A Component Model for the ABS Language*

Michaël Lienhardt[1], Ivan Lanese[1], Mario Bravetti[1], Davide Sangiorgi[1], Gianluigi Zavattaro[1], Yannick Welsch[2], Jan Schäfer[2], and Arnd Poetzsch-Heffter[2]

[1] Focus Team, University of Bologna, Italy
{lienhard,lanese,bravetti,davide.sangiorgi,zavattar}@cs.unibo.it
[2] Software Technology Group, University of Kaiserslautern, Germany
{welsch,jschaefer,poetzsch}@cs.uni-kl.de

Abstract. Finding good abstractions to model and express *partial update*, *mobility* and *wrapping* in object-oriented systems remains challenging. In this paper, we propose Comp, a process calculus approach for component models that merges aspects of object-orientation and *evolution*. The key features of Comp are: a hierarchical structure of components; the capacity to move, update, wrap components; method interfaces for components; and some isolation capacities to encode distribution and wrapping. Specifically, we introduce the syntax of Comp and formulate its operational semantics. We show a number of examples of use of Comp, with particular emphasis on common evolution patterns for components.

1 Introduction

Evolution is an important issue in complex software systems. The needs and requirements on a system may change over time. This may happen because the original specification was incomplete or ambiguous, or because new needs arise that had not been predicted at design time. As designing and deploying a system is costly, it is important that the system supports operations to adapt it to changes in the surrounding environment. By *dynamic evolution* of a system we refer to the possibility that functionalities offered by the system change over time. This may involve reconfiguring and updating applications to meet new requirements and new operating conditions, which were unexpected when the application was developed and deployed.

The goal of this work is to isolate interesting constructs for expressing dynamic evolution that can be easily integrated into existing object-oriented programming and modeling languages. More specifically, we are interested in integrating these constructs into the modeling language ABS [10], a language developed within the HATS European project [12], which is based on ideas from Creol [13,9] and JCoBox [27]. The ABS language is a formally defined object-oriented language whose purpose is to support the design of concurrent and distributed software systems, in particular by providing high-level specification and checking facilities. To integrate well with ABS (and similar object-oriented languages), we

* Partly funded by the EU project FP7-231620 HATS.

adopt its basic scheme to handle components[1]. Components are represented by objects. The object's methods enable communication between the object and its environment. Furthermore, the same language specification technique is used to foster the integration between the calculus and the formal analysis techniques developed for ABS and Creol [9,1].

Many different component models were developed over the past decade, like OSGi [2], Fractal [4], COM [8], Java Beans [30] and others [3,21,23]. These models focus on aspects different from those of Comp (see discussion in Sect. 5). If at all possible, dynamic evolution steps have to be handcoded in these models (e.g. in OSGi, components can be stopped and replaced by new versions, but this has to be programmed using the framework API). In particular, a high-level formal analysis of such steps is not supported. The central aspects and goals of Comp are summarized in the following.

Firstly, these previous models [2,4,8,30] are not developed for formal analysis and do not provide a formal semantics needed for verification. In particular, they are more or less coupled to complex programming languages and APIs written in these languages which makes it difficult to identify an analyzable formal core.

The second aspect is about interface specifications based on typing and ports. Many component models (e.g. [4,18,23]) support typed input and output ports to allow static checking of composition. To focus on dynamic evolution and keep the calculus manageable, we do not consider typing aspects in the core calculus presented in this paper. Similar to objects, a component has an interface consisting of a set of (untyped) methods. The component can communicate via channels with other components. In Sect. 4, we show how the basic constructs of the core calculus can be used to model more structured connections between components. An extension to static analysis techniques (e.g., type systems) is planned as future work.

The third aspect concerns scope and component visibility. In many component models [4,22,29], the hierarchical structure of a component system is rigid: the boundary of a component a hides all the inner components, which are unreachable from a's environment. Other component models do not support nesting of components at all. Comp provides component nesting and flexibility with respect to visibility of internal components in order to model common patterns of distributed systems that involve sharing of resources.

The last issue is about the *passivation* mechanism provided by some of the models such as the Kell-calculus [29] or MECo [22]. Passivation allows the programmer to freeze a component and capture it; the component can then be sent around at will, with even the possibility of duplicating it. Passivation is very powerful, but makes it quite hard to ensure safety of reconfiguration and to prove properties of systems. Also, the practical relevance of the act of copying a running component is dubious.

The language that we propose in this paper, called Comp, addresses the four issues above as follows:

[1] As we focus on dynamic aspects, components here are runtime entities, often called component instances to distinguish them from the programs.

- Comp has a formal semantics, defined using the reduction and labeled transition system styles.
- Components in Comp have an input interface, but no output interfaces; as a consequence, components can be used much in the same way as objects; however, component additionally have mobility capacities.
- Comp provides opening and closing operations for dynamically changing the visibility of a component. Thus, while by default communication in Comp remains global to fit the communication capacities of objects, if needed, the boundary of a component a can be closed to restrict access to specific components internal to a, say b; as a consequence, a component external to a will not be able to directly access b any more.
- Mobility of running components is allowed by means of movement primitives rather than by capturing and communicating components. These primitives are inspired by the constructs for achieving mobility in the Ambient calculus [6]. In Comp a component may thus move in the tree hierarchy of a component system. Processes (including component definitions) may be communicated, but they cannot be grabbed when running.

The main challenge in the formalization of a component model is to isolate key aspects of component-based systems and reflect them into specific constructs. In our case, the scenarios we want to describe with our model are presented as a set of examples that show how objects, components and runtime modifications of a program architecture can easily be combined. While developing the model we followed a general strategy that we call "*the architect principle*", which means that each component in the system is in charge of managing its children. We chose to follow this principle in our model because it improves its consistency, making it more easy to work with. Indeed, because of this principle, the one responsible for a modification is uniquely identified, which makes the behavior of systems clearer, and the modifications easier to code. Moreover, as the parent has the control over its children and their communications, it can help their integration into the rest of the system. We summarize below the main concepts that our approach is based on.

Components. Components are a way to structure a system into a set of units, each of them having a clear boundary. Processes inside the same boundary share some features, which can be of various kinds. For instance, they may share some computational resources, they may be in the same physical location or just inside the same security perimeter, or they may jointly implement some functionalities. In our case each component has its own data and its own execution space. More important, components represent the units of evolution in this paper. Put differently, evolution is obtained by adding, removing, replacing, wrapping or manipulating components and the component structure in a modular way. Notably, while a component is being, e.g., replaced, other components can continue to execute normally, thus minimizing service disruption.

Components give rise to a hierarchical structure, which allows for the modular definition of complex software architectures. Nesting of components may have different meanings, corresponding to the meanings of components themselves. In our case the parent of a component is in charge of updating its children, according to the "architect principle" described above.

Components can also be used to specify system deployment. Components in fact may represent physical locations and available resources, and deployment can be done by specifying how to associate to each object its enclosing component, thus defining how to deploy it. Notably, evolution constructs discussed later enable dynamic re-deployment of components to deal with modifications of the underlying architecture. We will not consider this possibility in depth.

Methods. Each component is equipped with a set of methods, defining its available functionalities. Having an explicit *input interface* is important, since this provides an abstract description of the functionalities of the component, to be used to ensure correctness of evolution steps. For instance, if an evolution step preserves the interface of the involved components, it will introduce no typing errors on runtime invocations.

Having methods in a component also matches the intuition that components correspond to objects with extended capabilities, which is useful for integration of components with objects. In contrast to other component models, COMP components have no output interface, since this has no correspondence inside objects. This would also make the semantics of the isolation mechanism more complex.

Isolation. A consequence of the component hierarchy is that a component may decide to hide its internal components, or to make (some of) them available to the external world. Hiding is fundamental for encapsulation: hidden components cannot be reached directly from the outside. Isolation can also be used to encode wrapping, where the wrapper component hides the wrapped one, while providing updated functionalities. In this way, for instance, methods can be removed, added or redefined. An internal component may however be left visible, which is useful for modeling shared resources.

According to the "architect principle", the decision about when and whether making a component available to the outside is taken by its parent.

Mobility. Having a hierarchical structure in place, the immediate way for achieving evolution is to allow components to move along the hierarchy. Remember that mobility can be either logical or physical, according to whether components model the software architecture of the system or its physical distribution.

Clearly, different primitives for mobility are possible. We decided to introduce two primitives for mobility, **in** and **out**, inspired from the Ambient calculus [6], that allow us to move a component inside a sibling one or outside its parent. These primitives disallow direct mobility between locations far from each other (unrealistic in many cases, e.g., for physical locations). Again, following the architect principle, a component may only be moved by its parent.

Channel-based communication. Our components communicate using channels. Analogously to passing object references, this kind of communication is completely independent from the component hierarchy. The main goal of this mechanism is to enable the support of futures (as available in ABS) for returning the result of a method to the caller regardless of which reconfigurations occurred since the call has been performed. Also, channel-based communication can be handy to synchronize different components performing some joint reconfiguration. We allow both names and processes to be communicated. Process communication in particular allows for code injection. Code injection and mobility are quite different concepts, since mobility refers to running components while code injection concerns idle processes.

Structure of the paper. We first present the calculus and explain its syntax in Sect. 2. We then formulate its operational semantics in Sect. 3, which is introduced by both: i) a reduction semantics that models the manipulation of the component's structure; and ii) a labeled semantics that models method invocations and channel-based communication. We illustrate the calculus with a number of examples in Sect. 4 showing, in particular, how various patterns of evolution of components are captured. Finally, we discuss related work in Sect. 5 and conclude in Sect. 6.

2 Primitives for Components and Evolution

This section presents the primitives we propose for modeling components and their evolution patterns. The formal semantics will be discussed in the next section. The syntax of Comp primitives is summarized in Fig. 1. Our syntax is based on a few syntactic categories (pairwise distinct): names for components, ranged over by a, b, l, x, names for channels, ranged over by c, d, y, ack, ch, names for methods, ranged over by $\mathtt{m}$, and process variables, ranged over by X. We denote with S isolation sets, namely sets of component names that can be of two kinds: finite sets, represented as $\{a_1, \ldots, a_n\}$ ($\{a\}$ will be shortened into a), or sets including all the component names but a finite set, represented as $\overline{\{a_1, \ldots, a_n\}}$.

Our values include component names, channel names and processes, and we use V to range over values. We use n to range over both component and channel names. We use J to range over process variables and component and channel names (assuming in this last case that they are bound by an input or a method definition).

The main construct of Comp is the component $a(S)\{M\}[P]$, where i) a is the name of the component; ii) S is the isolation set containing the names of inner components that are hidden from the environment; iii) M is the set of methods of the component; and iv) P is the body of the component, containing its currently running code and its sub-components.

Methods are defined as usual: $\mathtt{m}(J).P$ declares a method $\mathtt{m}$ with formal parameter J[2] and body P. As already said, J can be a component name, a channel name, or a process variable.

[2] We consider just one parameter, the extension to many parameters is trivial.

Component Names	a, b, x			
Channel Names	c, d, y, ack, ch			
Process Variables	X			
Names	n	$::=$	$a \mid c$	
Values	V	$::=$	$P \mid n$	
Placeholders	J	$::=$	$X \mid n$	
Isolation Sets	S	$::=$	$\{a_1, \ldots, a_n\} \mid \overline{\{a_1, \ldots, a_n\}}$	
Processes	P	$::=$	$a(S)\{M\}[P]$	Component
		$\mid$	$P \mid P$	Parallel Composition
		$\mid$	$A.P$	Action Prefix
		$\mid$	0	Null Process
		$\mid$	X	Process Variable Occurrence
		$\mid$	$\nu n\, P$	Restriction
Actions	A	$::=$	$a\_\mathtt{m}\langle V\rangle$	Method Call
		$\mid$	**open** S	Open
		$\mid$	**close** S	Close
		$\mid$	a **in** b	Move In
		$\mid$	a **out** b	Move Out
		$\mid$	$c(J)$	Message Receive
		$\mid$	$c\langle V\rangle$	Message Send
Methods Sets	M	$::=$	0	Empty Methods Set
		$\mid$	$\mathtt{m}(J).P$	Method Definition
		$\mid$	$M \mid M$	Methods Set

Fig. 1. COMP syntax

Processes can perform actions. The main actions are: i) method invocation $a\_\mathtt{m}\langle V\rangle$, that calls the method m of the component a with parameter V (here V is either a process P, for code injection, a component name b, or a channel name c); ii) **close** S, that hides the child components in the isolation set S from the rest of the system; in particular, **close** $\overline{\emptyset}$ will make the component a perfect black-box w.r.t. method invocations; iii) **open** S, that, on the opposite, reveals the components in S to the environment; iv) a **in** b, that puts the component a inside the parallel component b and v) a **out** b, that takes the component a that is inside b, and puts it outside as shown in Fig. 2. The command $a(S)\{M\}[P]$ creates a new component named a.

Finally, COMP contains several other constructs, standard from process calculi such as π-calculus [20] and Higher-Order π-calculus [26]: i) 0 is the process with no behavior (like **skip** in imperative languages); ii) X is a process variable; iii) $\nu n\, P$ is (channel or component) name restriction, which creates a new name n; and iv) $c(J)$, $c\langle V\rangle$ are input and output primitives for communication on channels.

As mentioned above, we use channel-based communication, well studied in process calculi, mainly to model the **return** statement of object-oriented languages. Remember that ABS has asynchronous method invocations and the

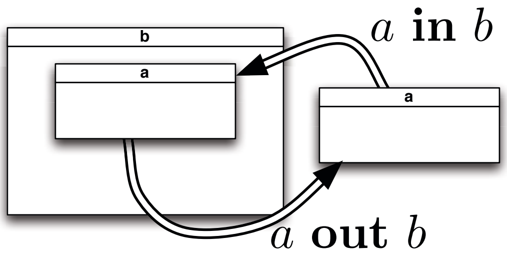

Fig. 2. In and Out primitives

communication of the return value is done using futures [9]. As one can see in the evolution patterns presented in Sect. 4, most of the occurrences of channel-based communication are already used for returning values.

Comp has no explicit operator for recursion or replication. However infinite behaviors can be encoded using higher-order communication or self-invocations of methods.

Bound names and variables (binders are name restriction, method definitions and input on channels) can be α-converted as usual. We write $n(P)$ for the set of names in process P and $fn(P)$ for its set of free names. We restrict our attention to processes with no free variables. We remark that names in isolation set $\overline{\{a_1, \ldots, a_n\}}$ are only $a_1, \ldots, a_n$.

3 Operational Semantics

The operational behavior of a process calculus is usually defined either by means of a reduction semantics, or by means of a labeled transition semantics. A reduction semantics typically uses the auxiliary relation of structural congruence, with which the participants of an interaction are brought into contiguous positions. This makes it possible to express interaction by means of simple term-rewriting rules. In a labeled transition semantics, in contrast, interacting parties can be far away, and the labels of the transition carry the information on the interaction up in the term, allowing synchronization.

For our calculus, we express component reconfigurations (those derived from mobility of components and modification of the isolation sets) by means of a reduction semantics, whereas channel-based interactions and method calls are described by means of a labeled transition system (LTS). The reason for the separation is that component reconfiguration is a local activity, whereas channel/method interaction is global (i.e., components far away can interact). The semantics of reconfiguration is simpler via a reduction semantics in the same way as in the Ambient calculus [6], which has inspired our movement primitives, and has a simple reduction semantics but a complex LTS semantics [19].

We write $P \rightarrow P'$ for an execution step of the process P that is derived using the reduction semantics, and $P \xrightarrow{\mu} P'$ for a step derived using the labeled semantics in which the label is μ. Finally, $\triangleright$ is the union of the two relations $\rightarrow$ and $\xrightarrow{\mu}$, and $\triangleright^*$

is the reflexive and transitive closure of $\triangleright$. The relations $\rightarrow$ and $\xrightarrow{\mu}$ are presented in the following two sections.

3.1 Semantics of Reconfiguration

We discuss here the formal semantics of reconfiguration in Comp. As already said, this consists in defining a structural congruence relation and a reduction relation $\rightarrow$.

Structural congruence. The structural congruence relation is written $\equiv$, and is defined as the smallest congruence that satisfies the rules presented in Fig. 3. The structural congruence $\equiv$ is quite standard: the parallel operator is commutative and associative and has unit 0 for processes and methods, while name restriction can be extruded if not capturing free names.

$$P \mid 0 \equiv P \qquad P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$$

$$M \mid 0 \equiv M \qquad M_1 \mid M_2 \equiv M_2 \mid M_1 \qquad (M_1 \mid M_2) \mid M_3 \equiv M_1 \mid (M_2 \mid M_3)$$

$$Q \mid \nu n\, P \equiv \nu n\,(Q \mid P) \quad \text{if } n \notin \mathit{fn}(Q)$$

$$a(S)\{M\}[\nu n\, P] \equiv \nu n\,(a(S)\{M\}[P]) \quad \text{if } n \notin \mathit{fn}(S) \cup \{a\} \cup \mathit{fn}(M)$$

Fig. 3. Structural congruence on Comp processes

Reduction rules. The relation $\rightarrow$ is defined as the smallest relation closed w.r.t. $\equiv$ and α-conversion that validates the rules presented in Fig. 4. The first two rules describe the manipulation of the program architecture. First, the command a **in** b takes two parallel components named a and b respectively and moves a

$$a\ \mathbf{in}\ b.P \mid a(S_a)\{M_a\}[P_a] \mid b(S_b)\{M_b\}[P_b] \rightarrow P \mid b(S_b)\{M_b\}[P_b \mid a(S_a)\{M_a\}[P_a]]$$

$$a\ \mathbf{out}\ b.P \mid b(S_b)\{M_b\}[P_b \mid a(S_a)\{M_a\}[P_a]] \rightarrow P \mid a(S_a)\{M_a\}[P_a] \mid b(S_b)\{M_b\}[P_b]$$

$$a(S)\{M\}[\mathbf{close}\ S'.P \mid P'] \rightarrow a(S \cup S')\{M\}[P \mid P']$$

$$a(S)\{M\}[\mathbf{open}\ S'.P \mid P'] \rightarrow a(S \setminus S')\{M\}[P \mid P']$$

$$\frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad \frac{P \rightarrow P'}{\nu n\, P \rightarrow \nu n\, P'} \qquad \frac{P \rightarrow P'}{a(S)\{M\}[P] \rightarrow a(S)\{M\}[P']}$$

Fig. 4. Rules for reconfiguration

inside b. The operator a **out** b has the opposite behavior, as it takes a component named b in parallel (i.e. in the same parent component), takes a component a inside b and puts a outside of it (i.e. in parallel to b). The two following rules allow to change component visibility. The action **close** S executed inside a component $a(S')\{M\}[P]$ will add S to S' while **open** S will remove S from S'. Since isolation sets are closed under set union and set difference the semantics of the two primitives is well-defined.

The last three rules are standard closures under parallel composition, name restriction and component contexts.

How the isolation set S influences the behavior of method calls is described in the following section.

3.2 Method Invocations and Channel Communications

Method invocations and channel communications are described using an LTS. The rules for channel communication are entirely standard, following the SOS style of message-passing calculi such as π-calculus and Higher-Order π-calculus, as channel-based communications are independent from the component hierarchy. Method calls are described by similar rules, which just have one more side condition that checks whether the method call is allowed to go through the component boundaries it has to cross.

The label (or action) μ of a transition can be of five different kinds: i) τ, corresponding to an inner step; ii) $\overline{c}(V)$, corresponding to the sending of the value V on the channel c; iii) $c(V)$, corresponding to the reception of the value V at the channel c; iv) $\overline{a\_\mathtt{m}}(V)$, corresponding to the invocation of the method $\mathtt{m}$ of the component a with the parameter V; and v) $a\_\mathtt{m}(V)$, corresponding to the triggering of method $\mathtt{m}$ of component a with the parameter V. To simplify our rules, we note Φ a label corresponding to a communication on a channel or the inner step τ, and $\Psi(a)$ a label corresponding to a method call to a component named a.

The relation $\xrightarrow{\mu}$ is defined as being the smallest closed relation w.r.t. $\equiv$ and α-conversion that validates the rules in Fig. 5. The first two rules are the basic cases of channel communication, while the two following ones deal with the basic cases of method invocation. We remember that values V can be component or channel names or processes. Similarly, J is used for denoting either a (bound) component/channel name or a process variable. The five next rules handle congruence: i) first, we state that the relation is closed w.r.t. parallel composition; ii) then, communication on channels can freely cross component boundaries; while iii) method calls to a component b cannot cross a component boundary with b in its isolation set S; also iv) an inner step is always propagated by components and v) can cross restrictions. We do not need to add rules for extrusions or propagating actions through restrictions, since restrictions can always be moved to the top level. These rules would be needed however, e.g., for defining a bisimilarity-based abstract semantics. The last rule shows how two processes synchronize to achieve either a channel communication or a method call (in labels, overline is only put on subjects for notational convenience), while the last one allows components to invoke their own methods.

$$c\langle V\rangle.P \xrightarrow{\overline{c}(V)} P \qquad c(J).P \xrightarrow{c(V)} P\{^V/_J\} \qquad a\_\mathtt{m}\langle V\rangle.P' \xrightarrow{\overline{a\_\mathtt{m}}(V)} P'$$

$$a(S)\{\mathtt{m}(J).P \mid M\}[P'] \xrightarrow{a\_\mathtt{m}(V)} a(S)\{\mathtt{m}(J).P \mid M\}[P' \mid P\{^V/_J\}] \qquad \frac{P_1 \xrightarrow{\mu} P_2}{P_1 \mid P \xrightarrow{\mu} P_2 \mid P}$$

$$\frac{P_1 \xrightarrow{\Phi} P_2}{a(S)\{M\}[P_1] \xrightarrow{\Phi} a(S)\{M\}[P_2]} \qquad \frac{P_1 \xrightarrow{\Psi(b)} P_2 \qquad b \notin S}{a(S)\{M\}[P_1] \xrightarrow{\Psi(b)} a(S)\{M\}[P_2]}$$

$$\frac{P_1 \xrightarrow{\tau} P_2}{\nu n\, P_1 \xrightarrow{\tau} \nu n\, P_2} \qquad \frac{P_1 \xrightarrow{\mu} P_2 \qquad P_1' \xrightarrow{\overline{\mu}} P_2'}{P_1 \mid P_1' \xrightarrow{\tau} P_2 \mid P_2'}$$

$$\frac{P_1 \xrightarrow{\overline{a\_\mathtt{m}}(V)} P_2}{a(S)\{\mathtt{m}(J).P \mid M\}[P_1] \xrightarrow{\tau} a(S)\{\mathtt{m}(J).P \mid M\}[P_2 \mid P\{^V/_J\}]}$$

Fig. 5. Rules for communication

4 Basic Reconfiguration Patterns

In this section we discuss different basic patterns of reconfiguration, showing how they can be encoded using Comp primitives.

Adding and removing a component. The two most basic operations to manipulate a program structure, in addition to the **in** and **out** operators, are the addition and removal of components. The addition of a component is straightforward, as it corresponds to the creation of a new component. The macro **Add**(a, S, M, P) creates a component with name a, isolation set S, set of methods M and body P:

$$\textbf{Add}(a, S, M, P) \triangleq a(S)\{M\}[P]$$

The encoding of component removal is more subtle. Instead of destroying the component, we simply hide it with the insurance that it will never be accessible again. Macro **Remove**(a, P) removes component a and then executes continuation process P, that is supposed to notify the environment that the removal has been performed, i.e. a is not available anymore.

$$\textbf{Remove}(a, P) \triangleq \nu b(b(\overline{\emptyset})\{0\}[0] \mid a \textbf{ in } b.P)$$

In order to remove a component, the **Remove**(a, P) process needs to be put in parallel with the component to be removed:

$$a(S)\{M\}[Q] \mid \textbf{Remove}(a, P)$$

Assuming that the action a **in** b inside **Remove**(a, P) is executed, the configuration becomes

$$\nu b(b(\overline{\emptyset})\{0\}[a(S)\{M\}[Q]] \mid P)$$

The macro creates a new black-box component b that no one knows (and thus no one can modify), then moves the component a inside it and executes continuation P. Note that b completely hides a and its subcomponents from the rest of the architecture: a will never be accessible again. Hence, as soon as the computation inside a finishes, it becomes behaviorally equivalent to 0 and can be safely garbage collected.

Our definition of component removal intentionally does not destroy the component. Indeed, suppose we want to remove the component a so to replace it with a newer version. The current version of a could be computing some result of past method calls. Immediately destroying the component a would create some consistency issues inside the system, as some other computations may be waiting for the results being computed by a. Our definition ensures that we can replace a with another component without creating such inconsistencies.

Hiding structure manipulation. It is common practice to isolate the part of the system one wants to modify prior to its manipulation to prevent possible interferences. In our case, interferences may emerge because two components can have the same name: suppose we want to create a component b and move an existing component a inside of it. The natural approach is to first create the component b and then move a, but, if another component with name b already existed, we will have two components with the same name b but possibly different behavior/structure. Because of the nondeterminism, we cannot ensure that component a is moved in the newly created component. It may as well happen that it is moved inside the old one. A simple way to avoid this is to create a temporary component b' where we put only the components that are expected to take part in the reconfiguration, i.e. a and the new b. After the reconfiguration process terminates, we can put the resulting subsystem back in place.

We propose here a simple definition of this isolation mechanism. The macro **Hide**(a, b, P, ack, P') hides the component a, applies to it the reconfiguration specified by P which notifies its end by sending an output on ack. Upon receipt of ack the new component b is put in the environment. Finally, process P' is executed to make the environment aware of the end of the reconfiguration.

$$\mathbf{Hide}(a, b, P, ack, P') \triangleq \nu b' \, (b'(\overline{\emptyset})\{0\}[P] \mid a \ \mathbf{in} \ b'.ack(x).b \ \mathbf{out} \ b'.P')$$

An example of usage of the pattern will be discussed in the next paragraph about component renaming. In our macro definition, channel ack is used to notify when the reconfiguration specified by P is terminated and the resulting component b can be released to the environment. In order to avoid interferences this name should not be used elsewhere, e.g. it may be freshly generated just before the macro (a bit more care is needed if P is received via a communication).

Renaming a component. Since components are identified by their names, renaming a component is a useful operation. Also, two components may have the same name. This raises the possibility of nondeterminism in method invocations or reconfiguration operations – which might or might not be desired depending on the situation. Renaming a component can thus be used to create or remove nondeterminism.

We show below how to define macro **Rename**(a, b, P), which renames component a into b (we assume a and b to be distinct) and executes P to notify to the environment about the end of the renaming operation. The idea is to create a new component with name b, with isolation set $\{a\}$ and with the same methods as a (we assume to know the interface $\mathtt{m_1}, \ldots, \mathtt{m_n}$ of the component to be renamed, i.e. the macro depends on the interface of the component to be renamed). The behavior of these methods will be just to forward the calls to a. By moving a into b (thus making a no longer accessible from the environment), b behaves as a. This concludes the renaming. Remark that the renaming definition is based on the isolation macro **Hide** defined above to avoid interferences.

$$\begin{aligned}&\mathbf{Rename}(a, b, P)\\ &\quad \triangleq \nu ack\, \mathbf{Hide}\left(a, b, \left(b(a) \left\{ \begin{array}{c} \mathtt{m_1}(x).a\_\mathtt{m_1}\langle x\rangle \\ \cdots \\ \mathtt{m_n}(x).a\_\mathtt{m_n}\langle x\rangle \end{array} \right\} [0] \mid a \ \mathbf{in}\ b.ack\langle 0\rangle.0 \right), ack, P\right)\end{aligned}$$

We show now how the renaming of a component works in practice. To simplify the presentation, we write M for the methods of the component a, and M' for the methods of b, defined as in the definition of the **Rename** operator.

$$\begin{aligned}
&a(S)\{M\}[P] \mid \mathbf{Rename}(a, b, P')\\
&\quad = a(S)\{M\}[P] \mid \nu ack\, \mathbf{Hide}(a, b, (b(a)\{M'\}[0] \mid a\ \mathbf{in}\ b.ack\langle 0\rangle.0), ack, P)\\
&\quad = a(S)\{M\}[P] \mid \nu ack, b'\ (b'(\overline{\emptyset})\{0\}[b(a)\{M'\}[0] \mid a\ \mathbf{in}\ b.ack\langle 0\rangle.0] \mid\\
&\qquad\qquad a\ \mathbf{in}\ b'.ack(x).b\ \mathbf{out}\ b'.P')\\
&\quad \equiv \nu ack, b'\ (a(S)\{M\}[P] \mid b'(\overline{\emptyset})\{0\}[b(a)\{M'\}[0] \mid\\
&\qquad\qquad a\ \mathbf{in}\ b.ack\langle 0\rangle.0] \mid a\ \mathbf{in}\ b'.ack(x).b\ \mathbf{out}\ b'.P')\\
&\quad \to \nu ack, b'\ (b'(\overline{\emptyset})\{0\}[a(S)\{M\}[P] \mid b(a)\{M'\}[0] \mid\\
&\qquad\qquad a\ \mathbf{in}\ b.ack\langle 0\rangle.0] \mid ack(x).b\ \mathbf{out}\ b'.P')\\
&\quad \to \nu ack, b'\ (b'(\overline{\emptyset})\{0\}[b(a)\{M'\}[a(S)\{M\}[P]] \mid ack\langle 0\rangle.0] \mid ack(x).b\ \mathbf{out}\ b'.P')\\
&\quad \xrightarrow{\tau} \nu ack, b'\ (b'(\overline{\emptyset})\{0\}[b(a)\{M'\}[a(S)\{M\}[P]]] \mid b\ \mathbf{out}\ b'.P')\\
&\quad \to \nu ack, b'\ (b'(\overline{\emptyset})\{0\}[0] \mid b(a)\{M'\}[a(S)\{M\}[P]] \mid P')
\end{aligned}$$

Wrapping. Wrapping is a key feature in evolvable architectures. The idea of wrapping is to replace an old component named, for instance, a with a new component b that exploits a to perform its behavior. One can also imagine that b changes/extends the behavior of a.

A simple definition of wrapping can be given inspired by the **Rename** macro defined above. The main difference is that now the information about the behavior of component b is needed as a parameter. We call this definition **NaifWrap**. We will present a refined definition just after. Macro $\mathbf{NaifWrap}(a, b(S)\{M\}[P], P')$ takes a component a and puts it inside the wrapper b, executing then process P' to make the environment aware of the completion of the operation. As for the renaming operation, we assume that names a and b are different.

$$\begin{aligned}&\mathbf{NaifWrap}(a, b(S)\{M\}[P], P')\\ &\qquad \triangleq \nu ack\, \mathbf{Hide}(a, b, (b(S \cup \{a\})\{M\}[P] \mid a \textbf{ in } b.ack\langle 0\rangle.0), ack, P')\end{aligned}$$

One may also want to wrap a component a using a wrapper with the same name. In this way in fact an external component will not notice that wrapping has been performed, and can interact with the wrapper as if it was the old component. This is particularly useful when one uses wrapping to extend the interface or the behavior of an existing component.

The naif definition of wrapping above does not work when the wrapper and the wrapped component have the same name. In this case the code of the wrapper must be able to call the old component a, but also to perform self-calls. On the contrary, neither the environment nor the old component a should notice that the wrapping has been performed. In particular, calls to a in the environment should activate the wrapper.

For these reasons we propose below a refined definition of wrapping. The new definition uses both renaming and naif wrap. Macro $\mathbf{Wrap}(b, a(S)\{M\}[P], P')$ takes component a and wraps it into a wrapper with the same name, which is supposed to use b (different from a) to reference the old component.

$$\begin{aligned}&\mathbf{Wrap}(b, a(S)\{M\}[P], P')\\ &\triangleq \nu ch, ack\,(\mathbf{Hide}(a, a, \begin{pmatrix}\mathbf{Rename}(a, b, ch\langle 0\rangle.0) \mid \\ ch(x).\mathbf{NaifWrap}(b, a(S)\{M\}[P], ack\langle 0\rangle.0)\end{pmatrix}, ack, P'))\end{aligned}$$

When the macro is put in parallel with a component a, a is renamed into b and then wrapped. In the definition, channel ch is used to ensure that renaming of a has been completed before performing the actual wrapping.

Update. Updating a component means adding to it some new features. This can be done by adding new methods, new processes or new sub-components.

In Comp, the interface of components is fixed. However adding new methods can be simulated. In fact, it is a particular case of the wrap operation defined above.

Adding new processes and sub-components can also be done, using higher-order communication. This can be programmed in such a way that the updated component can also perform some check to ensure the validity of the update before installing it. In particular, the component to be updated should provide a dedicated method for update. The macro $\mathbf{Update}(a, P)$ requiring component a to install the update P is defined as:

$$\mathbf{Update}(a, P) \triangleq a\_\mathtt{upd}\langle P\rangle$$

and assumes that component a has the form:

$$a(S)\{\mathtt{upd}(X).X \mid M\}[P']$$

The update mechanism could be used to apply one of the previously discussed reconfiguration patterns to subcomponents. In fact, the above patterns can be applied only to parallel components. In case one wants to apply them also to nested components, the update mechanism could be used to forward downward the reconfiguration request. For instance, assume that one process in parallel with a component a wants to rename a component b contained in a to name c. This could be accomplished by using the update mechanism to send to the component a the rename macro:

$$\mathbf{Update}(a, \mathbf{Rename}(b, c, 0)) \;\mid\; a(S)\{\mathtt{upd}(X).X \mid M\}[b(S')\{M'\}[P'] \mid P]$$

A more refined macro $\mathbf{Update}(a, c, P)$ may include also a channel name c where to ask for the credentials of the update. In such a way component a may insert some code for checking the validity of the update in between the receipt of the update request and the execution of X.

Links. In many component models in the literature [2,4,23,31], components come equipped with input ports, output ports, and links connecting input ports to output ports. In our model methods can be considered as interfaces of an input port. However, to keep the calculus focused and manageable, we choose not to include explicit output ports and links as language constructs. Here, we show that output ports and links can be defined by the calculus, thus enabling the programmer to exploit them. In particular, different disciplines of linking ports can be defined and analyzed using the calculus.

The output ports of a component a represent the dependencies that a needs in order to perform its task. Output ports can be seen in our model as virtual methods. Connecting one such method $\mathtt{m_v}$ to a real method $\mathtt{m_r}$ (typically, from another component) using a link means that the dependency $\mathtt{m_v}$ is satisfied by invoking method $\mathtt{m_r}$. In other words, a link acts as a forwarder. We encode a link between the output port $a.\mathtt{m_v}$ and the input port $a'.\mathtt{m_r}$ as a component (named here b) of the form:

$$b(\emptyset)\{0\}[a(\emptyset)\{\mathtt{m_v}(x).a'\_\mathtt{m_r}\langle x\rangle\}[0]]$$

Each link is a component with its own name, so that it can be referred to, e.g. remove it when it is no more needed.

We can finally present the macros for creating and deleting links. Macro $\mathbf{Connect}(a.\mathtt{m_v}, a'.\mathtt{m_r}, b, P)$ creates a link named b connecting port $a.\mathtt{m_v}$ to method $a'.\mathtt{m_r}$ and executing P upon completion. Macro $\mathbf{DisConnect}(b, P)$ removes link b and executes P upon completion.

$$\mathbf{Connect}(a.\mathtt{m_v}, a'.\mathtt{m_r}, b, P) \triangleq b(\emptyset)\{0\}[a(\emptyset)\{\mathtt{m_v}(J).a'\_\mathtt{m_r}\langle J\rangle\}[0]] \mid P$$

$$\mathbf{DisConnect}(b, P) \triangleq \mathbf{Remove}(b, P)$$

The links defined here are quite primitive, in the sense that they are subject to interference. However interferences could be avoided. First one would need to use fresh names to link components. Also, links are attached here to component names. On one hand this ensures that links are not spoiled by wrapping. On the other hand, they are spoiled when the target component is renamed. If this is not the desired behavior, one has to update the links when a component is renamed. This may require keeping track of all the links referring to a given component. We do not consider this issue in more detail here.

Distribution. In COMP, there are no dedicated constructs for modeling physical distribution. However, as we already said, components can also represent physical locations. We can consider for instance a top-level component representing the network, and containing a sub-component for each site. The network component can e.g. be used to implement communication protocols. It can even provide facilities for deployment and re-deployment of components.

A sample configuration is defined below. It includes two sites, $\texttt{Site}_1$ and $\texttt{Site}_2$, running processes P_1 and P_2 respectively. The two sites cannot communicate through method calls. In fact, each of them can just see the network. The network however provides an asynchronous communication mechanism, made available as a global component `Net` relaying messages from one site to another. We assume to this end that $\texttt{Site}_1$ (resp. $\texttt{Site}_2$) listens for incoming messages via method `in`, and can be reached by calling the method $\texttt{Net\_toS}_1\langle x\rangle$ (resp. $\texttt{Net\_toS}_2\langle x\rangle$) provided by the `Net` component.

$$\begin{array}{l}
\texttt{Network}(\emptyset)\{0\}[\\
\quad \texttt{Site}_1(\overline{\{\texttt{Net}\}})\{\texttt{in}(x).P_1'\}[P_1]\\
\quad \mid \texttt{Site}_2(\overline{\{\texttt{Net}\}})\{\texttt{in}(x).P_2'\}[P_2]\\
\quad \mid \texttt{Net}(\emptyset)\{\texttt{toS}_1(x).\texttt{Site}_1\texttt{\_in}\langle x\rangle \mid \texttt{toS}_2(x).\texttt{Site}_2\texttt{\_in}\langle x\rangle\}[0]\\
]
\end{array}$$

Note that in this example we use the set of names $\overline{\{\texttt{Net}\}}$ to specify that both sites $\texttt{Site}_1$ and $\texttt{Site}_2$ can only communicate with the component `Net` which encodes the network protocols. Also, the network protocols defined here are trivial (pure forwarding), but they can easily be extended to account correctness checking, buffering or other features.

Re-deployment. Since distribution is defined in terms of our component notion, and the component hierarchy can be dynamically updated at any moment, we can exploit our primitives for dynamic re-deployment.

Suppose we have a running system as above, and that we want to move a component from one location to another, e.g. for efficiency reasons. This can be obtained by adding to the component `Network` above some code for performing re-deployment. A basic example is given below, where we add a method `move` to the `Network`:

$$\texttt{Network}(\emptyset)\{\texttt{move}(a, l_1, l_2).a\ \textbf{out}\ l_1.a\ \textbf{in}\ l_2.0\}[\ldots]$$

Method `move` takes three parameters, the name a of the component to be moved, its current location l_1 and its destination l_2. Simple forms of re-deployment can be obtained by invoking method `Network_move` with suitable parameters. Clearly, more refined implementations of primitives for re-deployment are possible, keeping into account more complex reconfiguration requests or performing suitable checks on the request and on the state of the system before actually executing the request.

5 Related Work

Components have been introduced as a new programming paradigm in the mid-nineties, as a mean to solve several limitations of the object models [31]. One of the main such limitations is the lack of high level operators for adaptation/evolution, as motivated in [24,25]. Nowadays, many component models exist, distinguished by their definition of the component structure and by the operations provided on them. For instance, the OSGi component model [2] developed by IBM defines its components as a set of objects and classes with some extra information, as which services the component provides, and on which services it depends on. This model thus allows one to add components at runtime, with a constraint solver that checks and solves the dependencies of the added component. Let us note that components in OSGi can only be assembled in a flat structure, while in Fractal [4], developed by INRIA and France-Telecom, they can be assembled into a tree structure. But in contrast to OSGi, in Fractal, the programmer must explicitly specify how the dependencies are solved by using *bindings*, similar to the links presented in our examples.

The main focus of these component models, as well as for many others, like COM [8], Java Beans [30], Appia [21], Click [23], Coyote [3] or Dream [14], is practical and implementation support for code packaging and wiring of components (for instance, OSGi is the basis of the Eclipse IDE and its plugin mechanism). Some of the component models allow evolution steps for the components, but do not provide formally defined operators to reason about these steps. Indeed, to meet practical requirements, the models are tied to large programming languages and come together with complex APIs. This makes them inappropriate for the formal investigation of dynamic evolution steps in the context of ABS models. Moreover, even if many of the models are based on an object-oriented language, none of them are concerned with the dynamic behavior or describe the interaction between components and objects. For instance, the Fractal model states that communication between components can only occur by using the input and output ports. But because of its implementation in Java, it is possible to use the objects and their methods to make components communicate without an explicit use of ports. Our approach aims to solve these two problems: as our model is defined in the process calculus style it has a precise semantics, and we designed our components to be an extension of ABS object groups, with additional isolation and mobility capabilities: hence the interactions between components and objects are naturally described in our model.

A few component models are formally defined, most of them in the process calculus style. There are, for instance, the M-calculus [28] and the kell-calculus [29], both inspired by Fractal. Other calculi, like the different flavors of the Ambient calculus [6,32,16,5], the join-calculus [11] or the seal calculus [7] use named boxes as a means to structure the program into a tree hierarchy. Moreover, some of these calculi use this structure to control communication and to enable adaptation through the modification of the program structure. Finally, in Oz/K [17], the authors have proposed a core programming language with components as a main feature. These component models, being formally defined, are a better fit for ABS than the previous ones, but still have several limitations. First, only Oz/K integrates objects in its model. The other proposals do not provide any description of how components and objects interact (actually, many of them have no concept of object at all) and so cannot be directly integrated with ABS. Moreover, Oz/K has quite a complex communication pattern, and deals with adaptation via the use of *passivation*, which, as suggested by [15], is a too high level operation to hope for any tool to help proving behavioral properties. Our model is by design simple and close to objects so to solve by construction the limitations of existing formal component models described above.

6 Conclusion

In this paper we have presented Comp, a component model designed to be easily integrated into an object-oriented language like ABS, and defined with a formal semantics in order to be a basis for proofs of correctness and behavioral properties. We also put our model at work on a set of simple patterns for dynamic reconfiguration, showing e.g. that component removal, update and wrapping can easily be defined using the primitives that we propose.

The work presented here opens many possibilities for further development. First, we want a full integration between Comp and ABS. We expect no major difficulties, since Comp has been developed with this aim in mind, and components are designed as extended objects, but a few technical steps are required. First, the semantics of ABS is defined in a purely rewriting style, thus we should adapt the Comp semantics. Then one has to add component definitions by instantiation of suitable component classes (extending object classes). Finally, communication patterns based on channels have to be reformulated in terms of asynchronous method calls and futures.

Another important but orthogonal task that we plan to investigate is a type system for Comp. The main point here is the trade-off between the strong guarantees that the type system should provide (e.g., in terms of absence of message-not-understood errors), and the possibility of typing evolution steps able to change the structure of components.

A last point to be investigated is the introduction of error handling and compensations in the model. This would allow one to manage errors (due, e.g., to inconsistent reconfigurations) ensuring that the whole system can reach a consistent state. Such an approach is under analysis in ABS, but we plan to extend it to deal with components. In particular, the hierarchy of components may be used as hierarchy for error management. Also, compensations may allow to locally manage the effect of a global reconfiguration.

References

1. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Science of Computer Programming (2010) (in press)
2. OSGi Alliance. Osgi Service Platform, Release 3. IOS Press, Inc., Amsterdam (2003)
3. Bhatti, N.T., Hiltunen, M.A., Schlichting, R.D., Chiu, W.: Coyote: A system for constructing fine-grain configurable communication services. ACM Trans. Comput. Syst. 16(4) (1998)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: The Fractal Component Model and its Support in Java. Software - Practice and Experience 36(11-12) (2006)
5. Bugliesi, M., Castagna, G., Crafa, S.: Access control for mobile agents: the calculus of boxed ambients. ACM. Trans. Prog. Languages and Systems 26(1) (2004)
6. Cardelli, L., Gordon, A.D.: Mobile Ambients. Theoretical Computer Science 240(1) (2000)
7. Castagna, G., Vitek, J., Nardelli, F.Z.: The Seal calculus. Inf. Comput. 201(1) (2005)
8. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J.: OpenCOM v2: A Component Model for Building Systsms Software. In: Proceedings of IASTED Software Engineering and Applications, SEA 2004 (2004)
9. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
10. Full ABS Modeling Framework, Deliverable 1.2 of project FP7-231 620 (HATS) (March 2011), `http://www.hats-project.eu`
11. Fournet, C., Gonthier, G.: The join calculus: A language for distributed mobile programming. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 268–332. Springer, Heidelberg (2002)
12. European Project HATS, `http://www.hats-project.eu`
13. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (2007)
14. Leclercq, M., Quema, V., Stefani, J.-B.: DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs. IEEE Distributed Systems Online 6(9) (2005)
15. Lenglet, S., Schmitt, A., Stefani, J.-B.: Howe's Method for Calculi with Passivation. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 448–462. Springer, Heidelberg (2009), doi:10.1007/978-3-642-04081-8_30
16. Levi, F., Sangiorgi, D.: Mobile safe ambients. ACM. Trans. Prog. Languages and Systems 25(1) (2003)
17. Lienhardt, M., Schmitt, A., Stefani, J.-B.: Oz/k: A kernel language for component-based open programming. In: GPCE 2007: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, pp. 43–52. ACM, New York (2007)

18. Liu, X., Kreitz, C., van Renesse, R., Hickey, J., Hayden, M., Birman, K., Constable, R.: Building Reliable, High-Performance Communication Systems from Components. In: Proceedings of the 1999 ACM Symposium on Operating Systems Principles, Kiawah Island, SC (December 1999)
19. Merro, M., Nardelli, F.Z.: Behavioral theory for mobile ambients. J. ACM 52(6), 961–1023 (2005)
20. Milner, R., Parrow, J., Walker, J.: A calculus of mobile processes, I and II. Inform. and Comput. 100(1), 1–40, 41–77 (1992)
21. Miranda, H., Pinto, A.S., Rodrigues, L.: Appia: A flexible protocol kernel supporting multiple coordinated channels. In: 21st International Conference on Distributed Computing Systems (ICDCS 2001). IEEE Computer Society, Los Alamitos (2001)
22. Montesi, F., Sangiorgi, D.: A model of evolvable components. In: Wirsing, M., Hofmann, M., Rauschmayer, A. (eds.) TGC 2010, LNCS, vol. 6084, pp. 153–171. Springer, Heidelberg (2010), doi:10.1007/978-3-642-15640-3_11
23. Morris, R., Kohler, E., Jannotti, J., Frans Kaashoek, M.: The Click Modular Router. In: ACM Symposium on Operating Systems Principles (1999)
24. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Proceedings of the 20th International Conference on Software Engineering, ICSE 1998, pp. 177–186. IEEE Computer Society, Washington, DC, USA (1998)
25. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime software adaptation: framework, approaches, and styles. In: Companion of the 30th International Conference on Software Engineering, ICSE Companion 2008, pp. 899–910. ACM, New York (2008)
26. Sangiorgi, D.: From pi-calculus to higher-order pi-calculus - and back. In: Gaudel, M.-C., Jouannaud, J.-P. (eds.) CAAP 1993, FASE 1993, and TAPSOFT 1993. LNCS, vol. 668, pp. 151–166. Springer, Heidelberg (1993)
27. Schäfer, J., Poetzsch-Heffter, A.: Jcobox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
28. Schmitt, A., Stefani, J.-B.: The M-calculus: A Higher-Order Distributed Process Calculus. In: Proceedings 30th Annual ACM Symposium on Principles of Programming Languages, POPL (2003)
29. Schmitt, A., Stefani, J.-B.: The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005)
30. Sun Microsystems. JSR 220: Enterprise JavaBeans, Version 3.0 – EJB Core Contracts and Requirements (2006)
31. Szyperski, C.: Component Software, 2nd edn. Addison-Wesley, Reading (2002)
32. Teller, D., Zimmer, P., Hirschkoff, D.: Using Ambients to Control Resources. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 288–303. Springer, Heidelberg (2002)