

Component-Based Systems Reconfigurations Using Graph Transformations with *GROOVE*¹

Olga Kouchnarenko* and Jean-François Weber**

FEMTO-ST CNRS and University Bourgogne Franche-Comté, Besançon, France

*e-mail: okouchnarenko@femto-st.fr

**e-mail: jfweber@femto-st.fr

Received October 15, 2016

Abstract—Component-based systems permit standardisation and re-usability of code through the use of components. The architecture of component-based systems can be modified thanks to dynamic reconfigurations, which contribute to systems' (self-)adaptation by adding or removing components without incurring any system downtime. In this context, the present article describes a formal model for dynamic reconfigurations of component-based systems. It provides a way of expressing runtime reconfigurations of a system and proving their correctness according to a static invariant for consistency constraints and/or a user-provided post-condition. Guarded reconfigurations allow us to build reconfigurations based on primitive reconfiguration operations using sequences of reconfigurations and the alternative and the repetitive constructs, while preserving configuration consistency. A practical contribution consists of the implementation of a component-based model using the *GROOVE* graph transformation tool. This implementation is illustrated on a cloud-based multi-tier application hosting environment managed as a component-based system. In addition, after enriching the model with interpreted configurations and reconfigurations in a consistency compatible manner, component systems' implementations are related to their specifications by a simulation relation.

Keywords: component-based systems, dynamic reconfigurations, consistency, simulation relation, implementation, *GROOVE*

DOI: 10.3103/S014641161707015X

1. INTRODUCTION

Component-based systems permit standardisation and re-usability of code through the use of components; this approach also favours the separation of concerns principle. The architecture of component-based systems can be modified thanks to dynamic reconfigurations. They contribute to systems' (self-)adaptation [1] by adding or removing components without incurring any system downtime. Dynamic reconfigurations depending on systems' environment must happen not only in suitable circumstances, but also need to preserve the consistency of systems. Whereas the former can be ensured by adaptation policies, the latter is directly related to the definition of reconfigurations.

With relation to consistency constraints over component-based systems described in [2], 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 of sequences, repetitions, or choices over primitive reconfiguration operations, there is a need to express reconfigurations' preconditions and postconditions in a precise and concise way. For this reason, to express non primitive *guarded reconfigurations* we use the concept of *weakest precondition*, introduced in [3].

Furthermore, using the *GROOVE* graph transformation tool [4], we build an implementation to perform dynamic reconfigurations on graph-based models of component-based systems. This practical contribution allows us, not only, to simulate the run of a system being reconfigured, but, also, to generate all the possible reconfiguration combinations, or a subset of them. Since the present work aims at formalising reconfigurations using graph transformations, the third and main contribution consists in proving the correctness of interpreted systems, using graph rules to perform reconfigurations, wrt. our reconfiguration model. This also demonstrates the correctness of our implementation.

¹ The article was translated by the authors.

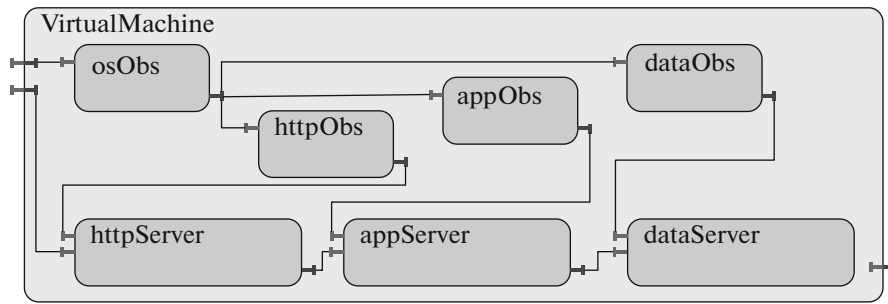


Fig. 1. Managed virtual machine with three-tier application components.

Let us remark that this work is motivated by applications in numerous frameworks that support the development of components together with their monitors/controllers, as, e.g., Fractal [5], CSP||B [6], FraSCAti [7], etc.

The paper is organised as follows: Section 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 Section 3. Using our case study, Section 4 describes an implementation of our model using the *GROOVE* tool to express reconfigurations by means of graph transformations. Finally, Section 5 shows correctness results, and Section 6 presents related work and our conclusion.

2. CLOUD ENVIRONMENT EXAMPLE

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.

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.

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 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.

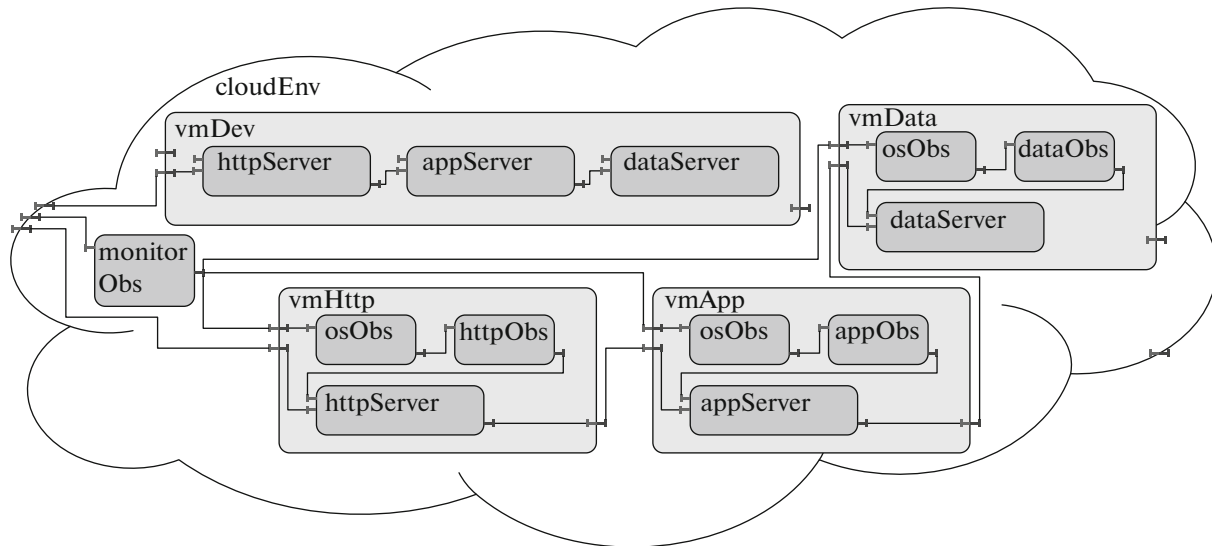


Fig. 2. Cloud environment example.

3. COMPONENT-BASED MODEL

3.1. Consistent Configurations

Component models are very heterogeneous. In general, software components are seen as black boxes having fully described interfaces, or grey boxes if some of their inner features are visible. The interactions are then specified using components' definitions and their interfaces. A recent survey on software architecture can be found in [8].

In this section, we revisit the architectural reconfiguration model introduced in [2, 9]. 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;
- $Rel = \left\{ \begin{array}{l} Container \uplus ContainerType \uplus Contingency \\ \uplus Parent \uplus Depth \uplus Binding \uplus Delegate \uplus State \uplus Value \end{array} \right.$

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 component²;
- $Depth : Components \rightarrow \mathbb{N}$ is a total function which gives the depth³ of components;
- $Binding : ProvidedInts \rightarrow RequiredInts$ is a partial function which binds together a provided interface and a required one;

² 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.

³ It is linked with $Parent$ relation.

- *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 $\{t \mid t \in PType\}$ is a total function which gives the current value 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 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 Def. 1. The interested reader is referred to [2].

Let $\mathcal{C} = \{c, c_1, c_2, \dots\}$ 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 [2]. 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);
- 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 $consistent(C)$ when $\forall c \in C. consistent(c)$.

The reader interested in a detailed description of our consistency constraints is referred to [2].

3.2. Reconfigurations and Consistency Propagation

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. Reconfigurations 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 parameter values of components (*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)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 $R_{add} = Parent \cup \{(comp_1, comp_2)\}(1)$.

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 [3], 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).

⁴ In the paper, the transitive closure of *Parent* relation is used for pre/post-conditions only to simplify their writing/readability; instead, the *Depth* relation can be used to express them as first order logic formulae.

Table 1. Consistency constraints

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

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

$$\forall c, c' \in \text{Components}.((c, c') \in \text{Parent} \Rightarrow \text{Depth}(c') < \text{Depth}(c)) \quad (\text{CC.3})$$

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

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

$$\forall ip \in \text{ProvidedInts}, \left(\text{Binding}(ip) = ir \Rightarrow \begin{array}{l} \text{Delegate}(ip) \neq id \\ \wedge \text{Delegate}(ir) \neq id \end{array} \right) \quad (\text{CC.6})$$

$$\forall i, i' \in \text{Interfaces}. \left(\text{Delegate}(i) = i' \Rightarrow \begin{array}{l} \forall ip.(ip \in \text{ProvidedInts} \Rightarrow \text{Binding}(ip) \neq i) \\ \wedge \forall ir.(ir \in \text{RequiredInts} \Rightarrow \text{Binding}(ir) \neq i) \end{array} \right) \quad (\text{CC.7})$$

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

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

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

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

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

Table 2. Preconditions of the primitive reconfiguration operation

$$\text{comp}_1, \text{comp}_2 \in \text{Components} \quad (2) \quad (\text{comp}_2, \text{comp}_1) \notin \text{Parent}^+ \quad (4)$$

$$\text{comp}_1 \neq \text{comp}_2 \quad (3) \quad \forall p \in \text{Parameters}. (p, \text{comp}_2) \notin \text{Container} \quad (5)$$

Inspired by [3] and using the same notations, Table 3 provides the grammar of axiom $\langle \text{guarded reconfiguration} \rangle$ for *guarded reconfigurations*. Let $\langle \text{ope} \rangle$ 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 $wp(\text{skip}, R) = R$. Afterwards, like in [3], 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 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 [] operator⁵.

⁵ As in [3], the order in which guarded reconfigurations appear is semantically irrelevant.

Table 3. Guarded reconfigurations grammar

$\langle \text{guarded reconfiguration} \rangle$	$::=$	$\langle \text{guard} \rangle \rightarrow \langle \text{guarded list} \rangle$
$\langle \text{guard} \rangle$	$::=$	$\langle \text{boolean expression} \rangle$
$\langle \text{guarded list} \rangle$	$::=$	$\langle \text{statement} \rangle \{; \langle \text{statement} \rangle\}$
$\langle \text{guarded reconfiguration set} \rangle$	$::=$	$\langle \text{guarded reconfiguration} \rangle \{ \langle \text{guarded reconfiguration} \rangle \}$
$\langle \text{alternative construct} \rangle$	$::=$	if $\langle \text{guarded reconfiguration set} \rangle$ fi
$\langle \text{repetitive construct} \rangle$	$::=$	do $\langle \text{guarded reconfiguration set} \rangle$ od
$\langle \text{statement} \rangle$	$::=$	$\langle \text{alternative construct} \rangle \mid \langle \text{repetitive construct} \rangle \mid \langle \text{ope} \rangle$

To present the semantics of the alternative construct, let IF denote **if** $B_1 \rightarrow S_1 \{ \dots \} B_n \rightarrow S_n$ **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** $B_1 \rightarrow S_1 \{ \dots \} B_n \rightarrow S_n$ **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 operations⁶ of the component-based system.

Definition 3 (Operational semantics). *The structural operational semantics of a component-based system with reconfigurations is defined by the labelled transition system $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$ where $\mathcal{C} = \{c, c_1, c_2, \dots\}$ 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 $c \xrightarrow{ope} c'$ 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, \dots such that $\forall i \geq 0. \exists ope_i \in \mathcal{R}_{run} (c_i \xrightarrow{ope_i} c_{i+1})$. 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 $\sigma(i), \sigma(i+1), \dots$, and σ_i^j denotes the segment path $\sigma(i), \sigma(i+1), \dots, \sigma(j-1), \sigma(j)$. Let Σ denote the set of paths, and $\Sigma^f (\subseteq \Sigma)$ the set of finite paths. A configuration c' is reachable from c when there is a path $\sigma = c_0, c_1, \dots, c_n$ 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 $reach(\mathcal{C}) = \{reach(c) \mid c \in \mathcal{C}\}$.

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

Proof (sketch). We start the proof by establishing that each primitive operation ope preserves configuration consistency. This means, for R being a postcondition of ope , that we have $CC \wedge wp(ope, R) = wp(ope, CC \wedge R)$. We show this result for $add(,)$ the proof is similar for the other primitive operations. Let be c such that consistent(c) and the preconditions of $add(,)$ hold on c . Then, the transition $c \xrightarrow{add} c'$ leads to configuration c' such that consistent(c'), i.e., that the postconditions of $add(,)$ satisfy the consistency constraints of Table 1 too; formally, $(l(c) \Rightarrow CC \wedge wp(add, R_{add})) \wedge (c \xrightarrow{add} c') \Rightarrow (l(c') \Rightarrow CC \wedge R_{add})$. Indeed, as the *Parent* relation from the postcondition (1) is not involved in (CC.1), (CC.4) to (CC.9), (CC.11), and (CC.12), these constraints hold on c' too. For the remaining constraints, one has:

(CC.2): As precondition (5) of Table 2 ensures that the parent component $comp_2$ has no parameters, (CC.2) holds on c' with $(comp_1, comp_2)$ added to *Parent* (cf. (1));

⁶ The normal crossing of different components also leads to configuration change, e.g., by modifying parameter values. However, following [12], we consider that those operations do not change system architecture.

(CC.3): Precondition (4) of Table 2 means that $comp_2$ cannot be a descendant of $comp_1$, thus preventing a cycle in the *Parent* relation for c' when $comp_2$ becomes a parent of $comp_1$;

(CC.10): There are two cases: Either there already was a delegation relation between interfaces of $comp_1$ and $comp_2$ on c before the application of the $add(,)$ operation, or not. In the latter case the constraint (CC.10) trivially holds on c' . In the former case, since $consistent(c)$, the *Parent* relation already had $(comp_1, comp_2)$ with well-typed interfaces for c , and the application of $add(,)$ does not change the types and the relation, therefore the constraint holds on c' .

Let be $c \in reach(\mathcal{C}^0)$; by definition, there exists $c_0 \in \mathcal{C}^0$ and a sequence of operations from \mathcal{R}_{run} to ultimately reach c . By definition, there also exists a sequence of primitive operations $ope_0, ope_1, \dotsc, ope_{n-1}$ and a set of intermediate configurations $\mathcal{C}' = \{c_1, c_2, \dots, c_{n-1}\}$ ⁷ such that $c_0 \xrightarrow{ope_0} c_1, c_1 \xrightarrow{ope_1} c_2, \dots, c_{n-1} \xrightarrow{ope_{n-1}} c$, where, for $0 \leq i \leq n-1$, c_i (resp. c_{i+1}) meets the preconditions (resp. postconditions) of ope_i (c_n standing for c). Indeed, if this sequence of primitive operations or \mathcal{C}' would not exist, c would not be reachable from any configuration in \mathcal{C}^0 .

Now, let us prove that a guarded reconfiguration having a sequence of primitive statements in its guarded list preserves consistency. Let gl_n be a guarded list composed of $n \geq 0$ primitive operations, i.e., $gl_n = ope_0; ope_1; \dots; ope_n$, with R_i and R_{i+1} being respectively preconditions and postconditions of ope_i , we note $CC_i = CC \wedge R_i$. Let us prove by induction on n that $CC_0 = wp(gl_n, CC_{n+1})$. For $n = 0$, we have $gl_n = ope_0$ and $CC_0 = wp(gl_0, CC_1)$. Let us now consider $gl_{n+1} = gl_n; ope_{n+1}$; we have $wp(gl_{n+1}, CC_{n+2}) = wp(gl_n, wp(ope_{n+1}, CC_{n+2}))$. Since $CC_0 = wp(gl_n, CC_{n+1})$ and $CC_{n+1} = wp(ope_{n+1}, CC_{n+2})$, we have, by definition [3], $CC_0 = wp(gl_n, CC_{n+1}) = wp(gl_n, wp(ope_{n+1}, CC_{n+2}))$.

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

Therefore, with the same reasoning, considering non primitive statements instead of 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 ($G \rightarrow S_0; S_1; \dots; S_n$) and *b*) for guarded reconfigurations having as guarded list a statement ($G \rightarrow \mathbf{fi} \textit{ grs} \mathbf{fi}$ or $G \rightarrow \mathbf{do} \textit{ grs} \mathbf{od}$, where \textit{grs} denotes $B_0 \rightarrow S_0[] B_1 \rightarrow S_1[] \dots [] B_n \rightarrow S_n$).

4. RECONFIGURATIONS WITH GROOVE

The use of graph transformation systems for modelling and analysing software systems architectures, which originates from [13, 14], is now standard. A graph transformation systems (GTS) is defined by a set of rules that can be used to modify the structure of an initial hypergraph. In general, reduction rules can be defined as graph morphisms. For the basic definitions, the reader is referred to [15].

Following these works, graphs and graph transformation rules are respectively used to represent configurations as well as reconfigurations. This makes standard graph transformation tools applicable to build the state space of reachable graphs, i.e. configurations, and thereby derive information about the system.

This section describes how our model has been implemented within the *GROOVE* graph transformation tool [4]. This implementation is then used to experiment with our case study.

4.1. Implementing with GROOVE

GROOVE uses simple graphs for modelling the structure of object-oriented systems at design-time, compile-time, and runtime. Graphs are made of nodes and edges that can be labelled. Graph transformations provide a basis for model transformation or for operational semantics of systems. Our implementation uses the *GROOVE* typed mode to guarantee that all graphs are well-typed. It consists of generic types

⁷ Note that \mathcal{C}' is not necessarily a subset of \mathcal{C} . For example, if each operation of \mathcal{R} is a sequence of two primitive operations, the intermediary configuration with odd index, i.e., c_1, c_3, \dots , would not belong to \mathcal{C} and $\mathcal{C}' \not\subset \mathcal{C}$.

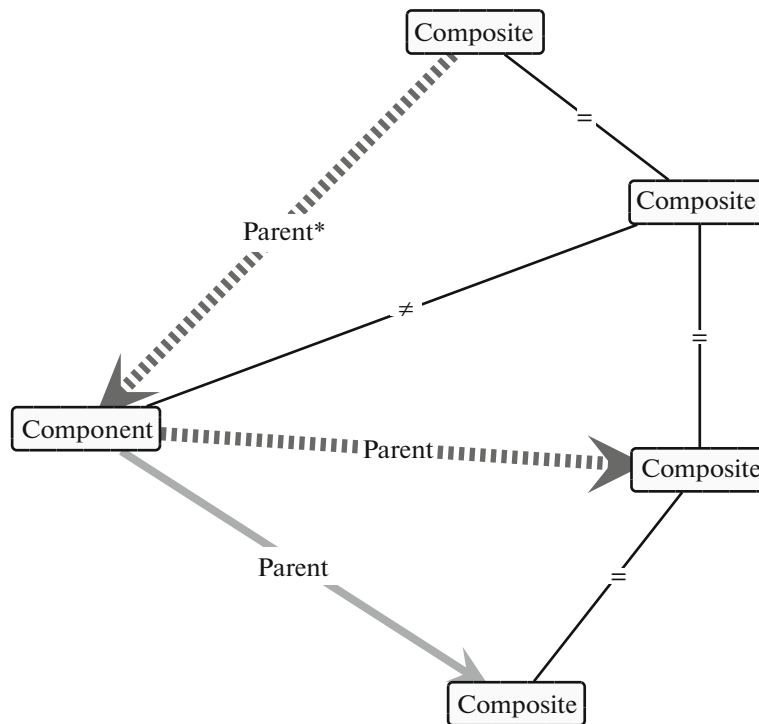


Fig. 3. add primitive operation in *GROOVE*.

and graph rules that can manage assigned priorities in such a way that a rule is applied only if no rule of higher priority matches the current graph.

In *GROOVE* graph transformations obey rules consisting of (a) patterns that must be present (resp. absent) for the rule to apply, (b) elements (nodes and edges) to be added (resp. deleted) from the graph, and (c) pairs of nodes to be merged. Colour and shape coding allow these rules to be easily represented. For example, our implementation models the add primitive operation using the graph rule represented in Fig. 3. In this figure, there are (a) a component and a composite component such that edges labelled “=” ensure that the “Composite” node is the same node of type composite, whereas, the edge labelled “≠” guarantees that the “component” node is a node of type component different from the one labelled “Composite”; (b) the red (dashed fat) which “embargo” edges labelled “*Parent**” (resp. “*Parent*”) ensuring that there is no parent relation transitive closure between nodes labelled “Composite” and “Component” (resp. there is no parent relation between nodes “Component” and “Composite”). If the above-mentioned conditions are satisfied, the green (fat) edge labelled “*Parent*” is created between the nodes “Component” and “Composite.”

Of course, such a graph transformation rule can always be expressed using (a) a LHS (left hand side) sub-graph presenting preconditions of the rule, (b) a NAC (Negative Application Condition) sub-graph specifying what may not occur when matching a rule, and (c) a RHS (right hand side) sub-graph presenting the postconditions. The LHS, NAC, and RHS sub-graphs expressing the rule described in *GROOVE* by Fig. 3 are displayed Fig. 4.

The input of our implementation is a graph containing a component-based system, represented using the model presented in Section 3. Such a graph displays a configuration, as in Def. 1, where elements and relations are respectively represented by nodes and edges.

Figure 5 shows a screenshot of *GROOVE* displaying, in the main panel, a graph modelling the *vmApp* component-based system used in example of Fig. 2. Components are shown with an attribute showing their state (either stopped or started) and have their labels prefixed with CC (Composite Component) or PC (Primitive Component), interfaces names start with a lowercase letter, whereas interface types start with a capital letter. The top left panel shows graph rules ordered by priority, whereas the bottom left panel contains *GROOVE* types.

4.2. 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.

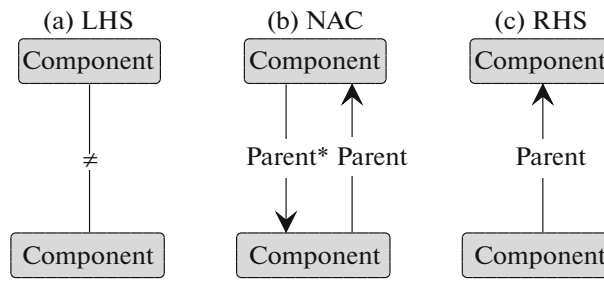


Fig. 4. Equivalent of *GROOVE* rule of Fig. 3 using LHS, NAC, and RHS graphs.

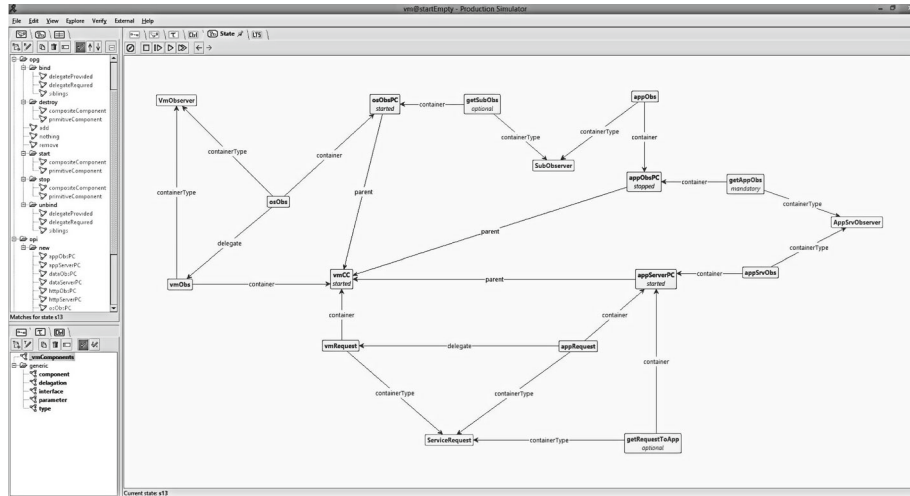


Fig. 5. Model of the *vmApp* component-based system displayed with *GROOVE*.

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.

Each VM has its features determined by an *install code (ic)* which is a binary number having each bit acting as a flag to enable or disable a given feature. This is summarised in Table 4 where the first line displays the features and the second one shows the related bit number. The following lines detail the generation of install codes for a server with a bare OS (*ic* = 0), an application server managed (*ic* = 5), and a managed (resp. unmanaged) LAMP server having 10 (resp. 11) as install code.

Our implementation creates the component-based system model representing the VM specified by a given install code. Figure 6 shows a graph transition system generated by *GROOVE* during the creation of VM with a bare OS (*ic* = 0), where the first state (*s0*) represents an empty graph, *s1* denotes a graph representing only the stopped VM composite component, and *s2* designates a graph with the same component being started. The transitions are labelled by the primitive reconfiguration operations being performed.

Similarly, for a component-based system representing a managed application server (*ic* = 5) the graph transition system is displayed in Fig. 7. In addition to the primitive reconfiguration operations used as transition labels, there is a label “*chk_present_appServerPC*” which represents an



Fig. 6. Bare OS (*ic* = 0).

Table 4. Install code generation principle

Feature	Data	App	http	Managed
Bit #	3	2	1	0
$ic = 0$	0	0	0	0
$ic = 5$	0	$2^2 = 4$	0	$2^0 = 1$
$ic = 10$	$2^3 = 8$	0	$2^1 = 2$	0
$ic = 11$	$2^3 = 8$	0	$2^1 = 2$	$2^0 = 1$

assertion verifying whether or not the application server sub-component is present. This way, using *GROOVE* control language, a function *manage()* adds and configures adequate monitoring sub-components.

For a VM having more than one service component, like a LAMP server ($ic = 10$ or $ic = 11$), having an http and a data service, the order for binding and starting these components is not predetermined as illustrated Fig. 8. The evolution is first performed in a deterministic way from state s_0 to s_8 . State s_{16} , on the top right denotes a graph matching the specification for an install code of value 10, i.e., an unmanaged LAMP server. From that state, we can apply the *manage()* *GROOVE* function, between s_{23} to s_{41} , to obtain a managed LAMP server ($ic = 11$). Let us notice that the evolution between s_8 and s_{16} is non-deterministic. We have two shortest paths ($s_8 \rightarrow s_{10} \rightarrow s_{16}$ and $s_8 \rightarrow s_{11} \rightarrow s_{16}$) that can easily be discovered using a breadth-first exploration.

Table 5 displays the number of states and transitions of the graph transition system for each install code. The graph transition system for $ic = 11$ displayed Fig. 8 has 42 states and 82 transitions. We can notice that, in our implementation, the order of primitive reconfiguration operations is fully determined⁸ for $0 \leq ic \leq 5$ and $8 \leq ic \leq 9$.

Considering Table 5 for $6 \leq ic \leq 7$, and $10 \leq ic \leq 13$, we see that for each VM with two services, the managed version has 16 more states and transitions than the unmanaged one. A similar deduction can also be made considering the last line of Table 5. This is because, as illustrated in Fig. 8, the *manage()* *GROOVE* function fully determines the order of primitive reconfiguration operations. Let us mention that the number of states and transitions for $ic = 6$ (resp. $ic = 7$) is different from the ones for $ic = 10$ or $ic = 12$ (resp. $ic = 11$ or $ic = 13$) due to the fact that, unlike the httpServer or appServer, the dataServer sub-component does not have a required interface (see Fig. 1), which induces more determinism to reach a configuration involving this component.

5. IMPLEMENTATION VS. SPECIFICATION

A formal semantics for the component-based system with interpreted primitive operations and guarded reconfigurations can be obtained by enriching the configurations with more precise memory states and the effect of these actions upon memory.

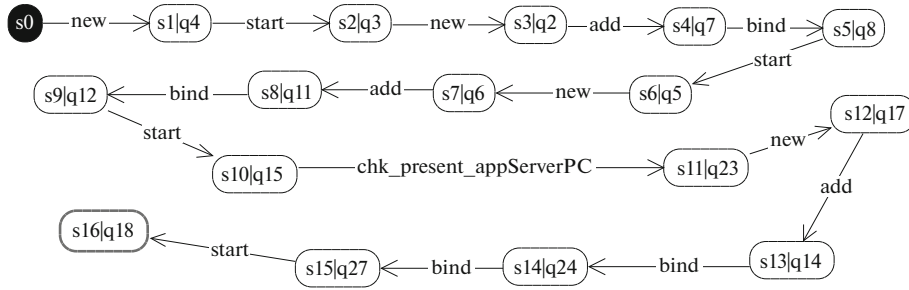
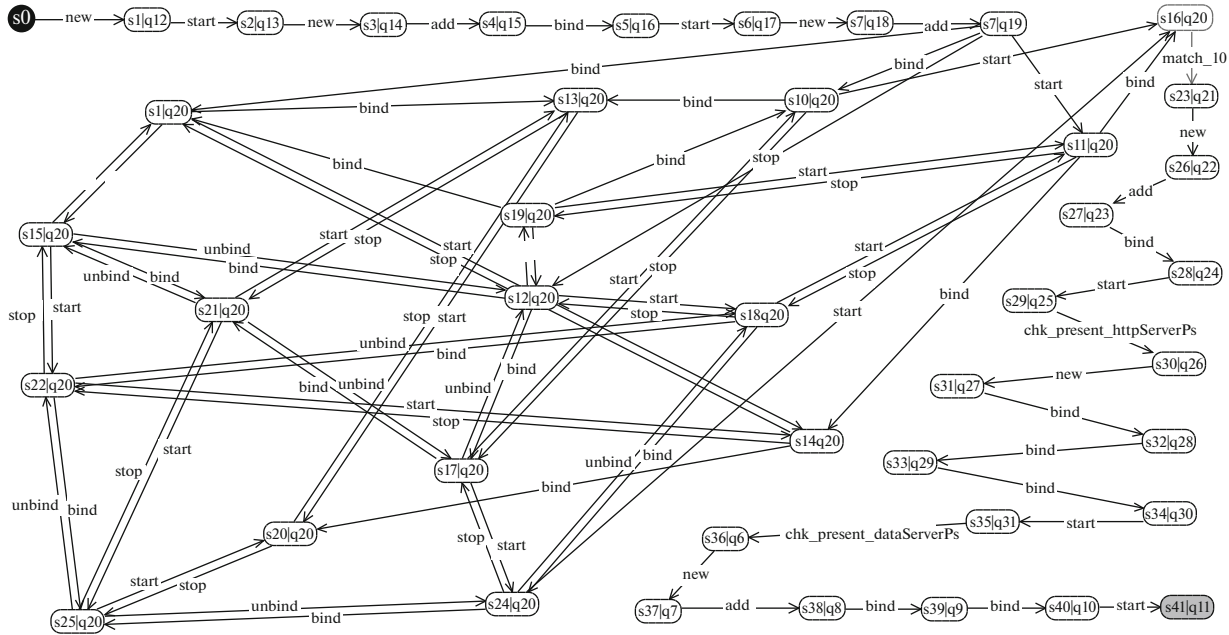
5.1. GROOVE-based Interpreted Model

Let us consider a set (infinite, in general) $GM = \{u, \dots\}$ of shared global memory states, and a set (infinite, in general) $LM = \{v, \dots\}$ 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*. Formally, all the actions $ope \in \mathcal{R}_{run}$ are interpreted as mappings \overline{ope} from $GM \times LM$ into itself. In addition, there are some actions specific to the implementation, \mathcal{R}_{imp} , as *manage* in Section 4. We say that $\mathcal{I} = (GM, LM, (\overline{ope})_{ope \in \mathcal{R}_{run} \cup \mathcal{R}_{imp}})$ is an interpretation of the underlying \mathcal{R}_{run} . Let $\mathcal{I}_{Run} = \{\mathcal{I}, \mathcal{I}_{GROOVE}, \dots\}$ denote the class of all interpretations, with \mathcal{I}_{GROOVE} the underlying *GROOVE* interpretation.

Interpreted configurations. In addition to already interpreted parameters and interfaces (cf. [2] 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, \dots, s_n are elements in $State_{\mathcal{I}}$ ⁹,

⁸ There is exactly one more state than the number of transitions, which shows that the graph transition system is an actual linear path.

⁹ Viewed as a relation.


 Fig. 7. Managed Application Server ($ic = 5$).

 Fig. 8. Managed LAMP Server ($ic = 11$).

$v_1, \dots, v_n \in LM$ are local memory states, then $((s_1, v_1), \dots, (s_n, v_n))$ is in $State_{\mathcal{F}}$. Then, the set of the interpreted configurations $\mathcal{C}_{\mathcal{F}}$ is defined by $GM \times State_{\mathcal{F}}$.

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 the \mathcal{F}_{GROOVE} in Section 4, each graph represents an interpreted configuration corresponding to a configuration in Def. 1, whereas, transitions between configurations are performed using graph rules.

For each primitive reconfiguration operation ope , the corresponding graph rule, denoted by \overline{ope} , has equivalent or stronger preconditions. For example, for the *add* primitive reconfiguration operation, preconditions (3) and (4) of Table 2 are encoded by, respectively, the LHS and NAC graphs (Figs. 4a, 4b) of the corresponding graph rule, whereas, the postcondition (1) is depicted Fig. 4c. Preconditions (2) and (5) are implicitly defined by the typing of the graph rule that contains a node of type *Component*¹⁰ (resp. *Composite*) corresponding to the component $comp_1$ (resp. $comp_2$) of Table 2. Because both nodes involved in the graph rule inherit from the *Component* type, the precondition (2) holds. Furthermore, the fact that the node corresponding to $comp_2$ is typed as composite, ensures that it does not contain any parameters, thus satisfying precondition (5).

Moreover, we can notice, Fig. 4b, in addition to the edge labelled *Parent** satisfying precondition (4), another edge labelled *Parent* ensuring that the node typed *Composite* is not the parent of the other node, i.e., $(comp_1, comp_2) \notin Parent$. This is not a precondition in Table 2 because of a set-based specification.

¹⁰Since this type is abstract, a node typed component is either primitive or composite.

Table 5. Number of states and transitions per install code

Unmanaged			Managed		
<i>ic</i>	states	transitions	<i>ic</i>	states	transitions
0	3	2	1	7	6
2	7	6	3	17	16
4	7	6	5	17	16
6	46	155	7	62	171
8	7	6	9	17	16
10	26	66	11	42	82
12	26	66	13	42	82
14	265	1456	15	288	1479

In the *GROOVE* implementation, however, without this NAC ($comp_1, comp_2$) \notin *Parent*, we may end up with two edges labelled *Parent* between the node typed as *Component* and the one typed as *Composite*, which would produce a graph that would not fit within the specification of Def. 1.

Finally, all constructs now behave deterministically, and a non-deterministic global behavior is produced by the arbitrary interleaving of components. This construction leads to the following definition.

Definition 4 (Implementation semantics). *The operational semantics of the implementation is defined by the labelled transition system $S_g = \langle \mathcal{C}_g, \mathcal{C}_g^0, \mathcal{R}_{run_g}, \rightarrow_g, l_g \rangle$, where \mathcal{C}_g is a set of configurations together with their memory states, \mathcal{C}_g^0 is a set of initial configurations, $\mathcal{R}_{run_g} = \{\overline{ope} \mid ope \in \mathcal{R}_{run} \cup \mathcal{R}_{imp}\}$, $\rightarrow_g \subseteq \mathcal{C}_g \times \mathcal{R}_{run_g} \times \mathcal{C}_g$ is the reconfiguration relation obeying graph rules, and $l_g : \mathcal{C}_g \rightarrow CP$ is a total interpretation function.*

It can be established that, by construction, we have consistent(\mathcal{C}_g^0). Moreover, in the interpreted model, if consistent(*c*) and there is a transition of label \overline{ope} from *c* to *c'* then consistent(*c'*).

5.2. Sound Implementations and Consistency Preservation

There exist some strong links between the interpreted model and the specification model. In this section we aim to establish that our *GROOVE* implementation behaves accordingly to the specification.

Let $\mathcal{R}_{run_g} = \{\overline{ope} \mid ope \in \mathcal{R}_{run} \cup \mathcal{R}_{imp}\}$ be a set of operations, where \overline{ope} is in a finite set of guarded reconfigurations built using primitive graph rules instantiated wrt. the implementation of the system under consideration. For the *GROOVE* implementation, we consider $\mathcal{R}_{imp} = \{match\}$, where *match* represents operations to evaluate the guards used in *GROOVE*, that do not alter the current graph, like “chk_present_appServerPC” in Fig. 7, for control flow purpose.

To establish links between the interpreted model and the specification model, we propose to use a version of the classical τ -simulation quasi-ordering [16], while relabeling the operations in \mathcal{R}_{imp} by τ . For all $ope \in \mathcal{R}$, we write $c \xRightarrow{ope} c'$ when there are $n, m \geq 0$ such that $c \xrightarrow{\tau^{n} ope \tau^m} c'$.

Definition 5 (τ -simulation). *Let S_1 and S_2 be two models over $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$. A binary relation $\sqsubseteq_\tau \subseteq \mathcal{C}_1 \times \mathcal{C}_2$ is a τ -simulation iff, for all ope in \mathcal{R} , (c_1, c_2) in \sqsubseteq_τ implies whenever $c_1 \xrightarrow{ope} c'_1$, then there exists $c'_2 \in \mathcal{C}_2$ such that $c_2 \xrightarrow{ope} c'_2$ and $(c'_1, c'_2) \in \sqsubseteq_\tau$.*

We say that S_1 and S_2 are τ -similar, written $S_1 \sqsubseteq_\tau S_2$, if there is a τ -simulation linking their initial states.

Let us consider interpreted reconfiguration operations in \mathcal{R}_{run_g} and the corresponding non-interpreted counterpart, when relabeling the operations in \mathcal{R}_{imp} by τ , we can state the following theorem.

Theorem 1 (Simulation). $S_g \sqsubseteq_\tau S$.

Proof (sketch). Let $c_{\mathcal{G}}^0$ be in $\mathcal{C}_{\mathcal{G}}^0$. By construction, it is associated with a non-interpreted configuration $c^0 \in \mathcal{C}^0$. There are two cases for $ope \in \mathcal{R}_{run} \cup \mathcal{R}_{imp}$. As τ 's covering operations in \mathcal{R}_{imp} are introduced to evaluate guards of sequences of guarded reconfigurations, they do not form infinite cycles composed only of τ -transitions. So, there always must be a way out of these cycles, if any, by a transition of label \overline{ope} .

Afterwards, it can be established that every primitive reconfiguration operation of the implementation has preconditions equivalent to or stronger than its counterpart in the specification model. This way, by using hypothesis on weakest preconditions in [3], it can be proved that guarded reconfigurations composed of primitive statements of the form $G \rightarrow \bar{s}$ with $\bar{s} \in \mathcal{R}_{run_{\mathcal{G}}} \setminus \mathcal{R}_{imp}$, have preconditions equivalent to or stronger than the corresponding statement $s \in \mathcal{R}_{run}$, as illustrated below.

$$\begin{array}{ccccccccccc}
 S_{\mathcal{G}} & & G_0 & \rightarrow & \bar{s}_0, & G_1 & \rightarrow & \bar{s}_1, & \dots, & G_n & \rightarrow & \bar{s}_n \\
 & & \downarrow & & & \downarrow & & & & \downarrow & & \\
 S & & G'_0 & \rightarrow & s_0, & G'_1 & \rightarrow & s_1, & \dots, & G'_n & \rightarrow & s_n
 \end{array}$$

Consequently, starting from any initial configurations in $\mathcal{C}_{\mathcal{G}}^0$, for any $c_{\mathcal{G}}^1 \in \mathcal{C}_{\mathcal{G}}$, if consistent ($c_{\mathcal{G}}^1$) there is $c^1 \in \mathcal{C}$ s.t. consistent (c^1), and if a guarded reconfiguration $G \rightarrow \bar{s}$ is applied to $c_{\mathcal{G}}^1$ there exists a guard G' , s.t. $G \Rightarrow G'$ and $G' \rightarrow s$ applies to c^1 . Moreover, the consistent target configurations $c_{\mathcal{G}}^2$ and c^2 are in \sqsubseteq_{τ} too because of their guards.

If no ope can be performed in $c_{\mathcal{G}}^1 \in \mathcal{C}_{\mathcal{G}}$ after having tested some guards covered by τ , $c_{\mathcal{G}}^1$ is not consistent, and consequently neither is $c^1 \in \mathcal{C}$. At this step, only several primitive reconfigurations can be applied, as their preconditions are equivalent, no \overline{ope} can be performed in c^1 either.

This result shows that the specification model is a correct approximation of the more realistic interpreted model. As the reachability properties are compatible with \sqsubseteq_{τ} and there is no infinite cycles composed only of τ -transitions, this leads us, consequently, to:

Proposition 2. *If configuration $c_{\mathcal{G}}$ is reachable in $S_{\mathcal{G}}$, its non-interpreted counterpart c is reachable in S . Conversely, if configuration c is not reachable in S , the corresponding interpreted configuration is not reachable in any $S_{\mathcal{G}}$.*

We can state, as a consequence of Theorem 1 and Propositions 1 and 2, the following result:

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

6. RELATED WORK AND CONCLUSION

6.1. Related Work

Self-adaptation is an important and active research field with applications in various domains [1]. Primitive reconfigurations being, as described in [17], the basis of runtime adaptation using temporal properties over architectural relations and external events, component-based system models described, as in this paper, in terms of graph transformations can benefit of (self) adaptation at runtime via temporal properties. Unlike [17], the present paper uses not only sequences of primitive operations but also the alternative and the repetitive constructs to compose reconfigurations. Thus, a given reconfiguration may 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 [5, 7, 17, 18]), but a truly *dynamic* reconfiguration.

Version consistency was introduced in [18] to minimise the interruption of service (*disruption*) and the delay with which component-based (distributed) systems are updated (*timeliness*) by mean of reconfigurations. It qualifies a state where transactions within the system are such that a given reconfiguration may not disrupt the system, and occurs in bounded time; version consistency was inspired by *quiescence* [19] and *tranquility* [20] with the intent to gather the best of both notions. Unlike [18–20], 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 described in [17] for centralised systems and [21] for decentralised or distributed systems.

Differently from [22], we do not assume that the reconfigurations always lead the component assembly to evolve from one consistent architecture to another consistent architecture. Unlike [23], we use exclusively first order logic to define consistency constraints, which makes the consistency propagation of guarded reconfiguration fully applicable to our operational semantics. Following [13], our notion of consistency can be viewed as a specific architecture style. Nevertheless, when using graph transformations, we represent interfaces, parameters and interfaces types of the component-based systems by specific graph nodes, whereas edges represent relations between these elements. We also consider that the parameters, viewed as the *public variables* in [13], can be modified via reconfigurations. As a consequence, thanks to these new features, we can monitor (temporal) properties at the interface level, similarly to [24]. Furthermore, similarly to the *coordinator* in [13], we perform adaptations using *controllers*, as for example in [21]. This can also be described in terms of graph transformations.

6.2. Conclusion

The roadmap in [1] emphasises an important challenge consisting in bridging the gap between the design and the implementation of self-adaptive systems. We have shown that the *GROOVE* framework can help bridge that gap.

Inspired by [3] and following [23], a grammar for guarded reconfigurations 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.

We also, as a practical contribution, implemented our model using the *GROOVE* graph transformation tool [4], where component-based systems are represented as graphs, elements (e.g., components, interface, parameters, etc.) consist of nodes, and relations between elements (e.g., *Parent*, *Bindings*, etc.) are showed as edges. This implementation, used to experiment with our running example (Managed/Unmanaged Cloud Environment), permits to model reconfigurations as graph rules based on LHS, RHS, and NAC graphs. When different outcomes can occur for each reconfiguration, the set of possible executions can be displayed as a LTS graph using our implementation under *GROOVE*; possible states, i.e., configurations of the system under scrutiny, are shown as nodes and reconfigurations between them as edges. We have also been able to prove the correctness of interpreted systems, using graph transformations to perform reconfigurations, wrt. our reconfiguration model, which demonstrates the correctness of our implementation.

As future work, we intend to analyse aforementioned LTS graphs, to detect or prevent the formation of cycles within reconfigurations. We are also planning to implement dynamic reconfigurations based on graph transformations to be able to apply adaptation policies at runtime.

A. Proof of Proposition 3.2 for Constructs Based on Primitive Statements

Let us consider a guarded reconfiguration set *grs* based on guarded reconfigurations containing guarded lists made only of primitive statements. This reconfiguration set, *grs* denotes $B_1' \rightarrow S_1[] \dots [] B_n' \rightarrow S_n$, with B_i' being a boolean and $S_i = ope_0^i; ope_1^i; \dots; ope_{n_i}^i$, where n_i represent the number of primitive statements ($ope_0^i, ope_1^i, \dots, ope_{n_i}^i$) of the guarded list S_i , and R_j^i (resp. R_{j+1}^i) represents the precondition (postcondition) of ope_j^i , for $0 < i \leq n$ and $0 \leq j \leq n_i$ (resp. $0 < j \leq n_i + 1$).

Since R_0^i is the precondition of S_i and we suppose the configuration before the application of S_i to be consistent, we rewrite *grs* as $B_1 \rightarrow S_1[] \dots [] B_n \rightarrow S_n$, with $B_i = B_i' \wedge CC \wedge R_0^i$. We also define $BB = (\exists i : 1 \leq i \leq n : Bi)$, as well as, the sets $I = \{i \in \mathbb{N}. 1 \leq i \leq n\}$ and $I_\top = \{i \in I. Bi\}$.

A.1. Alternative Construct

Let *IF* denote **if** $B_1 \rightarrow S_1[] \dots [] B_n \rightarrow S_n$ **fi**.

By definition, $wp(IF, R) = BB \wedge \forall i \in I : B_i \Rightarrow wp(S_i, R)$. We established before that, for S_i being a sequence of primitive statement, $CC \wedge R_0^i \Rightarrow wp(S_i, CC \wedge R_{n_i+1}^i)$; then, by definition $B_i \Rightarrow wp(S_i, CC \wedge R_{n_i+1}^i) \Rightarrow wp(S_i, CC)$. This means that $wp(IF, CC) = BB \wedge \forall i \in I : B_i \Rightarrow wp(S_i, CC)$, which can be enough to prove that consistency is preserves by the alternative construct.

It is possible, however, to establish a stronger postcondition, $CC \wedge \bigwedge_{i \in I_{\top}} R_{n_i+1}^i$, for the alternative construct by considering that each term of the conjunction $\bigwedge_{i \in I_{\top}} R_{n_i+1}^i$ is part of the postcondition of a guarded list eligible for execution.

Then, $wp\left(IF, CC \wedge \bigwedge_{j \in I_{\top}} R_{n_j+1}^j\right) = BB \wedge \forall i \in I : B_i \Rightarrow wp(S_i, CC \wedge R_{n_i+1}^i)$ because, by definition, $\forall i \in I_{\top}, B_i = \top$.

Therefore:

$$\begin{aligned} BB \wedge CC \wedge \bigwedge_{j \in I_{\perp}} R_0^j &\Rightarrow BB \wedge \bigwedge_{i \in I_{\top}} wp(S_i, CC \wedge R_{n_i+1}^i) \\ &\Rightarrow BB \wedge (\forall i \in I : B_i \Rightarrow wp(S_i, CC \wedge R_{n_i+1}^i)) \\ &\Rightarrow wp\left(IF, CC \wedge \bigwedge_{j \in I_{\top}} R_{n_j+1}^j\right). \end{aligned}$$

As an example, we can denote by **if** B **then** S **fi** a particular case, written **if** $B \rightarrow S[] \neg B \rightarrow skip$ **fi**, of the alternative construct which weakest precondition is $wp(\mathbf{if} B \mathbf{then} S \mathbf{fi}, CC \wedge (B \Rightarrow post_S)) = B \wedge wp(S, CC \wedge R_S)$, where R_S and $post_S$ are, respectively, the precondition and postcondition of S .

A.2. Repetitive Construct

Let DO denote **do** $B_1 \rightarrow S_1[] \dots [] B_n \rightarrow S_n$ **od**. Let be $H_0(R) = R \wedge \neg B$, and for $k > 0$, let be $H_k(R) = wp(IF, H_{k-1}(R)) \vee H_0(R)$, where IF denotes the same guarded configuration enclosed by “**if fi**”. Then, by definition, we have $wp(DO, R) = \exists k : k > 0 : H_k(R)$.

This means that the weakest precondition of this construct guarantees proper termination after at most k selections of a guarded list, leaving the system in a state satisfying R . Let us consider l_0, l_1, \dots, l_k , such that, for $0 \leq j \leq k, 1 \leq l_j \leq n$ and $S_{l_0}; S_{l_1}; \dots; S_{l_k}$, as the ordered sequence of statements selected during the duration of the construct until its termination. We proved before that such a sequence preserve consistency. Therefore $CC \wedge R_{n_{l_k+1}}^{l_k}$ is a valid postcondition and, since $CC \wedge R_{n_{l_k+1}}^{l_k} \Rightarrow CC \wedge \bigwedge_{i \in I} R_{n_i+1}^i$, we have

$$wp(DO, CC \wedge R_{n_{l_k+1}}^{l_k}) \Rightarrow wp\left(DO, CC \wedge \bigwedge_{i \in I} R_{n_i+1}^i\right).$$

We established before that, for S_i being a sequence of primitive statement, $CC \wedge R_0^i \Rightarrow wp(S_i, CC \wedge R_{n_i+1}^i)$; then $CC \wedge R_0^{l_0} \Rightarrow wp(S_{l_0}; S_{l_1}; \dots; S_{l_k}, CC \wedge R_{n_{l_k+1}}^{l_k})$.

Therefore:

$$\begin{aligned} CC \wedge \bigwedge_{i \in I} R_0^i &\Rightarrow CC \wedge R_0^{l_0} \\ &\Rightarrow wp(S_{l_0}; S_{l_1}; \dots; S_{l_k}, CC \wedge R_{n_{l_k+1}}^{l_k}) \end{aligned}$$

for any valid sequence $S_{l_0}; S_{l_1}; \dots; S_{l_k}$

$$\begin{aligned} &\Rightarrow wp(DO, CC \wedge R_{n_{l_k+1}}^{l_k}) \\ &\Rightarrow wp\left(DO, CC \wedge \bigwedge_{i \in I} R_{n_i+1}^i\right). \end{aligned}$$

This proves that the repetitive construct, applied to a guarded reconfiguration set based on guarded reconfigurations containing guarded lists made only of primitive statements, preserves consistency. It also provides stronger preconditions and postconditions that are used to conclude the full proof of Prop 1.

ACKNOWLEDGMENTS

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

REFERENCES

1. De Lemos, R., Giese, H., Müller, H.A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N.M., Vogel, T., et al., Software engineering for self-adaptive systems: A second research roadmap, in *Software Engineering for Self-Adaptive Systems II*, Springer, 2013, pp. 1–32.
2. Lanoix, A., Dormoy, J., and Kouchnarenko, O., Combining proof and model-checking to validate reconfigurable architectures, *Electron. Notes Theor. Comput. Sci.*, 2011, vol. 279, pp. 43–57.
3. Dijkstra, E.W., Guarded commands, nondeterminacy and formal derivation of programs, *Commun. ACM*, 1975, vol. 18, pp. 453–457.
4. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., and Zimakova, M., Modelling and analysis using GROOVE, *Int. J. Software Tools Technol. Transfer*, 2012, vol. 14, pp. 15–40.
5. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., and Stefani, J.B., The fractal component model and its support in java, *Software: Pract. Exper.*, 2006, vol. 36, pp. 1257–1284.
6. Schneider, S. and Treharne, H., Csp theorems for communicating b machines, *Formal Asp. Comput.*, 2005, pp. 390–422.
7. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., and Stefani, J.B., A component-based middleware platform for reconfigurable service-oriented architectures, *Software: Pract. Exper.*, 2012, vol. 42, pp. 559–583.
8. Garlan, D., Software architecture: A travelogue, in *Proceedings of the on Future of Software Engineering, FOSE 2014*, Herbsleb, J.D. and Dwyer, M.B., Eds., Hyderabad, 2014, pp. 29–39.
9. Dormoy, J., Kouchnarenko, O., and Lanoix, A., Using temporal logic for dynamic reconfigurations of components, *Lect. Notes Comput. Sci.*, 2012, vol. 6921, pp. 200–217.
10. Hamilton, A.G., *Logic for Mathematicians*, Cambridge: Cambridge University Press, 1978.
11. Hoare, C.A.R., An axiomatic basis for computer programming, *Commun. ACM*, 1969, vol. 12, pp. 576–580.
12. Garlan, D., Software architecture, in *Wiley Encyclopedia of Computer Science and Engineering*, Wah, B.W., Ed., John Wiley & Sons, Inc., 2008.
13. Le Metayer, D., Describing software architecture styles using graph grammars, *IEEE Trans. Software Eng.*, 1998, vol. 24, pp. 521–533.
14. Hirsch, D., Inverardi, P., and Montanari, U., Modeling software architectures and styles with graph grammars and constraint solving, in *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1), February 22–24, 1999*, Donohoe, P., Ed., San Antonio, 1999, pp. 127–144.
15. *Handbook of Graph Grammars and Computing by Graph Transformations*, vol. 1: *Foundations*, Rozenberg, G., Ed., World Scientific, 1997.
16. Milner, R., *Communication and Concurrency*, Prentice-Hall, Inc., 1989.
17. Kouchnarenko, O. and Weber, J.F., Adapting component-based systems at runtime via policies with temporal patterns, *Lect. Notes Comput. Sci.*, 2014, vol. 8348, pp. 234–253.
18. Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V., and Lu, J., Version-consistent dynamic reconfiguration of component-based distributed systems, in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ACM (2011)*, pp. 245–255.
19. Kramer, J. and Magee, J., The evolving philosophers problem: Dynamic change management, *IEEE Trans. Software Eng.*, 1990, vol. 16, pp. 1293–1306.
20. Vandewoude, Y., Ebraert, P., Berbers, Y., and D’Hondt, T., Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates, *IEEE Trans. Software Eng.*, 2007, pp. 856–868.
21. Kouchnarenko, O. and Weber, J.F., Decentralised evaluation of temporal patterns over component-based systems at runtime, *Lect. Notes Comput. Sci.*, 2015, vol. 8997, pp. 108–126.
22. Boyer, F., Gruber, O., and Pous, D., Robust reconfigurations of component assemblies, *Int. Conf. on Software Engineering, ICSE ’13, Piscataway, NJ*, 2013, pp. 13–22.
23. Kouchnarenko, O. and Weber, J.F., Practical analysis framework for component systems with dynamic reconfigurations, *ICFEM’15, 17th Int. Conf. on Formal Engineering Methods*, 2015, pp. 287–303.
24. Kähkönen, K., Lampinen, J., Heljanko, K., and Niemelä, I., The lime interface specification language and runtime monitoring tool, *Lect. Notes Comput. Sci.*, 2009, vol. 5779, pp. 93–100.