Practical Analysis Framework for Component Systems with Dynamic Reconfigurations

Olga Kouchnarenko[1,2] and Jean-François Weber[1(✉)]

[1] FEMTO-ST CNRS and Bourgogne Franche-Comté University, Besançon, France
[2] Inria/Nancy-Grand Est, Villers-lès-Nancy, France
{okouchnarenko,jfweber}@femto-st.fr

Abstract. Dynamic reconfigurations that modify the architecture of component-based systems without incurring any system downtime need to preserve the architectural consistency. In this context, we propose a reconfiguration model based on Hoare logic using sequences and (unlike most of the related work on reconfigurations) the alternative and the repetitive constructs.Using primitive reconfiguration operations as building blocks, this model takes advantage of the predicate-based semantics of programming language constructs and weakest preconditions to treat dynamic reconfigurations in a manner that preserves configuration consistency. Then, after enriching the model with interpreted configurations and reconfigurations in a consistency compatible manner, a conformance relation is exploited to validate component systems' implementations within the environment supporting the Fractal and FraSCAti frameworks. A practical contribution consists of promising experimental results obtained thanks to our implementations, notably on a cloud-based multi-tier hosting environment model managed as a component system.

1 Introduction

Dynamic reconfigurations that modify the architecture of self-adaptive [1] component-based systems without incurring any system downtime must happen not only in suitable circumstances, but also need to preserve the architectural consistency. Whereas the former can be ensured by adaptation policies [1,2], the latter is directly related to the definition of reconfigurations and to the reconfiguration ordering/protocol [3,4].

In [3], it is assumed that the reconfigurations always make the component assembly evolve from one consistent architecture to another consistent architecture, only through a path of architecturally consistent architectures. However, primitive reconfigurations like *unbind, stop,* etc. may disrupt such a path. With relation to consistency constraints defined in [5] over component-based architectures, their preservation of the system under scrutiny was uneasy to prove, mostly because of the lack of precise semantics for primitive reconfiguration operations. Therefore, when considering more complicated reconfigurations composed

This work has been partially funded by the Labex ACTION, ANR-11-LABX-0001-01.

M. Butler et al. (Eds.): ICFEM 2015, LNCS 9407, pp. 287–303, 2015.
DOI: 10.1007/978-3-319-25423-4_18

of sequences, repetitions, or choices over primitive reconfiguration operations, to address the above-mentioned issue, we propose to express reconfigurations' preconditions and postconditions using the concept of *weakest precondition* [6]. This precise and concise formalism allows us to express primitive and non primitive *guarded reconfigurations*; this is the first contribution simplifying both reconfiguration protocols and adaptation policies.

Then, after enriching the model with interpreted configurations and reconfigurations in a consistency-compatible way, a conformance relation is exploited to validate implementations of a component architectural model developed within our architecture manager supporting the Fractal [7] and FraSCAti [8] frameworks.

This second practical contribution allows us, not only, to simulate a desired run of a system being reconfigured, but, also, to generate all (or a subset of the) possible reconfiguration combinations useful, for example, for a (bounded) reachability analysis. The paper reports on promising experimental results obtained thanks to our implementations, notably on a cloud-based multi-tier application hosting environment model managed as a component software architecture.

The paper is organised as follows: Sect. 2 presents, as a case study, a cloud-based multi-tier application hosting environment managed as a component-based system. Background information on our component-based reconfiguration model, as well as elements of operational semantics are given in Sect. 3. In Sect. 4 a richer interpreted reconfiguration model is shown to be weakly simulated by the more abstract model; nevertheless, this simulation respects non-divergency. Using several case studies, Sect. 5 describes conformant implementations of the interpreted model within different environments. Section 6 presents related work and our conclusion.

2 Case Study

Internet service providers and telecommunications operators tend more and more to define themselves as cloud providers. In this context, automation of software and (virtual) hardware installation and configuration is paramount. It is not enough for an application to be cloud-ready; it has to be scalable and scalability mechanisms need to be integrated in the core of the cloud management system.

We consider a typical three-tier web application using a front-end web server, a middle-ware application server, and a back-end data providing service such as a database or a data store. Figure 1 shows a single virtual machine (or *VM*) hosting together the three services of such an application. The VM is represented as a composite component *virtualMachine* containing sub-components representing each service (*httpServer*, *appServer*, and *dataServer*) of the application. Each of the service sub-component has two provided interfaces: one to provide its service and another one used to monitor the service.

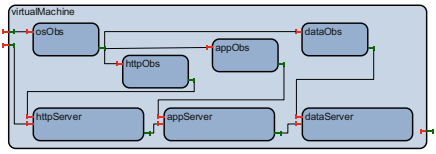

Fig. 1. Managed Virtual Machine with Three-tier Application Components

Furthermore, the VM of Fig. 1 also contains four *observers*, that are sub-components used to monitor services. The sub-component *osObs* is used to monitor the Operating System of the VM. It is also bound to the sub-components *httpObs*, *appObs*, and *dataObs* used respectively to monitor the services of the *httpServer*, *appServer*, and *dataServer* sub-components. Finally, the VM composite component itself has two provided interfaces: one used to provide services and a second one used for monitoring.

Of course, a VM does not have to be monitored, nor have to host the three types of services. Figure 2 illustrates a *cloud environment*, *clouEnv*, containing a VM used for development purpose (*vmDev*) that contains the three tiers of the application without being monitored; such a VM is called *unmanaged*. The three other VM are all monitored, i.e., *managed*, and each contains a tier of the application. The reader can note that each of the managed VM contains only the observers responsible for monitoring the operating system and the type of service provided. The cloud environment has three provided interfaces: two to provide its service, whether it is or not in a development version, and another one, used for monitoring, connected to a sub-component *monitorObs* bound to all the monitoring interfaces of the managed VM.

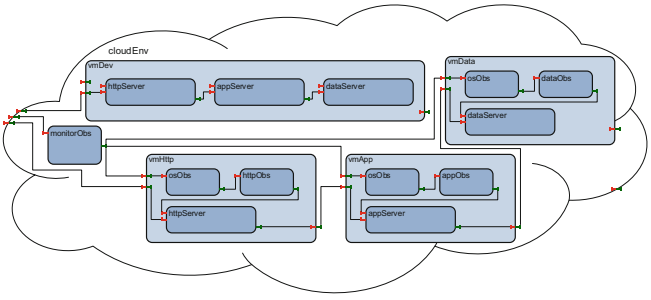

Fig. 2. Cloud Environment Example

A cloud provider must be able to provide on-demand (sets of) VMs configured with the right service components and the appropriate monitoring. In this context, we study the provisioning of a single VM as illustrated in Fig. 1. Depending on the services to provide and the monitoring state (managed vs unmanaged) the necessary components should be added. During the life cycle of the VM some configuration changes can happen; we consider them as reconfigurations of a component-based system. In this context, the challenge consists in performing adequate dynamic reconfigurations with minimum communication overhead, while avoiding reconfigurations that would lead to unwanted behaviours.

3 Component-Based Architecture

3.1 Configurations and Reconfigurations

Component models can be very heterogeneous. Most of them consider software components that can be seen as black boxes (or grey boxes if some of their inner features are visible) having fully-described interfaces. Behaviours and interactions are specified using components' definitions and their interfaces. In this

section, we revisit the architectural reconfiguration model introduced in [5,9]. In general, the system configuration is the specific definition of the elements that define or prescribe what a system is composed of, while a reconfiguration can be seen as a transition from a configuration to another.

Following [9], a configuration is defined to be a set of architectural elements (components, required or provided interfaces, and parameters) together with relations to structure and to link them.

Definition 1 (Configuration). *A configuration c is a tuple $\langle Elem, Rel \rangle$ where*

- $Elem = Components \uplus Interfaces \uplus Parameters \uplus Types$ *is a set of architectural elements, such that*
  - $Components$ *is a non-empty set of the core entities, i.e. components;*
  - $Interfaces = RequiredInts \uplus ProvidedInts$ *is a finite set of the (required and provided) interfaces;*
  - $Parameters$ *is a finite set of component parameters;*
  - $Types = ITypes \uplus PTypes$ *is a finite set of the interface types and the parameter data types;*
- $Rel = \begin{cases} Container \uplus ContainerType \uplus Contingency \\ \uplus\ Parent \uplus Binding \uplus Delegate \uplus State \uplus Value \end{cases}$
  is a set of architectural relations which link architectural elements, such that
  - $Container : Interfaces \uplus Parameters \rightarrow Components$ *is a* total function *giving the component which supplies the considered interface or the component of a considered parameter;*
  - $ContainerType : Interfaces \uplus Parameters \rightarrow Types$ *is a* total function *that associates a type to each (required or provided) interface and to each parameter;*
  - $Contingency : RequiredInts \rightarrow \{mandatory, optional\}$ *is a* total function *indicating whether each required interface is mandatory or optional;*
  - $Parent \subseteq Components \times Components$ *is a* relation *linking a sub-component to the corresponding composite component*[1]*;*
  - $Binding : ProvidedInts \rightarrow RequiredInts$ *is a* partial function *which binds together a provided interface and a required one;*
  - $Delegate : Interfaces \rightarrow Interfaces$ *is a* partial function *to express delegation links;*
  - $State : Components \rightarrow \{started, stopped\}$ *is a* total function *giving the status of instantiated components;*
  - $Value : Parameters \rightarrow \{t | t \in PType\}$ *is a* total function *which gives the current value (of type $t \in PType$) of each parameter.*

We also introduce a set CP of configuration propositions which are constraints on the architectural elements and the relations between them. These

[1] For any $(p, q) \in Parent$, we say that q has a sub-component p, i.e. p is a child of q. Shared components (sub-components of multiple enclosing composite components) can have more than one parent.

propositions are specified using first-order logic formulae [10]. The interpretation of functions, relations, and predicates over *Elem* is done according to basic definitions in [10] and Definition. 1. The interested reader is referred to [5].

Let $\mathcal{C} = \{c, c_1, c_2, \ldots\}$ be a set of configurations. An *interpretation* function $l : \mathcal{C} \rightarrow CP$ gives the largest conjunction of $cp \in CP$ evaluated to true on $c \in \mathcal{C}$. We say that a configuration $c = \langle Elem, Rel \rangle$ satisfies $cp \in CP$, when $l(c) \Rightarrow cp$; in this case, cp is valid on c, otherwise, c does not satisfy cp.

Among the configuration propositions, the architectural *consistency constraints* CC in Table 1 express requirements on component assembly common to all the component architectures [5]. Intuitively,

- a component *supplies*, at least, one provided interface (**CC.1**);
- the composite components have no parameter (**CC.2**);
- a sub-component must not include its own parent component (**CC.3**);
- two bound interfaces must have the same interface type (**CC.4**) and their containers are sub-components of the same composite (**CC.5**);
- when binding two interfaces, there is a need to ensure that they have not been involved in a delegation yet (**CC.6**); similarly, when establishing a delegation link between two interfaces, the specifier must ensure that they have not yet been involved in a binding (**CC.7**);
- a provided (resp. required) interface of a sub-component is delegated to at most one provided (resp. required) interface of its parent component (**CC.8**), (**CC.9**) and (**CC.11**); the interfaces involved in the delegation must have the same interface type (**CC.10**); and
- a component is *started* only if its mandatory required interfaces are bound or delegated (**CC.12**).

Definition 2 (Consistent configuration). *Let $c = \langle Elem, Rel \rangle$ be a configuration and CC the consistency constraints. The configuration c is* consistent*, written* `consistent(c)`*, if $l(c) \Rightarrow CC$. We write* `consistent`$(\mathcal{C})$ *when $\forall c \in \mathcal{C}$.* `consistent`$(c)$.

Table 1. Consistency Constraints

$$\forall c.(c \in Components \Rightarrow (\exists ip.(ip \in ProvidedInts \wedge Container(ip) = c))) \quad \textbf{(CC.1)}$$

$$\forall c, c' \in Components.((c, c') \in Parent \Rightarrow \forall\, p.(p \in Parameters \Rightarrow Container(p) \neq c')) \quad \textbf{(CC.2)}$$

$$\forall c, c' \in Components.((c, c') \in Parent^{+} \Rightarrow c \neq c') \quad \textbf{(CC.3)}$$

$$\forall ip \in ProvidedInts, \forall ir \in RequiredInts \cdot \left(Binding(ip) = ir \Rightarrow \begin{array}{l} ContainerType(ip) = ContainerType(ir) \\ \wedge Container(ip) \neq Container(ir) \end{array}\right) \quad \textbf{(CC.4)}$$

$$\forall ip \in ProvidedInts, \forall ir \in RequiredInts \cdot \left(Binding(ip) = ir \Rightarrow \exists c \in Components. \left(\begin{array}{l} (Container(ip), c) \in Parent \\ \wedge (Container(ir), c) \in Parent \end{array}\right)\right) \quad \textbf{(CC.5)}$$

$$\forall ip \in ProvidedInts, \forall ir \in RequiredInts. \left(Binding(ip) = ir \Rightarrow \begin{array}{l} ip \notin dom(Delegate) \\ \wedge ir \notin dom(Delegate) \end{array}\right) \quad \textbf{(CC.6)}$$

$$\forall i, i' \in Interfaces.(Delegate(i) = i' \Rightarrow i \notin dom(Binding) \wedge i \notin codom(Binding)) \quad \textbf{(CC.7)}$$

$$\forall i, i' \in Interfaces.(Delegate(i) = i' \wedge i \in ProvidedInts \Rightarrow i' \in ProvidedInts) \quad \textbf{(CC.8)}$$

$$\forall i, i' \in Interfaces.(Delegate(i) = i' \wedge i \in RequiredInts \Rightarrow i' \in RequiredInts) \quad \textbf{(CC.9)}$$

$$\forall i, i' \in Interfaces. \left(Delegate(i) = i' \Rightarrow \begin{array}{l} ContainerType(i) = ContainerType(i') \\ \wedge\ (Container(i), Container(i')) \in Parent \end{array}\right) \quad \textbf{(CC.10)}$$

$$\forall i, i', i'' \in Interfaces. \left(\begin{array}{l} (Delegate(i) = i' \wedge Delegate(i) = i'' \Rightarrow i' = i'') \\ \wedge (Delegate(i) = i'' \wedge Delegate(i') = i'' \Rightarrow i = i') \end{array}\right) \quad \textbf{(CC.11)}$$

$$\forall ir \in RequiredInts. \left(\begin{array}{l} State(Container(ir)) = started \\ \wedge Contingency(ir) = mandatory \end{array} \Rightarrow \exists i \in Interfaces. \left(\begin{array}{l} Binding(i) = ir \\ \vee Delegate(ir) = i \end{array}\right)\right) \quad \textbf{(CC.12)}$$

3.2 Reconfiguration Model and Consistency Propagation

Reconfigurations make the component-based architecture evolve dynamically. They are composed of primitive operations such as instantiation/destruction (*new*/*destroy*) of components; addition/removal (*add*/*remove*) of components; binding/unbinding (*bind*/*unbind*) of component interfaces; starting/stopping (*start*/*stop*) components; setting components' parameters values (*update*). These primitive operations obey pre/post predicates. For example, before adding a sub-component $comp_1$ to a composite $comp_2$, one must verify, as in Table 2, that (*a*) $comp_1$ and $comp_2$ exist (2) and are different (3), (*b*) $comp_2$ is not a descendant of $comp_1$ (4), and (*c*) $comp_2$ has no parameter (5). When these preconditions are met, the postcondition consists in adding $(comp_1, comp_2)$ to the $Parent$ relation, as expressed by $R_{add} = Parent \cup \{(comp_1, comp_2)\}$ (1).

Table 2. Preconditions of the *add* primitive reconfiguration operation

$comp_1, comp_2 \in Components$ (2)	$(comp_2, comp_1) \notin Parent^+$ (4)
$comp_1 \neq comp_2$ (3)	$\forall p \in Parameters.Container(p) \neq comp_2$ (5)

Inspired by the predicate-based semantics of programming language constructs [11], we consider a reconfiguration operation *ope*, and two configurations c and c' such that the transition between c and c' is performed using *ope* (denoted by $c \xrightarrow{ope} c'$). Then, given R, some conditions on the configuration of the system under scrutiny, the notation $wp(ope, R)$ denotes, as in [6], the *weakest precondition* for the configuration c such that activation of *ope* can occur and, if so, is guaranteed to lead to c' satisfying the postcondition R. More formally, in our case, if $l(c) \Rightarrow wp(ope, R)$ and $c \xrightarrow{ope} c'$ then $l(c') \Rightarrow R$. Therefore, considering the *add* primitive reconfiguration operation whose preconditions are displayed in Table 2, the weakest precondition $wp(add, R_{add})$ is the conjunction of preconditions (2) to (5).

Inspired by [6] and using the same notations, we propose in Table 3 the grammar of axiom <guarded reconfiguration> for *guarded reconfigurations*. Let <*ope*> represent a primitive reconfiguration operation, also called *primitive statement*. We extend the set of primitive reconfiguration operations with the *skip* operation, which does not induce any change on a given configuration. Hence, for any postcondition R, we have $wp(skip, R) = R$. Afterwards, like in [6], the semantics of the ";" operator is given by $wp(S_1; S_2, R) = wp(S_1, wp(S_2, R))$ where S_1 and S_2 are statements.

Guarded reconfiguration sets are used to define the alternative and the repetitive constructs; these sets are not statements. In a nutshell, the alternative construct selects for execution only guarded lists with a true guard, whereas, the repetitive construct selects for execution guarded lists with a true guard and is

Table 3. Guarded reconfigurations grammar

<guarded reconfiguration>	::=	<guard> → <guarded list>
<guard>	::=	<boolean expression>
<guarded list>	::=	<statement> {; <statement>}
<guarded reconfiguration set>	::=	<guarded reconfiguration> {‖ <guarded reconfiguration>}
<alternative construct>	::=	**if** <guarded reconfiguration set> **fi**
<repetitive construct>	::=	**do** <guarded reconfiguration set> **od**
<statement>	::=	<alternative construct> | <repetitive construct> | < *ope* >

repeated until none of the guards is true. If a guarded reconfiguration set is made of more than one guarded reconfiguration, they are separated by the ‖ operator[2]

To present the semantics of the alternative construct, let IF denote **if** $B_1 \rightarrow S_1 [\!] \ldots [\!] B_n \rightarrow S_n$ **fi** and BB denote $(\exists i : 1 \leq i \leq n : B_i)$, then $wp(IF, R) = BB \wedge (\forall i : 1 \leq i \leq n : B_i \Rightarrow wp(S_i, R))$. For the repetitive construct, let DO denote **do** $B_1 \rightarrow S_1 [\!] \ldots [\!] B_n \rightarrow S_n$ **do**. Let $H_0(R) = R \wedge \neg BB$ and for $k > 0$, $H_k(R) = wp(IF, H_{k-1}(R)) \vee H_0(R)$, then $wp(DO, R) = \exists k : k \geq 0 : H_k(R)$. Intuitively, $H_k(R)$ is the weakest precondition guaranteeing termination after at most k selections of a guarded list, leaving the system in a configuration such that R holds. Let $\mathcal{R}_{run} = \mathcal{R} \cup \{run\}$ be a set of operations, where $\mathcal{R}$ is a finite set of guarded reconfigurations instantiated wrt. the system under consideration, and run is the name of a generic action representing all the running operations[3] of the component-based system.

Definition 3 (Reconfiguration model). *The operational semantics of a component-based system is defined by the labelled transition system* $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$ *where* $\mathcal{C} = \{c, c_1, c_2, \ldots\}$ *is a set of configurations,* $\mathcal{C}^0 \subseteq \mathcal{C}$ *is a set of initial configurations,* $\rightarrow \subseteq \mathcal{C} \times \mathcal{R}_{run} \times \mathcal{C}$ *is the reconfiguration relation obeying wp() predicates, and* $l : \mathcal{C} \rightarrow CP$ *is a total interpretation function.*

Let us note $c \overset{ope}{\rightarrow} c'$ for $(c, ope, c') \in \rightarrow$. Given the model $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$, a path σ of S is a sequence of configurations $c_0, c_1, c_2, \ldots$ such that $\forall i \geq 0. \exists\, ope_i \in \mathcal{R}_{run}.(c_i \overset{ope_i}{\rightarrow} c_{i+1})$. An execution is a path σ in Σ s.t. $\sigma(0) \in \mathcal{C}^0$. We write $\sigma(i)$ to denote the i-th configuration of σ. The notation σ_i denotes the suffix path $\sigma(i), \sigma(i+1), \ldots$, and σ_i^j denotes the segment path $\sigma(i), \sigma(i+1), \ldots, \sigma(j-1), \sigma(j)$. Let Σ denote the set of paths, and Σ^f $(\subseteq \Sigma)$ the set of finite paths. A configuration c' is reachable from c when there is a path $\sigma = c_0, c_1, \ldots, c_n$ in Σ^f s.t. $c = c_0$ and $c' = c_n$ with $n \geq 0$. Let c be a configuration, the set of all configurations reachable from c is denoted $reach(c)$. This notion can be lifted from configurations to sets of configurations by $reach(\mathcal{C}) = \{reach(c) \,|\, c \in \mathcal{C}\}$.

Proposition 1 (Consistency propagation). *Given* $\mathcal{C}^0 \subseteq \mathcal{C}$, `consistent`$(\mathcal{C}^0)$ *implies* `consistent`$(reach(\mathcal{C}^0))$.

[2] As in [6], the order in which guarded reconfigurations appear is semantically irrelevant.

[3] The normal running of different components also changes the architecture, e.g., by modifying parameter values or stopping components.

Proof (sketch). We start the proof (see [12] for a more complete proof) by showing that each primitive operation *ope* preserves configuration consistency. We use this result to establish (by induction) that a guarded reconfiguration having a sequence of primitive statements in its guarded list also preserves consistency.

This allows us to show that guarded reconfigurations having a statement based on a guarded reconfiguration set made only of primitive statements ($G \rightarrow$ **fi** grs **fi** or $G \rightarrow$ **do** grs **od**, where grs denotes $B_0 \rightarrow ope_0 [\!] B_1 \rightarrow ope_1 [\!] \ldots [\!] B_n \rightarrow ope_n$) also preserve consistency using only hypothesis on the statements' preconditions and postconditions.

Therefore, with the same reasoning, considering general (i.e., primitive or non primitive) statements instead of only primitive ones and using only hypothesis on statements' preconditions and postconditions, we can prove that consistency is preserved *a*) for guarded reconfigurations having a guarded list composed of a sequence of (non primitive) statements ($G \rightarrow S_0; S_1; \ldots; S_n$) and *b*) for guarded reconfigurations having as guarded list a statement ($G \rightarrow$ **fi** grs **fi** or $G \rightarrow$ **do** grs **od**, where grs denotes $B_0 \rightarrow S_0 [\!] B_1 \rightarrow S_1 [\!] \ldots [\!] B_n \rightarrow S_n$). □

4 Interpreted Architecture Model

In the specification model, primitive operations and guarded reconfigurations were left abstract enough and *run* was uninterpreted. A formal semantics for the component-based system with interpreted operations can be obtained by enriching the configurations with more precise memory states and the effect of these actions upon memory.

4.1 Interpreted Configurations and Reconfigurations

Let us consider a set (infinite, in general) $GM = \{u, ...\}$ of shared global memory states, and a set (infinite, in general) $LM = \{v, ...\}$ of memory states local to a given component. These memory states are read and modified by the primitive and non-primitive reconfigurations, and also by actions implementing *run*.

Interpreted configurations. In addition to already interpreted parameters and interfaces (cf. [5] for more detail), the state of components can be described more precisely by using local memory states. The set of the interpreted states of components is the least set $State_{\mathcal{I}}$ s.t. if $s_1, \ldots, s_n$ are elements in $State_{\mathcal{I}}$[4], $v_1, \ldots, v_n \in LM$ are local memory states, then $((s_1, v_1), \ldots, (s_n, v_n))$ is in $State_{\mathcal{I}}$. Then, the set of the interpreted configurations $\mathcal{C}_{\mathcal{I}}$ is defined by $GM \times State_{\mathcal{I}}$.

Interpreted transitions. Our basic assumption is that all primitive actions have a deterministic effect upon the local and global memory, always terminate (either normally or exceptionally), and are effective. For each primitive reconfiguration operation *ope*, the corresponding interpreted reconfiguration, denoted by $\overline{ope}$, has equivalent or stronger preconditions, such that all constructs behave deterministically. A non-deterministic global behaviour is produced by the arbitrary interleaving of components.

[4] Viewed as a relation.

Formally, all the actions $ope \in \mathcal{R}_{run}$ are interpreted as mappings $\overline{ope}$ from $GM \times LM$ into itself. Additionally, there are some actions specific to the interpretation, $\mathcal{R}_{int}$, for example for testing guards. We say that $\mathcal{I} = (GM, LM, (\overline{ope})_{ope \in \mathcal{R}_{run} \cup \mathcal{R}_{int}})$ is an interpretation of the underlying $\mathcal{R}_{run}$. Let $\mathcal{I}_{\mathcal{R}_{run}}$ denote the class of all interpretations. This construction leads to

Definition 4 (Interpreted reconfiguration model). *The interpreted operational semantics of component-based system is defined by the labelled transition system* $S_{\mathcal{I}} = \langle \mathcal{C}_{\mathcal{I}}, \mathcal{C}^0_{\mathcal{I}}, \mathcal{R}_{run_{\mathcal{I}}}, \rightarrow_{\mathcal{I}}, l_{\mathcal{I}} \rangle$ *where* $\mathcal{C}_{\mathcal{I}}$ *is a set of configurations together with their memory states,* $\mathcal{C}^0_{\mathcal{I}}$ *is a set of initial configurations,* $\mathcal{R}_{run_{\mathcal{I}}} = \{\overline{ope} \mid ope \in \mathcal{R}_{run} \cup \mathcal{R}_{int}\}$, $\rightarrow_{\mathcal{I}} \subseteq \mathcal{C}_{\mathcal{I}} \times \mathcal{R}_{run_{\mathcal{I}}} \times \mathcal{C}_{\mathcal{I}}$ *is the interpreted reconfiguration relation, and* $l_{\mathcal{I}} : \mathcal{C}_{\mathcal{I}} \rightarrow CP$ *is a total interpretation function.*

It is easy to see that, by construction, $\texttt{consistent}(\mathcal{C}^0_{\mathcal{I}})$. Moreover, if $\texttt{consistent}(c)$ and $c \xrightarrow{\overline{ope}}_{\mathcal{I}} c'$ then $\texttt{consistent}(c')$.

4.2 Compatible Interpretation

To establish links between the reconfiguration model and the corresponding interpreted model, we propose to use a version of the classical τ-simulation quasi-ordering [13], while relabeling the operations in $\mathcal{R}_{int}$ by τ. For all $ope \in \mathcal{R} \cup \{\epsilon\}$, where ϵ denotes the empty word, we write $c \overset{ope}{\Rightarrow} c'$ when there are $n, m \geq 0$ such that $c \xrightarrow{\tau^n ope \tau^m} c'$.

Definition 5 (d-simulation). *Let* $S_1 = \langle \mathcal{C}_1, \mathcal{C}^0_1, \ldots \rangle$ *and* $S_2 = \langle \mathcal{C}_2, \mathcal{C}^0_2, \ldots \rangle$ *be two models over* $\mathcal{R}$. *A binary relation* $\sqsubseteq_d \subseteq \mathcal{C}_1 \times \mathcal{C}_2$ *is a d-simulation iff, for all ope in* $\mathcal{R} \cup \{\epsilon\}$, $(c_1, c_2) \in \sqsubseteq_\tau$ *implies (1) whenever* $c_1 \overset{ope}{\Rightarrow}_1 c'_1$, *then there exists* $c'_2 \in \mathcal{C}_2$ *such that* $c_2 \overset{ope}{\Rightarrow}_2 c'_2$ *and* $(c'_1, c'_2) \in \sqsubseteq_d$, *and (2)* $c_1 \overset{ope}{\not\Rightarrow}$ *implies* $c_2 \overset{ope}{\not\rightarrow}$.

We write $S_1 \sqsubseteq_d S_2$ when $\forall c^0_1 \in \mathcal{C}^0_1 \exists c^0_2 \in \mathcal{C}^0_2 . (c^0_1, c^0_2) \in \sqsubseteq_d$.
Let us consider interpreted reconfiguration operations in $\mathcal{R}_{run_{\mathcal{I}}}$ and the corresponding non-interpreted counterpart in $\mathcal{R}_{run}$. When relabelling the operations in $\mathcal{R}_{int}$ by τ, we can state–modulo the overline notation–that the more abstract model τ-simulates the interpreted model (because of the non-determinism when testing guards in the non-interpreted model); nevertheless, this simulation respects non-divergency.

Theorem 1 (Compatibility). $S_{\mathcal{I}} \sqsubseteq_d S$.

Proof (sketch). There are two cases for $ope \in \mathcal{R}_{run} \cup \mathcal{R}_{int}$. As τ's covering operations in $\mathcal{R}_{int}$ are introduced to evaluate guards of sequences of guarded reconfigurations, they do not form infinite cycles of τ-transitions. So, there always must be a way out of these cycles, if any, by a transition of label $\overline{ope}$.

By construction any primitive reconfiguration operation of the interpreted model has preconditions equivalent to or stronger than its counterpart in the non-interpreted model. This way, by using hypothesis on weakest preconditions

in [6], we can prove that guarded reconfigurations composed of primitive statements, $G \rightarrow \overline{s}$, with $\overline{s} \in \mathcal{R}_{run_{\mathcal{I}}} \backslash \mathcal{R}_{int}$ have preconditions equivalent to or stronger than the corresponding statement $s \in \mathcal{R}_{run}$. Consequently, starting from initial configurations, for any $c_1 \in \mathcal{C}_{\mathcal{I}}$, if $\texttt{consistent}(c_1)$ there is $c_2 \in \mathcal{C}$ s.t. $\texttt{consistent}(c_2)$, and if a guarded reconfiguration $G \rightarrow \overline{s}$ is applied to c_1 there exists a guard G', s.t. $G \Rightarrow G'$ and $G' \rightarrow s$ applies to c_2. Moreover, the consistent target configurations are in $\sqsubseteq_d$ too because of their guards.

If no *ope* can be performed in $c_1 \in \mathcal{C}_{\mathcal{I}}$ after having tested some guards covered by τ, c_1 is not consistent, and consequently neither is $c_2 \in \mathcal{C}$. At this step, only several primitive reconfigurations can be applied, as their preconditions are equivalent, no *ope* can be performed in c_2 either. □

4.3 Property Preservation

Theorem 1 can be exploited for property preservation. For example, as the reachability properties are compatible with $\sqsubseteq_d$, this leads us, consequently, to:

Proposition 2. *If configuration c is not reachable in S, it is not reachable in any $S_{\mathcal{I}}$. Conversely, if configuration c is reachable in S, there exists an interpretation $\mathcal{I}$ such that c is reachable in $S_{\mathcal{I}}$.*

In addition, safety properties expressed via non-reachability properties can be ensured. Moreover, as a consequence of Theorem 1 and Propositions 1 and 2, we can state:

Proposition 3. *Let $S_{\mathcal{I}} = \langle \mathcal{C}_{\mathcal{I}}, \mathcal{C}^0_{\mathcal{I}}, \mathcal{R}_{run_{\mathcal{I}}}, \rightarrow_{\mathcal{I}}, l_{\mathcal{I}} \rangle$ be the interpreted model and $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$ the specification model. Given $\mathcal{C}^0_{\mathcal{I}} \subseteq \mathcal{C}_{\mathcal{I}}$, if $S_{\mathcal{I}} \sqsubseteq_d S$ then $\texttt{consistent}(\mathcal{C}^0_{\mathcal{I}})$ implies $\texttt{consistent}(reach(\mathcal{C}^0_{\mathcal{I}}))$.*

It must be noticed that differently from [3], we do not assume that the reconfigurations always make evolve the component assembly from one consistent architecture to another consistent architecture, only through a path of consistent configurations. Indeed, this assumption seems to be too strong notably wrt. primitive reconfigurations.

5 Implementation and Architecture Conformance

5.1 Implementation Protocol

We developed a prototype tool, contained in a java package named *cbsdr*[5], supporting the interpreted reconfiguration model to design and simulate component-based systems with dynamic reconfigurations. Using generic java classes, we can use our implementation to perform reconfigurations on applications deployed using Fractal [7] or FraSCAti [8]. The Fractal framework is based on a hierarchical and reflective component model. Its goal is to reduce the development, deployment, and maintenance costs of software systems in general[6]. FraSCAti is an open-source implementation of the *Service Component Architecture*[7] (SCA).

[5] *cbsdr* stands for Component-Based System Dynamic Reconfiguration.
[6] http://fractal.ow2.org/tutorial/index.html.
[7] http://www.oasis-opencsa.org/sca.

It can be seen as a framework having a Fractal base with an extra layer implementing SCA specifications. In [8], a smart home scenario illustrates the capabilities and the various reconfigurations of the FraSCAti platform.

Figure 3 shows the *cbsdr* interface displaying a given state of the VM from our running example developed using Fractal (top frame). The left frame shows the various states of the run under scrutiny, whereas the bottom frame can be used to display various information such as the evolution of parameters of the model, console output, or the outcome of reconfigurations performed.

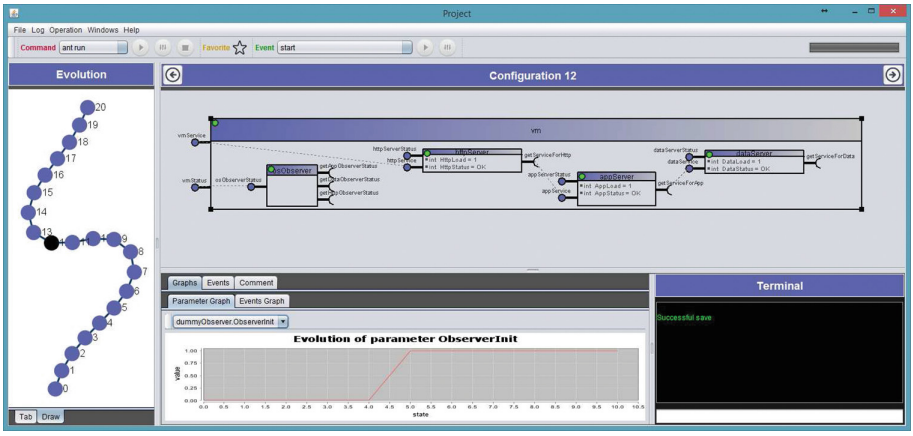

Fig. 3. Model of the VM component-based system displayed in our interface

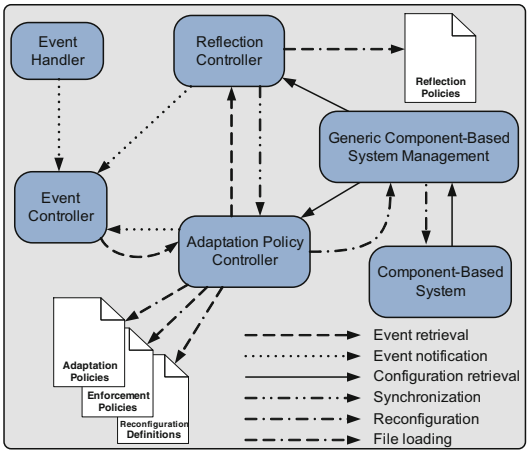

Fig. 4. *cbsdr*Implementation Architecture

This interface allows the monitoring of a component-based system and the generation of (external) events during a run of *cbsdr*, but can also be used to analyse the logs of a run already performed. It is interesting to note that primitive, as well as, non primitive reconfiguration operations can be performed and analysed.

Thanks to this application, in addition to the abovementioned functionalities, we are able to perform adaptations using dynamic reconfigurations triggered by temporal properties at runtime, as described in [2]. In this case, the implementation (see Fig. 4) works as follows: (*a*) adaptation polices are loaded and applied using a control

loop, (*b*) FTPL[8] expressions are evaluated and (if any) candidate reconfigurations are ordered by priority using fuzzy logic values embedded in adaptation policies, (*c*) candidate reconfigurations are applied to the component-based system model using our reconfiguration semantics in order to verify that the corresponding target configuration does not violate any of the properties to enforce, and (*d*) the target configuration obtained using the reconfiguration with highest priority that does not violate any of the properties to enforce is applied to the component based system using a protocol similar to the one described in [3]. The fact that we are using temporal properties based on architectural relations as well as internal and external events allows us to significantly reduce communication overhead (*a*) by, as in [14], using decentralised evaluation of temporal properties or (*b*) by allowing the user to submit simultaneous (external) events to the system, as explained below.

5.2 Architecture Conformance

The reconfiguration model is a correct approximation of the more realistic interpreted implementation model. This fact can be expressed by using the notion of conformance of the component architecture model. Basically, following the most commonly used *ioco* relation in [15], an implementation $S_{\mathcal{I}}$ is conformant to its specification S if, after a trace of S, one should foresee the output of $S_{\mathcal{I}}$ in S, and the implementation is authorised to reach a state where it cannot produce any output only if this is the case in the specification too.

Using various simulation relations permits expressing trace-inclusion-based conformance and stronger conformance relations at the level of transition systems. Thus, thanks to the proof arguments of Theorem 1, and the subsequent trace inclusion modulo τ, we have the following conformance result, with $\mathcal{S}_{\mathcal{I}_{cbsdr}}$ being the *cbsdr* implementation.

Proposition 4. *$\mathcal{S}_{\mathcal{I}_{cbsdr}}$ is conformant to $\mathcal{S}$.*

5.3 Running Example

We consider a VM represented, as in Fig. 1, as a composite component *virtualMachine* that may contain sub-components representing services *httpServer*, *appServer*, or *dataServer* of an application. This VM may also contain *observers* that are sub-components used to monitor services. The sub-component *osObs* is used to monitor the Operating System of the VM and can be bound to the sub-components *httpObs*, *appObs*, or *dataObs* used respectively to monitor the services of the *httpServer*, *appServer*, and *dataServer* sub-components.

The Fractal and FraSCAti versions of the VM example can be controlled by our implementation using external events as *init*, *manage*, *setdata*, etc., to

[8] FTPL stands for TPL (Temporal Pattern Language) prefixed by 'F' to denote its relation to Fractal-like components and to first-order integrity constraints over them.

(respectively) initialise the VM, monitor the VM, or set the data server of the VM up. If the VM is monitored, it is described as *managed*, otherwise it is said to be *unmanaged*. Depending of the service to provide and the state of the VM (managed vs. unmanaged), only the necessary component should be added.

For example, let us consider a managed VM providing only the HTTP service: it contains the *httpServer* component and, since it is managed, it also contains the *osObs* and the *httpObs* components. Therefore, the generation of the *setdata* external event triggers (via adaptation policies) the addition of the *dataServer* and *dataObs* components. Of course, if the initial VM was unmanaged, the generation of the *setdata* external event would only result in the addition of the *dataServer* component. Nevertheless, in this case (unmanaged HTTP VM), the generation of the *setdata* and *manage* external events would result in a VM containing all the components pertaining to a managed VM providing the HTTP and the DATA services (i.e., *httpServer*, *dataServer*, *osObs*, *httpObs*, and *dataObs*).

This is due to the fact that we use FTPL temporal logic expressions as "**after** *unsetdata* ((**always** $\top$) **until** *setdata*)" to guarantee that, in case of opposite events like *setdata* and *unsetdata*, the corresponding expression is potentially true until the occurrence of the opposite events. This way, the ordered sequence of events *init*, *manage*, *sethttp*, *setapp*, and *setdata* is equivalent to a single communication containing all these events at once; this significantly reduces communication overhead.

5.4 Other Examples

To illustrate how the *cbsdr* tool works, we present below two examples: a small http server, and a model of the location component of the *cycab*, an autonomous vehicle. This latter example confirms that not only pertinent reconfigurations can be triggered, but also reconfigurations leading to unwanted behaviours are avoided. Finally, we conclude this section by presenting some results about the CPU consumption of the *cbsdr* tool used with both Fractal and FraSCAti frameworks.

Http Server. Figure 5 shows an experimentation with the http server composite component during which, as in [16], http requests were simulated. Depending on the *load* and *request deviation* to measure whether or not requests are similar, it may make sense to add a cache (the need can be *low*, *medium*, or *high* determining the size of the cache) or an additional file server.

Interestingly, response times measured when our http server is controlled and adapted by the *cbsdr* application match almost exactly the times measured (under similar load and request deviation patterns) for a http server having a cache (of size *high*) and two file servers. No memory nor disk overhead were noted.

Cycab. Figure 6 uses the model of the location system of an autonomous car. Thanks to adaptation using temporal properties at runtime, we can remove

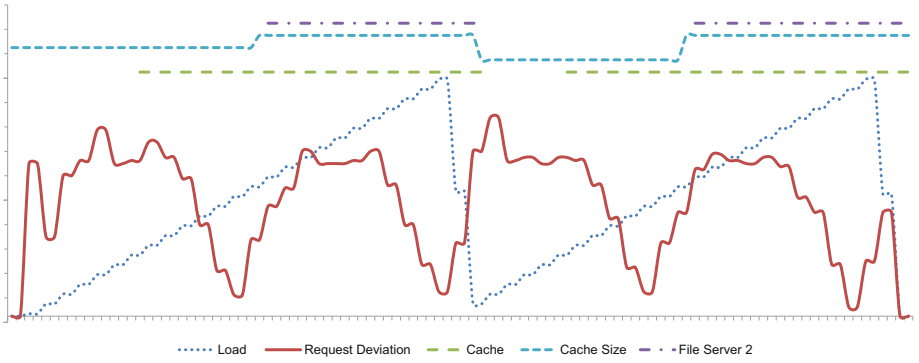

Fig. 5. Experiment with the http server composite component

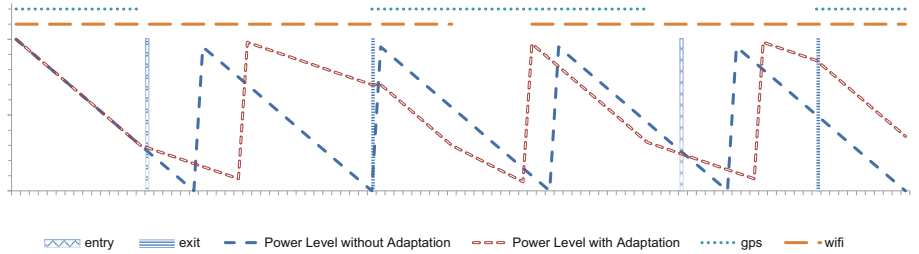

Fig. 6. Experiment with the cycab location composite component

the *gps* or *wifi* location components to save energy when needed (e.g., the *gps* component does not work in tunnels — between *entry* and *exit*).

The run represented in Fig. 6 shows a consumption of energy around 32 % lower using adaptation (empty dashed red graph) compared to a run not using it (full dashed blue graph). It is important to notice that when the vehicle is in a tunnel, the *cbsdr* tool prevents the occurrence of the reconfiguration that would normally add the *gps* component when the power level is high. The reader interested in a more detailed description is referred to [2].

CPU Overhead. We tested our implementation on the above-mentioned examples using both Fractal and FraSCAti framework. More than 300 tests were performed to assess the resources overhead caused by our implementation. Table 4 summarises the increase of CPU usage when adaptation is used compared to similar runs not using any adaptation mechanism. CPU overhead is expressed in Table 4 in the format $\bar{\mu} \pm \sigma$ with $\bar{\mu}$ being the average and σ the standard deviation.

Table 4. Measured increase of CPU usage expressed in percent ($\bar{\mu} \pm \sigma$)

Framework	Fractal	FraSCAti
CPU User time	17 ± 3	11 ± 2
CPU System time	2 ± 2	14 ± 2
Percent of CPU	17 ± 2	15 ± 7

6 Related Work and Conclusion

6.1 Related Work

Self-adaptation is an important and active research field with applications in various domains [1]. This roadmap emphasises an important challenge consisting in bridging the gap between the design and the implementation of self-adaptive systems. In [2] component-based systems reconfiguration was performed at runtime using adaptation policies triggered by temporal patterns. The reconfigurations considered, however, were merely sequences of primitive reconfiguration operations. In the present paper, since we use the alternative and the repetitive constructs to compose reconfigurations, a given reconfiguration can have different outcomes, depending on the context, or due to non-deterministic mechanisms. It is not only a static sequence of reconfiguration instructions (as it is the case in [2,7,8,17]), but a truly *dynamic* reconfiguration. Differently from [3], we do not assume that the reconfigurations always lead the component assembly to evolve from one consistent architecture to another consistent architecture.

Version consistency was introduced in [17] to minimise the interruption of service (*disruption*) and the delay with which component-based (distributed) systems are updated (*timeliness*) by means of reconfigurations. It qualifies a state where transactions within the system are such that a given reconfiguration may not disrupt the system and occur in bounded time; version consistency was inspired by *quiescence* [18] and *tranquility* [19] with the intent to gather the best of both notions. Unlike [17–19], we only consider architectural constraints as preconditions to apply guarded reconfigurations; this way, by considering components as black boxes, the separation of concerns principle is respected. The applicative consistency (related to transactions within the system or external events) can be maintained at runtime using adaptation policies mechanisms as in [2] for centralised system and in [14] for decentralised or distributed systems.

Following [20], our notion of consistency can be viewed as a specific architecture style. Nevertheless, when using graph grammars, we represent interfaces types of the component-based systems by specific graph nodes, this way, like in [21], we can monitor (temporal) properties at the interface level.

Let us remark that the present work is motivated by other frameworks that support the development of components. For example, experimenting with our VM example within the *GROOVE* environment [22] leads us to the presentation of paths with transitions labelled by the primitive reconfiguration operations being performed. Consequently, consistency and conformity issues are pertinent to *GROOVE* too.

6.2 Conclusion

Inspired by [6], we proposed a grammar for guarded reconfigurations. This allowed us to build reconfigurations based on primitive reconfiguration operations using sequences of reconfigurations as well as the alternative and the repetitive constructs. The ability to determine weakest preconditions for the

application of reconfigurations enabled us to prove that these guarded reconfigurations preserve configuration consistency.

This way, a conformance relation can be established to validate implementations of component-based systems architectural models using either our java-based *cbsdr* application or the *GROOVE* graph transformation tool. This makes these tools applicable to build some parts of state space of reachable graphs, i.e., configurations, and thereby derive information about the system. Furthermore, one of the key advantages of this work is that it is readily applicable to practical reconfiguration operations.

As a future work, we intend to exploit the results of the present paper to extend adaptation policies defined in [2] with guarded reconfigurations. Then, we could aim to perform sound and complete compositions of such adaptation policies. This would permit us to move further toward our overall goal, which is the adaptation of component-based system at runtime using adaptation policies based on temporal logic properties.

References

1. de Lemos, R., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013)
2. Kouchnarenko, O., Weber, J.-F.: Adapting component-based systems at runtime via policies with temporal patterns. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) FACS 2013. LNCS, vol. 8348, pp. 234–253. Springer, Heidelberg (2014)
3. Boyer, F., Gruber, O., Pous, D.: Robust reconfigurations of component assemblies. In: International Conference on Software Engineering, ICSE 2013, pp. 13–22. IEEE Press, Piscataway (2013)
4. Myllärniemi, V., Ylikangas, M., Raatikainen, M., Pääkkö, J., Männistö, T., Aaltonen, T.: Configurator-as-a-service: tool support for deriving software architectures at runtime. In: The WICSA/ECSA 2012 Companion Volume, pp. 151–158. ACM (2012)
5. Lanoix, A., Dormoy, J., Kouchnarenko, O.: Combining proof and model-checking to validate reconfigurable architectures. ENTCS **279**, 43–57 (2011)
6. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**, 453–457 (1975)
7. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The fractal component model and its support in java. Soft. Pract. Experience **36**, 1257–1284 (2006)
8. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.B.: A component-based middleware platform for reconfigurable service-oriented architectures. Softw. Pract. Experience **42**, 559–583 (2012)
9. Dormoy, J., Kouchnarenko, O., Lanoix, A.: Using temporal logic for dynamic reconfigurations of components. In: Barbosa, L.S., Lumpe, M. (eds.) FACS 2010. LNCS, vol. 6921, pp. 200–217. Springer, Heidelberg (2012)
10. Hamilton, A.G.: Logic for mathematicians. Cambridge University Press, England (1988)

11. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**, 576–580 (1969)
12. Kouchnarenko, O., Weber, J.F.: Practical Analysis Framework for Component Systems with Dynamic Reconfigurations (2015) Long version of the present paper – https://hal.archives-ouvertes.fr/hal-01135720
13. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Upper Saddle River (1989)
14. Kouchnarenko, O., Weber, J.-F.: Decentralised evaluation of temporal patterns over component-based systems at runtime. In: Lanese, I., Madelaine, E. (eds.) FACS 2014. LNCS, vol. 8997, pp. 108–126. Springer, Heidelberg (2015)
15. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Softw. Concepts Tools **17**, 103–120 (1996)
16. Chauvel, F., Barais, O., Plouzeau, N., Borne, I., Jézéquel, J.M.: Composition et expression qualitative de politiques d'adaptation pour les composants Fractal. In: Actes des Journées nationales du GDR GPL (2009)
17. Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V., Lu, J.: Version-consistent dynamic reconfiguration of component-based distributed systems. In: The 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of software engineering, pp. 245–255. ACM (2011)
18. Kramer, J., Magee, J.: The evolving philosophers problem: dynamic change management. IEEE Trans. Software Eng. **16**, 1293–1306 (1990)
19. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: a low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Trans. Software Eng. **33**, 856–868 (2007)
20. Le Metayer, D.: Describing software architecture styles using graph grammars. IEEE Trans. Software Eng. **24**, 521–533 (1998)
21. Kähkönen, K., Lampinen, J., Heljanko, K., Niemelä, I.: The LIME interface specification language and runtime monitoring tool. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 93–100. Springer, Heidelberg (2009)
22. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. Int. J. Softw. Tools Technol. Transfer **14**, 15–40 (2012)