal{H} \\ & \quad \Longrightarrow (\langle \twoheadrightarrow \mathsf{this}, \_, publish, u\rangle, \langle \mathsf{this} \rightarrow \_, \_, publish, u\rangle) \sqsubseteq \mathcal{H}) \wedge \\ & (\exists c, ns\,.\,\langle \mathsf{this} \rightarrow c, \_, signal, ns\rangle \sqsubseteq \mathcal{H} \\ & \quad \Longrightarrow \exists u\,.\,\langle \twoheadrightarrow \mathsf{this}, \_, add, c\rangle, \langle \twoheadrightarrow \mathsf{this}, \_, publish, u\rangle, \langle \mathsf{this} \ \twoheadleftarrow, u, ns\rangle, \\ & \qquad \langle \mathsf{this} \rightarrow c, \_, signal, ns\rangle \sqsubseteq \mathcal{H}) \end{aligned}$$

These invariants are straightforwardly verified in the above proof system. As explained, the corresponding invariants for the object instances $s : Service(limit)$ and $p : Proxy(limit, s)$ are obtained by substituting actual values for this and class parameters:

$$\begin{aligned} I_{s:Service(limit)}(\mathcal{H}) \ \triangleq\ & (\exists c\,.\,\langle s \rightarrow \_, \_, add, c\rangle \sqsubseteq \mathcal{H} \\ & \quad \Longrightarrow (\langle \twoheadrightarrow s, \_, subscribe, c\rangle, \langle s \rightarrow \_, \_, add, c\rangle) \sqsubseteq \mathcal{H}) \wedge \\ & (\exists u\,.\,\langle s \rightarrow \_, \_, publish, u\rangle \sqsubseteq \mathcal{H} \\ & \quad \Longrightarrow (\langle s \rightarrow \_, u, detectNews, \varepsilon\rangle, \langle s \rightarrow \_, \_, publish, u\rangle) \sqsubseteq \mathcal{H}) \end{aligned}$$

$$\begin{aligned} I_{p:Proxy(limit,s)}(\mathcal{H}) \ \triangleq\ & (\exists u\,.\,\langle p \rightarrow \_, \_, publish, u\rangle \sqsubseteq \mathcal{H} \\ & \quad \Longrightarrow (\langle \twoheadrightarrow p, \_, publish, u\rangle, \langle p \rightarrow \_, \_, publish, u\rangle) \sqsubseteq \mathcal{H}) \wedge \\ & (\exists c, ns\,.\,\langle p \rightarrow c, \_, signal, ns\rangle \sqsubseteq \mathcal{H} \\ & \quad \Longrightarrow \exists u\,.\,\langle \twoheadrightarrow p, \_, add, c\rangle, \langle \twoheadrightarrow p, \_, publish, u\rangle, \langle p \ \twoheadleftarrow, u, ns\rangle, \\ & \qquad \langle p \rightarrow c, \_, signal, ns\rangle \sqsubseteq \mathcal{H}) \end{aligned}$$

The global invariant of a system S with one server object $s : Service(limit)$ and some clients, created by an initial object, say $c : C(\bar{e})$, is then

$$\begin{aligned} I_S(h) \ \triangleq\ & \langle c \uparrow c, C, \bar{e}\rangle \leq h \wedge wf(h) \wedge I_{s:Service(limit)}(h/s) \\ & \bigwedge_{(p:Proxy(limit,s))\in new_{ob}(h)} I_{p:Proxy(limit,s)}(h/p) \end{aligned}$$

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

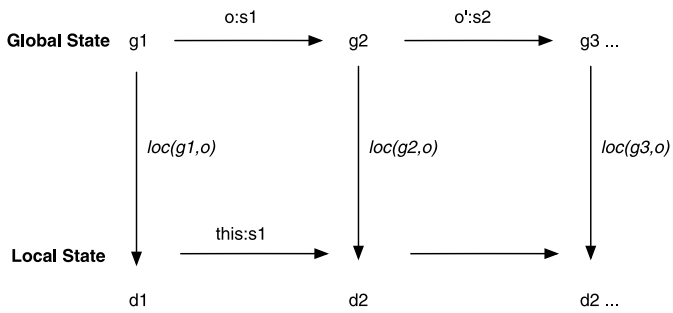


Fig. 8. Projection from the global states to the local states. Statement s_2 in object o' follows statement s_1 in object o. Local state d_2 is maintained by the transition $g_2 \xrightarrow{o':s_2} g_3$.

where wellformedness allows us to relate the different object histories. Note that invariants of other objects are ignored as we have not given invariants for other classes (and are by default *true*). From this global invariant we may inductively derive the system property defined above, by means of induction with respect to the length of the history, similarly to the buffer example in [17]. By strengthening the class invariant of *Proxy*, we can also prove other properties such as:

For each proxy, if *nextProxy* is null, the number of contained clients is less or equal to *limit*, otherwise equal to *limit*.

7. Soundness and completeness

We say that a reasoning system is sound if any provable property is valid, i.e.,

$$\vdash \{P\}\,\bar{s}\,\{Q\} \Rightarrow \models \{P\}\,\bar{s}\,\{Q\}$$

To prove that a reasoning system is sound, we need to show that all axioms of the system are valid and that all inference rules are sound, in the sense that they preserve validity. Validity of Hoare triples, denoted $\models \{P\}\,\bar{s}\,\{Q\}$, is defined by means of the operational semantics. We base the semantics on the operational semantics augmented with histories, as given by unlabeled transitions of the form $g_1 \rightarrow g_2$. Note that each rule is local to one object, and we write $g_1 \xrightarrow{o} g_2$ to indicate that the execution step is made by object o. When exactly one statement s is executed by o and we wish to highlight this statement, we write $g_1 \xrightarrow{o:s} g_2$:

Definition 7 *(Explicit execution step).*

$$g_1 \xrightarrow{o} g_2 \triangleq g_1 \rightarrow g_2 \wedge \forall o' \,.\, (o \neq o') \Leftrightarrow (g_1[o'] = g_2[o'])$$

expresses a transition from system (sub)state g_1 to system (sub)state g_2 due to an execution step made by object o (while all other objects are unchanged).

$$g_1 \xrightarrow{o:s} g_2 \triangleq g_1 \xrightarrow{o} g_2 \wedge g_1[o].Code = s;\, g_2[o].Code$$

expresses a transition from the system (sub)state g_1 to the system (sub)state g_2 due to an execution step made by object o through execution of statement s.

The latter relation is lifted to sequences of statements $\bar{s}$, executed by the same object o, by

$$g_1 \xrightarrow{o:\,s;\bar{s}} g_2 \triangleq \exists g' \,.\, g_1 \xrightarrow{o:s} g' \wedge g' \xrightarrow{o:\bar{s}} g_2$$

letting $g_1 \xrightarrow{o:\varepsilon} g_2 \triangleq g_1 = g_2$.

In Section 5 the axioms and inference rules are defined locally for each statement. Therefore, to express the soundness of our reasoning system, a notion of local state transitions are needed. Accordingly, we develop a proof structure to extract local state transition from the rewriting rules as shown in Fig. 8.

We consider pre- and post-conditions over local states and the local history. Such an assertion can be evaluated in a state defining values for attributes (of the appropriate class), parameters and local variables (of the method) and the local history. We let $P \xrightarrow[o]{o':s} Q$ express that if the condition P holds for object o before execution of s by object o', then Q holds for o after the execution. For convenience, we add a similar notation without s. This is defined as follows:

Definition 8 *(Validity of pre/post-conditions over execution steps).*

$$P \xrightarrow[o]{o'} Q \triangleq \forall g, g', \bar{z} \,.\, wf(g[H]) \wedge wf(g'[H]) \wedge g \xrightarrow{o'} g' \wedge loc(g, o)[P] \Rightarrow loc(g', o)[Q]$$

$$P \xrightarrow[o]{o':\bar{s}} Q \triangleq \forall g, g', \bar{z} \,.\, wf(g[H]) \wedge wf(g'[H\ ]) \wedge g \xrightarrow{o':\bar{s}} g' \wedge loc(g, o)[P] \Rightarrow loc(g', o)[Q]$$

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

where $\bar{z}$ is the list of auxiliary variables in P and/or Q, not bound by g nor g'. Here $loc(g, o)$ denotes the local state of object o, as derived from the global state g. The function $loc : \mathsf{Config} \times \mathsf{Oid} \to \mathsf{State}$ is defined by

$$loc(g, o) \triangleq g[o].State + \left[\mathcal{H} \mapsto g[H]/o\right]$$

where the resulting $\mathcal{H}$ ranges over local histories (i.e., in the alphabet of o), and where this is bound to o in g as explained earlier. Thus the extraction is made by taking the state of object o and adding the history localized to o. Then $loc(g, o)[x]$ is the value bound to variable x in the local state $loc(g, o)$. We also use the notation $loc(g, o)[e]$ to evaluate expressions e in the local state of o; and in particular $loc(g, o)[P]$ is the truth-value of the condition P in the local state of o, made by replacing the free variables in P by the corresponding values given by the local state and the given history.

Due to the local understanding of histories in local assertions and disjointness of alphabets for different objects, we have the following non-interference result:

Lemma 3 *(Non-interference).*

$$o \neq o' \Rightarrow P \xrightarrow[o]{o'} P$$

expressing that an assertion on object o is not affected by execution of other objects (o'). This means that local reasoning can be done locally.

The proof is straight forward by considering the rules of the operational semantics: Each rule involves exactly one object, and no object o' may change the state of object o (assuming $o \neq o'$). Object o' may change the global history but not the projection on o since each object has disjoint alphabets.

The above non-interference result allows the following local understanding of validity of Hoare triples for statement lists:

Definition 9 *(Validity of Hoare triples for statement lists).*

$$\models \{P\}\, \bar{s}\, \{Q\} \triangleq \forall o \,.\, P \xrightarrow[o]{o:\bar{s}} Q$$

Here o is the executing object and the object on which P and Q are interpreted. Thus a Hoare triple is valid if for any object o executing $\bar{s}$, the postcondition holds in the poststate, provided the precondition holds in the prestate, assuming wellformedness. As seen this is not affected by the environment of o.

Due to Definition 9 which incorporates global wellformedness, and the entailment of local wellformedness from global wellformedness, we have the following result:

Lemma 4 *(Equivalence of validity).*

$$\models \{P\}\, \bar{s}\, \{WF \Rightarrow Q\} \Leftrightarrow \models \{P\}\, \bar{s}\, \{Q\}$$

A Hoare axiom is said to be sound if it is valid. A proof rule is sound if validity of the premises imply validity of the conclusion.

Note that global validity, say of a global invariant I over H, could be defined directly on global states (with explicit H), i.e., the meaning of $\{I(H)\}\, s\, \{I(H)\}$ is that $I(g[H]) \Rightarrow I(g'[H])$ for all transitions $g \xrightarrow{o:s} g'$.

Theorem 1 *(Soundness).*

$$\vdash \{P\}\, \bar{s}\, \{Q\} \Rightarrow \models \{P\}\, \bar{s}\, \{Q\}$$

Theorem 2 *(Relative completeness).*

$$\models \{P\}\, \bar{s}\, \{Q\} \Rightarrow \vdash \{P\}\, \bar{s}\, \{Q\}$$

assuming completeness of the underlying logic for verification conditions.

The proofs for soundness and completeness are considered below. The proof is done by identifying weakest liberal preconditions for each construct, according to the given semantics, and prove that these are derivable in the system. In fact for each statement the given proof rules express the weakest liberal preconditions (modulo wellformedness). Thus the system is complete in the sense of Cook since the proof of completeness satisfies Cook's condition on expressiveness [21].

We first state a lemma useful for reducing the problem at hand. Due to Lemma 2 and Lemma 4, we derive the following result:

Lemma 5 *(Proof alternatives). For proving* $\vdash \{P\}\,\bar{s}\,\{Q\} \Leftrightarrow \models \{P\}\,\bar{s}\,\{Q\}$*, it suffices to prove* $\vdash \{P\}\,\bar{s}\,\{WF \Rightarrow Q\} \Leftrightarrow \models \{P\}\,\bar{s}\,\{WF \Rightarrow Q\}$.

In other words, to prove that the reasoning system for $\{P\}\,\bar{s}\,\{Q\}$ is sound and complete, it suffices to prove that the reasoning system is sound and complete for postconditions of the form $WF \Rightarrow Q$.

7.1. Proof of soundness and completeness

The proof of soundness is by induction on the proof structure. It suffices to prove that each axiom is valid and that each inference rule is sound. Below we consider validity of each axiom and soundness of the imp rule and method rule.

The proof of completeness is by induction on the number of applications of operational rules. We show below the base case corresponding to executing one basic statement. Each base case will involve at least one application of an operational rule. In the inductive step we may assume $\models \{P\}\,\bar{s}\,\{Q\}$ implies $\vdash \{P\}\,\bar{s}\,\{Q\}$ for n execution steps and it is sufficient to prove $\models \{P\}\,\bar{s}; s\,\{Q\}$ implies $\vdash \{P\}\,\bar{s}; s\,\{Q\}$ for any basic statement s.

We first show soundness and completeness for empty statement lists. For each basic statement s we will show that validity of $\{P\}\,s\,\{Q\}$ can be reduced to an implication $P \Rightarrow Q'$ and that $\{Q'\}\,s\,\{Q\}$ is a Hoare axiom, i.e., Q' is the weakest possible precondition according to validity. Thus soundness follows, and completeness also follows when Q' is the precondition of the axiom, since $\vdash \{P\}\,s\,\{Q\}$ can be obtained by the imp rule, using $P \Rightarrow Q'$. We show below for each basic statement s that the axiom for s expresses exactly the weakest possible precondition according to validity. Thus by the imp rule, completeness follows for s as above, proving $\vdash \{P\}\{Q'\}\,s\,\{Q\}$.

For sequential composition we notice first that by Lemma 5 and the imp rule, one may keep assertions on the form $WF \Rightarrow P$. And for these kinds of assertions the validity of sequential composition reduces to the standard one. Thus this case is not so interesting. Standard rules for if-statements and loops are not considered here. At the end we consider soundness and completeness for reasoning about annotated method declarations.

The implication rule. For the implication rule we observe that by definition $\models \{P\}\ \{Q\}$ reduces to

$$\forall o, g, \bar{z}\,.\, wf(g[H]) \wedge loc(g, o)[P] \Rightarrow loc(g, o)[Q]$$

which by definition of local wellformedness reduces to $WF \wedge P \Rightarrow Q$ (for all free variables involved), which is exactly the condition of the implication rule. Thus the given Hoare rule is sound.

For completeness of reasoning about empty sequence lists, we have that $\models \{P\}\ \{Q\}$ implies $\vdash \{P\}\ \{Q\}$ since $\models \{P\}\ \{Q\}$ gives $WF \wedge P \Rightarrow Q$ which again gives $\vdash \{P\}\ \{Q\}$, and since we assume completeness of the underlying logic.

7.1.1. Basic statements

Before we look at the different basic statements, we first notice that given a semantic rule of form $o \mapsto \mathbf{ob}(\delta, s; \bar{s}) \xrightarrow{\alpha} o \mapsto \mathbf{ob}(\delta', \bar{s})$, we may reduce $\models \{P\}\,s\,\{Q\}$ to

$$wf(h) \wedge wf(h \cdot \alpha) \wedge \delta[P^{\mathcal{H}}_{h/o}] \Rightarrow \delta'[Q^{\mathcal{H}}_{h/o\cdot\alpha}]$$

(for all o, h, and other free variables) which by prefix closure of wf is

$$wf(h \cdot \alpha) \wedge \delta[P^{\mathcal{H}}_{h/o}] \Rightarrow \delta'[Q^{\mathcal{H}}_{h/o\cdot\alpha}]$$

which by definition of local wellformedness is

$$wf_o(\mathcal{H} \cdot \alpha) \wedge \delta[P] \Rightarrow \delta'[Q^{\mathcal{H}}_{\mathcal{H}\cdot\alpha}]$$

for all local histories $\mathcal{H}$ and other free variables, since h/o spans all possible local histories, i.e., for all local histories h we have that $h/o = h$, and using the definition of local wellformedness. This can be reformulated as

$$\delta[P] \Rightarrow (wf_o(\mathcal{H}\ \cdot \alpha) \Rightarrow \delta'[Q^{\mathcal{H}}_{\mathcal{H}\cdot\alpha}])$$

which is the same as

$$\delta[P] \Rightarrow \delta'[WF \Rightarrow Q]^{\mathcal{H}}_{\mathcal{H}\cdot\alpha}$$

According to Lemma 5, this can be simplified to

$$\delta[P] \Rightarrow \delta'[Q^{\mathcal{H}}_{\mathcal{H}\cdot\alpha}]$$

(for all free variables) since we may assume that the postcondition Q already has WF as a condition. We use this general reduction result when considering the different statements below.

Skip statement. The operational semantics of the skip statement is given by

$$o \mapsto \mathbf{ob}(\delta, \texttt{skip}; \bar{s}) \xrightarrow{empty} o \mapsto \mathbf{ob}(\delta, \bar{s})$$

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

As explained above, $\models \{P\}\, skip\, \{Q\}$ reduces to $\delta[P] \Rightarrow \delta[Q]$. For a local state δ consisting of substitutions $a_i \mapsto v_i$ and $l_j \mapsto u_j$ (for a_i and l_j ranging over fields and method local variables) we may rename v_i and u_j to a_i and l_j respectively, and obtain

$$P \Rightarrow Q$$

for all a_i and l_j and $\mathcal{H}$. For a given postcondition Q, the weakest possible precondition satisfying this is Q. Therefore the axiom $\{Q\}\mathbf{skip}\{Q\}$ defines the weakest precondition. Thus soundness follows and completeness also follows since the precondition of the axiom is the weakest possible according to validity, and since any stronger precondition can be obtained by the imp rule.

Assignment statement. The operational semantics of assignment statement is given by

$$o \mapsto \mathbf{ob}(\delta, v := e; \bar{s}) \xrightarrow{empty} o \mapsto \mathbf{ob}(\delta[v := e], \bar{s})$$

As explained above, $\models \{P\}\, v := e\, \{Q\}$ reduces to $\delta[P] \Rightarrow \delta[v := e][Q]$ which (as explained for **skip**) reduces to

$$P \Rightarrow Q^v_e$$

(for all free variables). Thus, for postcondition Q the weakest possible precondition is Q^v_e which is exactly what is stated in the axiom $\{Q^v_e\}\, v := e\, \{Q\}$.

Return statement. The operational semantics of the return statement is given by

$$o \mapsto \mathbf{ob}(\delta, \mathbf{put}\ e) \xrightarrow{\langle \leftarrow o, \delta[\mathsf{future}], \delta[e]\rangle} \begin{array}{l} o \mapsto \mathbf{ob}(\delta, empty) \\ \delta[\mathsf{future}\,] \mapsto \mathbf{fut}(\delta[e]) \end{array}$$

As explained above, $\models \{P\}\, \mathbf{put}\ e\, \{Q\}$ reduces to

$$\delta[P] \Rightarrow \delta[\mathcal{H} := \mathcal{H} \cdot \langle \leftarrow o, \delta[\mathsf{future}], \delta[e]\rangle][Q]$$

which is

$$P \Rightarrow Q^{\mathcal{H}}_{\mathcal{H}\cdot\langle \leftarrow \mathsf{this}, \mathsf{future}, e\rangle}$$

Thus, for postcondition Q the weakest possible precondition is

$$Q^{\mathcal{H}}_{\mathcal{H}\cdot\langle \leftarrow \mathsf{this}, \mathsf{future}, e\rangle}$$

which is exactly the same as the precondition of the axiom $\{Q^{\mathcal{H}}_{\mathcal{H}\cdot\langle \leftarrow \mathsf{this}, \mathsf{future}, e\rangle}\}\, \mathbf{put}\ e\, \{Q\}$.

Query statement. The operational semantics of query statement is given by

$$\begin{array}{l} u \mapsto \mathbf{fut}(d) \\ o \mapsto \mathbf{ob}(\delta, v := \mathbf{get}\ e; \bar{s}) \\ \mathbf{if}\ \delta[e] = u \end{array} \xrightarrow{\langle o \twoheadleftarrow, u, d\rangle} \begin{array}{l} u \mapsto \mathbf{fut}(d) \\ o \mapsto \mathbf{ob}(\delta[v := d], \bar{s}) \end{array}$$

As explained above, $\models \{P\}\, v := \mathbf{get}\ e\, \{Q\}$ reduces to

$$\delta[P] \Rightarrow \delta[v := d, \mathcal{H} := \mathcal{H} \cdot \langle o \twoheadleftarrow, e, d\rangle][Q]$$

which, as explained above, reduces to

$$P \Rightarrow Q^{v,\mathcal{H}}_{d,\mathcal{H}\cdot\langle \mathsf{this} \twoheadleftarrow, e, d\rangle}$$

(for all d, $\mathcal{H}$, and other free variables) which means

$$\forall d\,.\, P \Rightarrow Q^{v,\mathcal{H}}_{d,\mathcal{H}\cdot\langle \mathsf{this}\twoheadleftarrow, e, d\rangle}$$

Since d may not occur in P, this reduces to

$$P \Rightarrow \forall d\,.\, Q^{v,\mathcal{H}}_{d,\mathcal{H}\cdot\langle \mathsf{this}\twoheadleftarrow, e, d\rangle}$$

Thus, for postcondition Q the weakest possible precondition is

$$\forall d\,.\, Q^{v,\mathcal{H}}_{d,\mathcal{H}\cdot\langle \mathsf{this}\twoheadleftarrow, e, d\rangle}$$

which is exactly the same as the precondition of the axiom $\{\forall v'\,.\, Q^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle \mathsf{this}\twoheadleftarrow, e, v'\rangle}\}\, v := \mathbf{get}\ e\, \{Q\}$.

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

Asynchronous method call statement. The abstract operational semantics of asynchronous method call is given by

$$\begin{array}{lll} o \mapsto \mathbf{ob}(\delta, fr := v!m(\bar{e}); \bar{s}) & & o \mapsto \mathbf{ob}(\delta[fr := u], \bar{s}) \\ H \mapsto h & \rightarrow & u \mapsto \mathbf{msg}(\delta[v], m, \delta[\bar{e}]) \\ \mathtt{if}\ \ u \notin id(h) & & H \mapsto h \cdot \langle o \rightarrow \delta[v], u, m, \delta[\bar{e}]\rangle \end{array}$$

As explained above, $\models \{P\}\, fr := v!m(\bar{e})\, \{Q\}$ reduces to

$$\delta[P] \ \wedge\ u \notin id(\mathcal{H}) \Rightarrow \delta[fr := u, \mathcal{H} := \mathcal{H} \cdot \langle o \rightarrow \delta[v], u, m, \delta[\bar{e}]\rangle][Q]$$

which is, assuming Q has *WF* as a condition according to Lemma 5,

$$P \Rightarrow \forall u \,.\, Q^{fr,\mathcal{H}}_{u,\mathcal{H}\cdot\langle \mathsf{this} \rightarrow v, u, m, \bar{e}\rangle}$$

since *WF* implies uniqueness of future identities in invocation events. Thus, for postcondition Q the weakest possible precondition is

$$\forall fr' \,.\, Q^{fr,\mathcal{H}}_{fr',\mathcal{H}\cdot\langle \mathsf{this} \rightarrow v, fr', m, \bar{e}\rangle}$$

which is exactly the same as the precondition of the axiom $\{\forall fr' . Q^{fr,\mathcal{H}}_{fr',\mathcal{H}\cdot\langle \mathsf{this} \rightarrow v, fr', m, \bar{e}\rangle}\}\, fr := v!m(\bar{e})\, \{Q\}$.

New statement. The abstract operational semantics of object creation is given by

$$\begin{array}{lll} o \mapsto \mathbf{ob}(\delta, v := \mathbf{new}\, C(\bar{e}); \bar{s}) & & o \mapsto \mathbf{ob}(\delta[v := o'], \bar{s}) \\ H \mapsto h & \rightarrow & o' \mapsto \mathbf{ob}(\delta_{init}, init) \\ \mathtt{if}\ \ o' \notin id(h) & & H \mapsto h \cdot \langle o \uparrow o', C, \delta[\bar{e}]\rangle \end{array}$$

As explained above, $\models \{P\}\, v := \mathbf{new}\, C(\bar{e})\, \{Q\}$ reduces to

$$\delta[P] \ \wedge\ o' \notin id(\mathcal{H}) \Rightarrow \delta[v := o', \mathcal{H} := \mathcal{H} \cdot \langle o \uparrow o', C, \bar{e}\rangle][Q]$$

which is, assuming Q has *WF* as a condition according to Lemma 5,

$$P \Rightarrow \forall v' \,.\, Q^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle o \uparrow v', C, \bar{e}\rangle}$$

since *WF* implies uniqueness of object identities in object creation events. Thus, for postcondition Q the weakest possible precondition is

$$\forall v' \,.\, Q^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle o \uparrow v', C, \bar{e}\rangle}$$

which is exactly the same as the precondition of the axiom $\{\forall v' \,.\, Q^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle o \uparrow v', C, \bar{e}\rangle}\}\, v := \mathbf{new}\, C(\bar{e})\, \{Q\}$.

For each basic statement, we have now shown that the reasoning rule expresses the weakest possible precondition according to the semantics. As explained, this ensures soundness and completeness. We have paid special attention to all statements that involve futures and histories, apart from method declarations which are treated next.

7.1.2. Annotated method declarations

We first define validity of annotated method declarations, which has not been considered so far. Then we consider soundness and completeness for reasoning about annotated method declarations.

Definition 10 *(Validity of annotated method declarations).*

$$\begin{array}{l} \models \{P\}\, (m(\bar{x})\{\mathbf{var}\ \ y; \bar{s}\})\, \{Q\} \triangleq \forall o, g, g'', \bar{y}', \bar{z} \,.\, wf(g[H]) \wedge wf(g''[H]) \wedge \\ \qquad g' \xrightarrow[o]{o:\bar{s}} g'' \wedge loc(g, o)[P^{\bar{y}}_{\bar{y}'}] \Rightarrow loc(g'', o)[Q^{\bar{y}}_{\bar{y}'}] \end{array}$$

where $\bar{y}'$ are fresh logical variables and g' denotes $g[H := H \cdot \langle \twoheadrightarrow o, \mathsf{future}, m, \bar{x}\rangle, \bar{y} := default]$. Here *default* represents the default values of the appropriate types, and $\bar{z}$ is the list of logical variables other than $\bar{y}'$. The use of $\bar{y}'$ reflects that $\bar{y}$ in P and Q does not refer to the local variables $\bar{y}$ of the method.

To prove that the reasoning rule

$$\mathsf{method}\ \ \frac{\{P^{\bar{y}}_{\bar{y}'} \wedge \bar{y} = default\}\, \bar{s}\, \{Q^{\bar{y}}_{\bar{y}'}\}}{\{P^{\mathcal{H}}_{\mathcal{H}\cdot\langle \twoheadrightarrow \mathsf{this}, \mathsf{future}, m, \bar{x}\rangle}\}\, (m(\bar{x})\{\mathbf{var}\ \bar{y}; \bar{s}\})\, \{Q\}}$$

is sound, we need to show that the validity of the premise implies the validity of the conclusion. The validity of the conclusion reduces to

$$\forall o, g', g'', \bar{y}', \bar{z}\,.\, wf(g'[H]) \wedge wf(g''[H\ ]) \wedge g'[\bar{y} \mapsto default] \xrightarrow[o:\bar{s}]{o} g'' \wedge loc(g', o)[P^{\bar{y}}_{\bar{y}'}] \Rightarrow loc(g'', o)[Q^{\bar{y}}_{\bar{y}'}]$$

which is exactly the validity of the premise. We therefore have

$$\models \{P^{\mathcal{H}}_{\mathcal{H}\cdot\langle\twoheadrightarrow \mathsf{this},\mathsf{future},m,\bar{x}\rangle}\}\ (m(\bar{x})\{\mathbf{var}\ \bar{y};\bar{s}\})\ \{Q\} \Leftrightarrow \models \{P^{\bar{y}}_{\bar{y}'} \wedge \bar{y} = default\}\ \bar{s}\ \{Q^{\bar{y}}_{\bar{y}'}\}$$

Thus the given Hoare rule is sound.

For completeness, we must prove

$$\vdash \{R\}\ (m(\bar{x})\{\mathbf{var}\ \bar{y};\bar{s}\})\ \{Q\}$$

for any R and Q assuming

$$\models \{R\}\ (m(\bar{x})\{\mathbf{var}\ \bar{y};\bar{s}\})\ \{Q\} \tag{2}$$

Taking P of the above equivalence as $R^{\mathcal{H}}_{pop(\mathcal{H})}$ we get

$$\models \{(R^{\mathcal{H}}\_pop(\mathcal{H}))^{\mathcal{H}}_{\mathcal{H}\cdot\langle\twoheadrightarrow \mathsf{this},\mathsf{future},m,\bar{x}\rangle}\}\ (m(\bar{x})\{\mathbf{var}\ \bar{y};\bar{s}\})\ \{Q\} \Leftrightarrow \models \{(R^{\mathcal{H}}_{pop(\mathcal{H})})^{\bar{y}}_{\bar{y}'} \wedge \bar{y} = default\}\ \bar{s}\ \{Q^{\bar{y}}_{\bar{y}'}\}$$

which reduces to

$$\models \{R\}\ (m(\bar{x})\{\mathbf{var}\ \bar{y};\bar{s}\})\ \{Q\} \Leftrightarrow \models \{(R^{\mathcal{H}}_{pop(\mathcal{H})})^{\bar{y}}_{\bar{y}'} \wedge \bar{y} = default\}\ \bar{s}\ \{Q^{\bar{y}}_{\bar{y}'}\}$$

since $(R^{\mathcal{H}}_{pop(\mathcal{H})})^{\mathcal{H}}_{\mathcal{H}\cdot\langle\twoheadrightarrow \mathsf{this},\mathsf{future},m,\bar{x}\rangle}$ is the same as R. By (2) and the induction hypothesis (i.e., that completeness applies to structurally simpler programs) we have

$$\vdash \{(R^{\mathcal{H}}_{pop(\mathcal{H})})^{\bar{y}}_{\bar{y}'} \wedge \bar{y} = default\}\ \bar{s}\ \{Q^{\bar{y}}_{\bar{y}'}\}$$

And the method proof rule gives us

$$\vdash \{R\}\ (m(\bar{x})\{\mathbf{var}\ \bar{y};\bar{s}\})\ \{Q\}$$

Consequently, we have completeness for reasoning about annotated method declarations.

8. Addition of non-blocking queries and process control

We consider here an extension of the considered core language by constructs for process control, allowing conditional and unconditional process release and release while waiting for a future to be resolved, thereby avoiding blocking.

suspend	unconditional release
await b	conditional release
$x :=$ **await** *future*	releasing query
$x :=$ **await** $o.m(\bar{e})$	releasing call

Conditional release allows releasing the processor while waiting for a condition b to be satisfied. Unconditional release may be defined in terms of conditional release as follows:

$$\mathbf{suspend} \triangleq \mathbf{await}\ true$$

A releasing query allows releasing the processor while waiting for the future to be resolved. Similarly, a releasing call statement releases while waiting for the call to be completed. A releasing call is defined in terms of releasing query, waiting for the associated future:

$$x := \mathbf{await}\ o.m(\bar{e}) \triangleq v := o!m(\bar{e}); x := \mathbf{await}\ v$$

where v is a fresh future variable.

Thus it suffices to formalize conditional release and releasing query.

8.1. Operational semantics

The operational semantics is given in Fig. 9. The old rules for the core language are unchanged, apart from adding the queue element to the objects considered in the rules. The old notion of wellformedness is already compatible with the added statements. Notice that an idle object (i.e., an object with an empty statement list) may choose to continue processing a suspended method activation from the queue, provided it is enabled, or start a new method activation by the old start rule.

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

$$\text{await bool}: \quad \begin{array}{l} o \mapsto \mathbf{ob}((a|l), \texttt{await}\ \ b; \bar{s}, q) \\ \xrightarrow{empty} \ o \mapsto \mathbf{ob}((a|empty), empty, q \cdot (l, \texttt{await}\ \ b; \bar{s})) \end{array}$$

$$\text{await fut}: \quad \begin{array}{l} o \mapsto \mathbf{ob}((a|l), x := \texttt{await}\ \ e; \bar{s}, q) \\ \xrightarrow{empty} \ o \mapsto \mathbf{ob}((a|empty), empty, q \cdot (l, x := \texttt{await}\ \ e; \bar{s})) \end{array}$$

$$\text{choose bool}: \quad \begin{array}{l} o \mapsto \mathbf{ob}((a|l'), empty, (l, \texttt{await}\ \ b; \bar{s}) \cdot q) \\ \xrightarrow{empty} \ o \mapsto \mathbf{ob}((a|l), \bar{s}, q) \\ \quad \texttt{if} \quad (a|l)[b] = true \end{array}$$

$$\text{choose fut}: \quad \begin{array}{l} u \mapsto \mathbf{fut}(d) \\ o \mapsto \mathbf{ob}((a|l'), empty, (l, x := \texttt{await}\ \ e; \bar{s}) \cdot q) \\ \xrightarrow{empty} \ o \mapsto \mathbf{ob}((a|l), x := d; \bar{s}, q) \\ u \mapsto \mathbf{fut}(d) \\ \quad \texttt{if} \quad (a|l)[e] = u \end{array}$$

Fig. 9. Operational rules for process control.

8.2. Axiomatic semantics

Let e is an expression denoting a future. For partial correctness reasoning, we may define $x := \texttt{await}\ e$ by

$$\texttt{suspend}\ ;\ x := \texttt{get}\ \ e$$

since the effect of temporary blocking is not visible within partial correctness. Thus, $\{P\}\, x := \texttt{await}\ e\, \{Q\}$ is the same as $\{P\}\, \texttt{await}\ true; x := \texttt{get}\ e\, \{Q\}$. Hence, it remains to give reasoning rules for conditional release.

Reasoning over conditional release points can be done by means of the local invariant I. The invariant I must be proved to hold after initialization and that it is maintained by each method. The rule below ensures that the invariant is reestablished at release points.

$$\{I \wedge L \wedge h_0 = \mathcal{H}\}\ \texttt{await}\ \ b\ \{b \wedge I \wedge L \wedge h_0 \leq \mathcal{H}\}$$

where L is an assertion referring to method parameters, local variables, and logical variables only. Thus the invariant provides information about the fields (and class parameters), and the local assertion L allows reasoning about local variables and parameters. A logical variable h_0 is used to express that the history may only be appended. Note that the triple $\{h_0 = \mathcal{H}\}\, s\, \{h_0 \leq \mathcal{H}\}$ can be derived for all statements s.

The reasoning rule is somewhat simpler than in [17]. The present version of conditional release will release even when the condition is satisfied.

8.3. Example

We reconsider the previous example, and show a more efficient implementation of the *publish* method, using a releasing query statement to avoid blocking the proxy object. This allows a higher degree of concurrent activity.

```
Void publish(Fut<News> fut){
  var News ns = None;
  ns := await fut;
  myClients!signal(ns);
  if nextProxy = null then s!produce() else nextProxy!publish(fut) fi}
```

We are still able to prove the same local and global invariants as before, because the local invariants are defined by means of subsequences of the history. In particular, the local history of the proxy object may grow at the process releasing point which is between $\langle \twoheadrightarrow \text{this}, \_, publish, u\rangle$ and $\langle \text{this} \twoheadleftarrow, u, ns\rangle$. Since the class invariant defined in Section 6 does not require tight subsequence of history, the class invariant is maintained here.

9. Discussion

The setting of concurrent objects communicating solely by asynchronous method calls allows compositional reasoning in a way which resembles sequential reasoning. The main complication compared to sequential reasoning is the manipulation of the local history variable. Disjointness of alphabets for disjoint objects implies that local histories are disjoint, in the sense that processing steps made by one object do not affect the invariants of other objects. This ensures local reasoning inside each class, and composition of concurrent objects amounts to conjunction of invariants and relating local histories to the global history. We have extended this framework to futures, supporting local reasoning about futures based on locally visible events involving futures. Thus in order to obtain global knowledge about futures one relies on the composition rule. We find that specification of futures is tightly connected to that of the objects creating and operating on the futures. Our approach makes this possible.

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

We have shown how to extend the framework to the high-level process control statements (suspension and await statements) of the Creol/ABS languages, including constructs for fetching futures without blocking. This is useful to obtain more efficient computations and to avoid deadlocks. It also opens up for local calls of the form $x :=$ **await** this.$m(\ldots)$ which are then executed before x is assigned the resulting future. In contrast a local call of the form $x :=$ this.$m(\ldots)$ would deadlock since our semantics would block, as shown in Section 5.1. Reentrant handling of such local calls could be done as in Creol [8]. However, efficient implementation and specialized reasoning of local calls are omitted here for simplicity reasons as we focus on object interaction mechanisms.

Our setting does not allow remote access to fields, as found in several mainstream object-oriented languages including Java and Spec$^\sharp$ [22]. This would necessitate constructs for programming critical regions, for instance by means of locks using a thread-based concurrency model. Remote field access would severely complicate our setting. In order to do compositional reasoning with histories one would need to consider read and write operations to shared variables [23,24], as well as reasoning problems related to aliasing. Non-observable events would then be reflected in the histories, making both specification and reasoning much more low-level and much more complicated. As an alternative to history-based specifications, several approaches for Java and Spec$^\sharp$ use specifications with model variables. However, one needs more fine-grained control of when a class invariant holds. Rather than invariants maintained by each methods body (and reestablished at release points when considering suspend and await statements), one would need to consider invariants maintained by all atomic statements, as in [23], or include explicit control of when an invariant holds, for instance based on packing and unpacking of invariants as in [25].

Aliasing occurs when more than one variable refer to the same object. Therefore, if both variables v_1 and v_2 refer to the same object o, modifications of v_1 on o's fields affect the value of the variable v_2 as well. Thus with remote field assignments the classical assignment axiom (as in Fig. 7) would not be sound. We consider a programming language in which access to the internal state variables of other objects is only possible through remote method calls. This is the reason why the classical assignment axiom is sound in our case.

Our setting extended with await statements allows *callbacks*, i.e., inside a method body one may make calls to the caller object or other objects received as parameters. The implicit *caller* parameter (typed by a cointerface as in Creol) opens up for callbacks to the caller object without passing that object as an explicit parameter [8]. An object o making a non-blocking call, say $x :=$ **await** $r.m(\bar{e})$, is free to handle such callbacks, since the execution of the caller is suspended. The callback from r can then be executed as soon as any ongoing method execution of the object o reaches the end or a release point. In this way callbacks are possible without reentry mechanisms. (As before, calls for which the future is not needed by the caller, also allow callbacks, as was illustrated in the *produce* method of class *Service* in Fig. 3.) In contrast a callback from r while the object o is blocking for some future would lead to deadlock. Therefore deadlock can be avoided if all queries are non-blocking, i.e., using the **await** mechanism.

Furthermore, a blocking query made by an object o will not cause deadlock if the callee does not cause a query to o. In order to get a syntactic guarantee for deadlock freedom, one could consider a partial ordering implying that there is no circular query chain starting with a blocking query (considering the implicit queries defined by calls with dot-notation). For simplicity we assume that each class has exactly one interface and we ignore complications with interface inheritance. We may then consider a strict partial ordering of interfaces, and restrict (blocking and non-blocking) queries to smaller interfaces according to the ordering. The body of a method used to implement a certain interface may query a future if the associated callee has a smaller interface according to the ordering. A complication is that a query on a future may not identify the callee (the object producing the future value). However, if the call introducing the future is in the same body, one may statically associate a callee with the future. For futures appearing as formal parameters it suffices to consider all possible callees associated with the corresponding actual parameters – in the whole program. (A similar discussion applies to futures appearing as method results.) This gives a static approximation of the possible callees. When the callee is of the same interface as the caller we may use the partial order implied by ownership, i.e., we may allow calls to (statically known) child objects.

The example in Fig. 3 can be seen to be deadlock-free according to this strategy. Class *Service* has one blocking query in method *subscribe*, given by the call *lastProxy.add*. This query is OK with *ProxyI* less than *ServiceI*. Class *Proxy* has one blocking query in method *add*, given by the call *nextProxy.add*. This query is made to a child object of the same interface. In method *publish* there is a query on *fut*, which is OK with *ProducerI* less than *ProxyI*, since *prod* is the (only) associated callee of the corresponding actual parameter. From Appendix A, we see that the remaining classes are OK with *NewsProducerI* less than *ProducerI*. Clearly, the interface ordering required is a proper partial order.

10. Related work

Models for asynchronous communication without futures have been explored for process calculi with buffered channels [11], for agents with message-based communication [26], for method-based communication [27], and in particular for Java [28]. Reasoning about distributed and object-oriented systems is challenging, due to the combination of concurrency, compositionality and object orientation. Moreover, the gap in reasoning complexity between sequential and concurrent, object-oriented systems makes tool-based verification difficult in practice. A recent survey of these challenges can be found in [29]. The present approach follows the line of work based on communication histories to model object communication events in a distributed setting [29,10,30,17,31,32,11,33,24]. Objects are concurrent and interact solely by method calls and

futures, and remote access to object fields are forbidden. Object generation is reflected in the history by means of object creation events. This enables compositional reasoning of concurrent systems with dynamic generation of objects and aliasing (while avoiding alias reasoning problems).

Futures, first introduced in Multilisp [5] are language constructs that improve concurrency and data flow synchronization in a natural and transparent way. Some frameworks allow futures to be passed to other processes. Such shared futures are called *first class futures*, which offer greater flexibility in application design and can significantly improve concurrency in object-oriented paradigms. A reasoning system for asynchronous method communication without futures is introduced in [17], from which we redefine the six-event semantics to reflect actions on first class futures, ending up with a five-event semantics. The semantics provides a clean separation of the activities of the different objects, which leads to disjointness of local histories. Thus, object behavior can be specified in terms of the observable interaction of the current object only. This simplifies the model and the accompanied proof system, thereby reducing the gap between reasoning about sequential systems and concurrent object-oriented systems. Especially, when reasoning about a class, it is not necessary to explicitly account for the activity of objects in the environment. A class invariant defines a relation between the inner state and the observable communication of instances, and can be verified independently for each class. The class invariant can be instantiated for each object of the class, resulting in a history invariant over the observable behavior of the object. Compositional reasoning is ensured as history invariants may be combined to form global system specifications. The composition rule is similar to [17], which is inspired by previous approaches [33,24].

By creating unique references for method calls, the *label* construct of Creol [8] resembles futures, as callers may postpone reading result values. Verification systems capturing Creol labels can be found in [29,31]. However, a label reference is local to the caller, and cannot be shared with other objects. In [29], a compositional verification system for *Creol* is presented.

A reasoning system for asynchronous method calls and futures has been presented in [34], using a combination of global and local invariants. Futures are treated as visible objects rather than reflected by events in histories. In contrast to our work, global reasoning is obtained by means of global invariants, rather than by compositional rules. Thus the environment of a class must be known at verification time. The completeness proof is based on a semantic characterization of the global invariant in terms of futures and two history variables. One denotes the sequence of generated communication events, which is updated by method calls, upon each method activation, and by each return statement. The other records the local state in order to reason about the internal process queue.

A compositional reasoning system for *ABS* with futures has been presented in our earlier work [9] based on local communication histories. We here show that a revised and simplified version of this system is sound and (relatively) complete with respect to a revised version of an operational semantics which incorporates a notion of global communication history. The present system is based on disjointness of events and uses five kinds of events, i.e., four related to futures and one for object creation, which is a simplification compared to earlier work with disjoint events. Soundness of the composition rule is studied in [20].

A compositional Hoare Logic for concurrent processes (objects) is presented in [35]. Soundness and relative completeness are proved with respect to the operational semantics. Communication histories capture the sequences of output messages, input messages and the generated object identities. History information is used in both the operational semantics and the reasoning system. In contrast to the present work, communication is by message passing rather than by method interaction, the objects communicate through FIFO channels, and futures are not considered.

In [36], a reasoning system for a subset of Eiffel is presented. Soundness and completeness of the reasoning system is proved with respect to the given operational semantics. However, concurrency is not considered as the language is sequential. Object orientation is the main focus, and in particular exception handling (but not futures) is included. In Eiffel, so-called *once routines* cache the first execution result for the later received invocations, where the arguments are irrelevant. Therefore, a "once routine" has less flexibility than the concept of shared futures in *ABS*.

11. Conclusion

In this paper we present a sound and relative complete compositional reasoning system for distributed objects with shared futures, based on a general concurrency and communication model centered around concurrent objects, asynchronous methods calls, and futures. This model is chosen due to advantages with respect to program reasoning, while integrating asynchronous interaction and object orientation in a natural manner. Compositional reasoning is facilitated by expressing object properties in terms of observable interaction between the object and its environment, recorded on communication histories. Object generation is reflected in the history by means of object creation events. A method call cycle with multiple future readings is reflected by four kinds of events, giving rise to disjoint communication alphabets for different objects. Specifications in terms of history invariants may then be derived independently for each object and composed in order to derive properties for concurrent object systems.

At the class level, invariants define relationships between class attributes and the observable communication of class instances. The presented Hoare style system is proven sound and relatively complete with respect to the given operational semantics. This system is easy to apply in the sense that class reasoning is similar to standard sequential reasoning, but with the addition of effects on the local history for statements involving futures. In particular, reasoning inside classes is not affected by the complexity of concurrency and synchronization; and one may express assumptions about inputs from the environment when convenient.

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

At the global level, the notion of wellformedness allows us to connect the information in the different local invariants of each object, according to the natural event ordering (as expressed in Fig. 4). The notion of wellformedness and event kinds are simpler than in earlier work, and thus local reasoning as well as global reasoning are simplified. The presented reasoning system is illustrated by a version of the Publisher–Subscriber example. As seen in the example the presence of future implies that specifications typically involve existentially quantified future identities to express causal relationships in communication patterns. The example also shows that histories are highly expressive, letting projections and functions over the history express minimal requirements.

An interpreter for the considered core language based on the operational semantics has been implemented in Maude, including calculation of the global history. With the underlying tool support of Maude one can prototype and model check programs written in the language with respect to properties involving local and global histories. An *ABS* reasoning system is currently being implemented within the KeY framework at Technische Universität Darmstadt. The tool support from KeY for (semi-)automatic verification is valuable for verifying *ABS* programs. The current axiomatic system has been integrated in the tool. In [37] we compare initial experiments with this extension of the KeY system with the developed runtime checker for *ABS* programs with futures.

Appendix A. Complete code of publisher–subscriber example

```
data News = E1 | E2 | E3 | E4 | E5 | None;

interface ServiceI{
    Void subscribe(ClientI cl);
    Void produce()}

interface ProxyI{
    ProxyI add(ClientI cl);
    Void publish(Fut<News> fut)}

interface ProducerI{
    News detectNews()}

interface NewsProducerI{
    Void add(News ns);
    News getNews();
    List<News> getRequests()}

interface ClientI{
    Void signal(News ns)}


class Service(Int limit, NewsProducerI np) implements ServiceI{
    ProducerI prod; ProxyI proxy; ProxyI lastProxy;
    {prod := new Producer(np); proxy := new Proxy(limit,this); lastProxy := proxy; this!produce()}

    Void subscribe(ClientI cl){lastProxy := lastProxy.add(cl)}

    Void produce(){var Fut<News> fut := prod!detectNews(); proxy!publish(fut)}}


class Proxy(Int limit, ServiceI s) implements ProxyI{
    List<ClientI> myClients := Nil; ProxyI nextProxy;

    ProxyI add(ClientI cl){
      var ProxyI lastProxy = this;
      if length(myClients) < limit then myClients := appendright(myClients, cl)
      else if nextProxy == null then nextProxy := new Proxy(limit,s) fi;
      lastProxy := nextProxy.add(cl) fi; put lastProxy}

    Void publish(Fut<News> fut){
      var News ns = None;
      ns = fut.get; myClients!signal(ns);
      if nextProxy == null then s!produce() else nextProxy!publish(fut) fi}}
```

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

```
class Producer(NewsProducerI np) implements ProducerI{
    News detectNews(){
      var List<News> requests := Nil; News news := None;
      requests := np.getRequests();
      while requests == Nil do requests := np.getRequests() od
      news := np.getNews(); put news}}

class NewsProducer implements NewsProducerI{
    List<News> requests := Nil;
    Void add(News ns){requests := appendright(requests,ns)}
    News getNews(){var News firstNews := head(requests); requests := tail(requests); put firstNews}
    List<News> getRequests(){put requests}}

class Client implements ClientI{
    News news := None;
    Void signal(News ns){news := ns}}
```

We have here augmented the given core language with ABS syntax for data types.

References

[1] International Telecommunication Union, Open Distributed Processing – Reference Model, parts 1–4, Tech. Rep., ISO/IEC, Geneva, Jul. 1995.
[2] A. Ahern, N. Yoshida, Formalising Java RMI with explicit code mobility, Theor. Comput. Sci. 389 (3) (2007) 341–410.
[3] O.-J. Dahl, O. Owe, Formal methods and the RM-ODP, Tech. Rep. Research Report 261, Dept. of Informatics, Univ. of Oslo, Full version of a paper presented at NWPT'98: Nordic Workshop on Programming Theory, Turku, 1998.
[4] H.G. Baker Jr., C. Hewitt, The incremental garbage collection of processes, in: Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages, ACM, New York, NY, USA, 1977, pp. 55–59.
[5] R.H. Halstead Jr., Multilisp: a language for concurrent symbolic computation, ACM Trans. Program. Lang. Syst. 7 (4) (1985) 501–538.
[6] B.H. Liskov, L. Shrira, Promises: linguistic support for efficient asynchronous procedure calls in distributed systems, in: D.S. Wise (Ed.), Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88), ACM Press, 1988, pp. 260–267.
[7] A. Yonezawa, J.-P. Briot, E. Shibayama, Object-oriented concurrent programming in ABCL/1, in: Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86), SIGPLAN Not. 21 (11) (1986) 258–268.
[8] E.B. Johnsen, O. Owe, An asynchronous communication model for distributed concurrent objects, Softw. Syst. Model. 6 (1) (2007) 35–58.
[9] C.C. Din, J. Dovland, O. Owe, Compositional reasoning about shared futures, in: G. Eleftherakis, M. Hinchey, M. Holcombe (Eds.), Proc. International Conference on Software Engineering and Formal Methods (SEFM'12), in: LNCS, vol. 7504, Springer-Verlag, 2012, pp. 94–108.
[10] M. Broy, K. Stølen, Specification and Development of Interactive Systems, Monographs in Computer Science, Springer-Verlag, 2001.
[11] C.A.R. Hoare, Communicating Sequential Processes, International Series in Computer Science, Prentice Hall, 1985.
[12] O.-J. Dahl, Object-oriented specifications, in: Research Directions in Object-Oriented Programming, MIT Press, Cambridge, MA, USA, 1987, pp. 561–576.
[13] O.-J. Dahl, Verifiable Programming, International Series in Computer Science, Prentice Hall, New York, NY, 1992.
[14] E. Ábrahám, I. Grabe, A. Grüner, M. Steffen, Behavioral interface description of an object-oriented language with futures and promises, J. Log. Algebr. Program. 78 (7) (2009) 491–518.
[15] A.S.A. Jeffrey, J. Rathke, Java Jr.: fully abstract trace semantics for a core Java language, in: Proc. European Symposium on Programming, in: LNCS, vol. 3444, Springer-Verlag, 2005, pp. 423–438.
[16] B. Alpern, F.B. Schneider, Defining liveness, Inf. Process. Lett. 21 (4) (1985) 181–185.
[17] C.C. Din, J. Dovland, E.B. Johnsen, O. Owe, Observable behavior of distributed systems: component reasoning for concurrent objects, J. Log. Algebr. Program. 81 (3) (2012) 227–256.
[18] Full ABS Modeling Framework (Mar 2011), deliverable 1.2 of project FP7-231620, (HATS), available at http://www.hats-project.eu.
[19] J. Dovland, E.B. Johnsen, O. Owe, M. Steffen, Lazy behavioral subtyping, J. Log. Algebr. Program. 79 (7) (2010) 578–607.
[20] C.C. Din, O. Owe, Compositional and sound reasoning about active objects with shared futures, Research Report 437, Dept. of Informatics, University of Oslo, Feb. 2014, FAC J., submitted for publication, available at http://urn.nb.no/URN:NBN:no-41224.
[21] K.R. Apt, Ten years of Hoare's logic: a survey — part I, ACM Trans. Program. Lang. Syst. 3 (4) (1981) 431–483.
[22] M. Barnett, K.R.M. Leino, W. Schulte, The Spec♯ programming system: an overview, in: Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, CASSIS'04, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 49–69, http://dx.doi.org/10.1007/978-3-540-30569-9_3.
[23] O. Owe, Axiomatic treatment of processes with shared variables revisited, Form. Asp. Comput. 4 (4) (1992) 323–340.
[24] N. Soundararajan, A proof technique for parallel programs, Theor. Comput. Sci. 31 (1–2) (1984) 13–29.
[25] K.M. Leino, P. Müller, A verification methodology for model fields, in: P. Sestoft (Ed.), Programming Languages and Systems, in: Lecture Notes in Computer Science, vol. 3924, Springer, Berlin, Heidelberg, 2006, pp. 115–130, http://dx.doi.org/10.1007/11693024_9.
[26] G. Agha, S. Frolund, W. Kim, R. Panwar, A. Patterson, D. Sturman, Abstraction and modularity mechanisms for concurrent computing, IEEE Parallel Distrib. Technol. 1 (2) (1993) 3–14.
[27] B. Morandi, S.S. Bauer, B. Meyer, SCOOP – a contract-based concurrent object-oriented programming model, in: P. Müller (Ed.), Advanced Lectures on Software Engineering, LASER Summer School 2007/2008, in: LNCS, vol. 6029, Springer, 2008, pp. 41–90.
[28] K.E.K. Falkner, P.D. Coddington, M.J. Oudshoorn, Implementing asynchronous remote method invocation in java, in: 6th Australian Conference on Parallel and Real-Time Systems, Springer-Verlag, 1999, pp. 22–34.
[29] W. Ahrendt, M. Dylla, A system for compositional verification of asynchronous objects, Sci. Comput. Program. 77 (12) (2012) 1289–1309.
[30] O.-J. Dahl, Can program proving be made practical?, in: M. Amirchahy, D. Néel (Eds.), Les Fondements de la Programmation, Institut de Recherche d'Informatique et d'Automatique, Toulouse, France, 1977, pp. 57–114.
[31] J. Dovland, E.B. Johnsen, O. Owe, Verification of concurrent objects with asynchronous method calls, in: Proceedings of the IEEE International Conference on Software Science, Technology & Engineering (SwSTE'05), IEEE Computer Society Press, 2005, pp. 141–150.

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

[32] J. Dovland, E.B. Johnsen, O. Owe, Observable behavior of dynamic systems: component reasoning for concurrent objects, in: D. Goldin, F. Arbab (Eds.), Proc. Workshop on the Foundations of Interactive Computation (FInCo'07), in: Electr. Notes Theor. Comput. Sci., vol. 203, Elsevier, 2008, pp. 19–34.
[33] N. Soundararajan, Axiomatic semantics of communicating sequential processes, ACM Trans. Program. Lang. Syst. 6 (4) (1984) 647–662.
[34] F.S. de Boer, D. Clarke, E.B. Johnsen, A complete guide to the future, in: R. de Nicola (Ed.), Proc. 16th European Symposium on Programming (ESOP'07), in: LNCS, vol. 4421, Springer-Verlag, 2007, pp. 316–330.
[35] F.S. de Boer, A Hoare logic for dynamic networks of asynchronously communicating deterministic processes, Theor. Comput. Sci. 274 (2002) 3–41.
[36] M. Nordio, C. Calcagno, P. Müller, B. Meyer, Soundness and completeness of a program logic for Eiffel, Tech. Rep. 617, ETH, Zurich, 2009.
[37] C.C. Din, O. Owe, R. Bubel, Runtime assertion checking and theorem proving for concurrent and distributed systems, in: Proceedings of the 2nd Intl. Conf. on Model-Driven Engineering and Software Development, Modelsward'14, SCITEPRESS, 2014, pp. 480–487.

Please cite this article in press as: C.C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (2014), http://dx.doi.org/10.1016/j.jlamp.2014.03.003

Appendix C

Towards the typing of resource deployment

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

```
// available deployment components declaration:
  data DCData = d0, d1 ;

//main statement:
    D x = new cog D() ;
    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:

```
  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

$\mathbb{r} ::= \_ \mid X \mid [cog{:}c, \overline{x}{:}\overline{\mathbb{r}}] \mid fut(\mathbb{r})$	future record
$\mathbb{b} ::= \mathtt{0} \mid \langle c \mapsto c' \rangle^{n \div n} \mid \langle c \mapsto \mathtt{d} \rangle^{n \div n} \mid \langle \mathtt{C!m}(\overline{\mathbb{r}}) \to \mathbb{r}' \rangle^{m \div n}$ $\mid \mathbb{b} + \mathbb{b} \mid \mathbb{b} \,[]\, \mathbb{b} \mid \langle \mathbb{b} \rangle^{m \div n}$	behavioural type

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*) $fut(\mathbb{r})$, 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

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

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 $(\mathbb{r}, \mathbb{r}')\ \{\mathbb{b}_{\mathtt{m}}\} \to \_$, where $\mathbb{r}$ and $\mathbb{r}'$ are the future records of the receiver of the method and of the argument, respectively, $\mathbb{b}_{\mathtt{m}}$ 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 $\mathbb{b}_{\mathtt{m}}$. It is worth to notice that cog names occurring in $\mathbb{b}_{\mathtt{m}}$ may be *not bound*. These *free names* correspond to `new cog` instructions.

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

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

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

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

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

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

Method `multi_create` in `D` has type:

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

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 $[start \mapsto \{\mathtt{d0}\}]$ 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

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

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

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

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:

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

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

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

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

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 $\mathbb{b}_8 = \langle \mathtt{C!m}([cog : c_1], [cog : c_2]) \to \_\rangle^{7 \div 7} \,[\!]\, \langle \langle c_2 \mapsto c_3\rangle^{1 \div 1} + \langle c_2 \mapsto \mathtt{d1}\rangle^{1 \div 1}\rangle^{7 \div 8} \,[\!]\, \langle \mathtt{C!m}([cog : c_3], [cog : c_2]) \to \_\rangle^{8 \div 9}$.

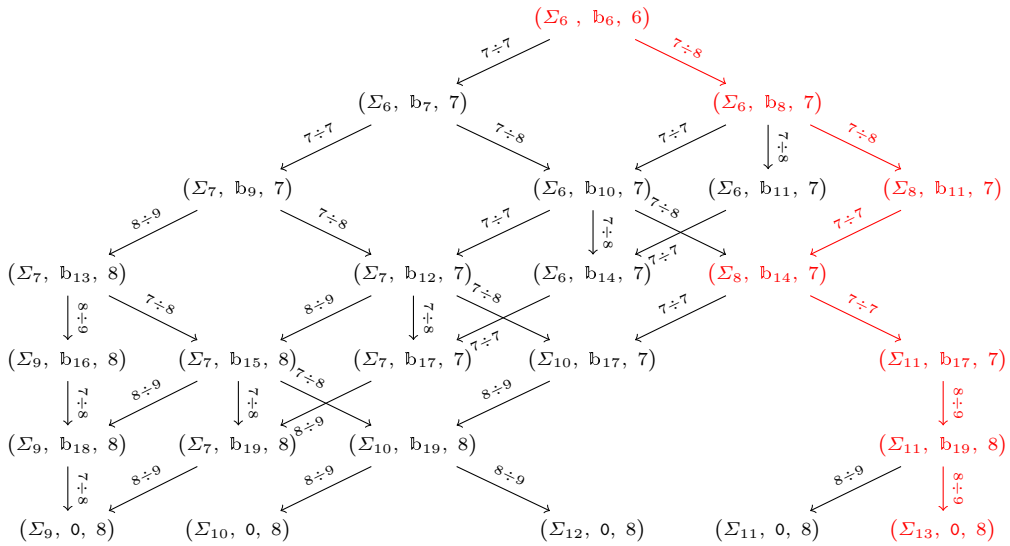


Fig. 1. An example of transition system of behavioural types

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 $[\mathit{start} \mapsto \{\mathtt{d3}\},\ c_1 \mapsto \{\mathtt{d0}\},\ c_2 \mapsto \{\mathtt{d2}\},\ c_3 \mapsto \{\mathtt{d2}\}]$. and $\mathbb{b}_{11}$ is $\langle \mathtt{C!m}([\mathit{cog} : c_1], [\mathit{cog} : c_2]) \to \_ \rangle^{6 \div 6} \,[]\, \langle \mathtt{C!m}([\mathit{cog} : c_3], [\mathit{cog} : c_2]) \to \_ \rangle^{7 \div 8}$.

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 $\mathbb{b}_{14} = \langle \langle c_1 \mapsto c_2 \rangle^{1 \div 1} + \langle c_1 \mapsto \mathtt{d1} \rangle^{1 \div 1} \rangle^{7 \div 7} \,[]\, \langle \mathtt{C!m}([\mathit{cog} : \mathtt{c}_3], [\mathit{cog} : \mathtt{c}_2]) \to \_ \rangle^{8 \div 9}$.

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 $\Sigma_{11} = [\mathit{start} \mapsto \{\mathtt{d3}\},\ c_1 \mapsto \{\mathtt{d1}\},\ c_2 \mapsto \{\mathtt{d2}\},\ c_3 \mapsto \{\mathtt{d2}\}]$ and $\mathbb{b}_{17} = \langle \mathtt{C!m}([\mathit{cog} : c_3], [\mathit{cog} : c_2]) \to \_ \rangle^{8 \div 9}$. 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 $\Sigma_{13} = [\mathit{start} \mapsto \{\mathtt{d3}\},\ c_1 \mapsto \{\mathtt{d1}\},\ c_2 \mapsto \{\mathtt{d2}\},\ c_3 \mapsto \{\mathtt{d1}\}]$.

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

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

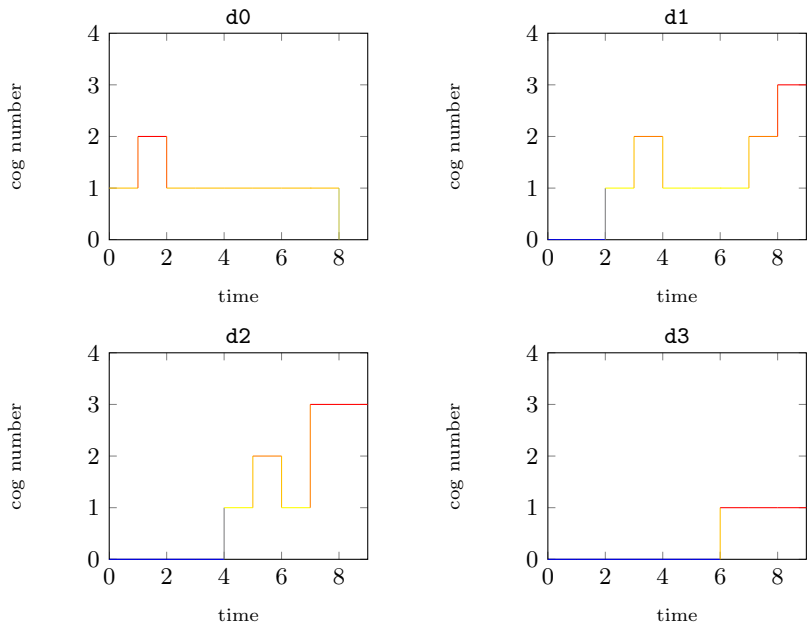
For example, letting the deployment environments of Figure 1 be

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

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:

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

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

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

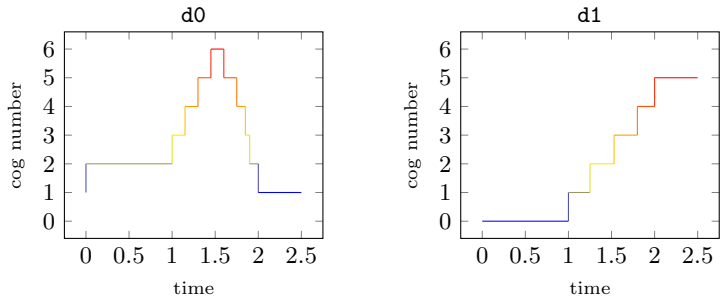
where

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

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.