Component-based Systems Reconfigurations Using Graph Grammars

Olga Kouchnarenko, Jean-François Weber

Component-based Systems Reconfigurations Using Graph Grammars*

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

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

Abstract. Dynamic reconfigurations can modify the architecture of component-based systems without incurring any system downtime. In this context, the main contribution of the present article is the establishment of correctness results proving component-based systems reconfigurations using graph grammars. New guarded reconfigurations allow us to build reconfigurations based on primitive reconfiguration operations using sequences of reconfigurations and (unlike most of the related work on reconfigurations) 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 sound implementation is illustrated on a cloud-based multi-tier application hosting environment managed as a component-based 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 consistency of systems. Whereas the former can be ensured by adaptation policies, the latter is directly related to the definition of reconfigurations.

For specifying behaviour properties of component-based systems, a linear temporal logic based on Dwyer's work on patterns and scopes [2] has been proposed in [3]; it is inspired by a JML specification extension using temporal patterns [4]. This logic, called FTPL[3], is used to trigger adaptation policies in [5]. Furthermore, a decentralised evaluation of FTPL properties over sets of components has been studied in [6]. With relation to consistency constraints over component-based systems defined in [7], 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, we

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

[3] 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.

need to express reconfigurations' preconditions and postconditions in a precise and concise way. For this reason, we use the concept of *weakest precondition*, introduced in [8], to express non primitive *guarded reconfigurations*; this is the first contribution.

Furthermore, using the *GROOVE* graph transformation tool [9], we build an implementation using graph grammars to perform dynamic reconfiguration on graph-based models of component-based systems. This second practical contribution allows us, not only, to simulate the run of a system being reconfigured, but, also, to generate all (or a subset of the) possible reconfiguration combinations. Since the present work aims at formalising reconfigurations using graph grammars, 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.

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 [10], CSP‖B [11], FraSCAti [12], 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 Sect. 3. Using our case study, Section 4 describes an implementation of our model using the *GROOVE* tool to express reconfigurations by means of graph grammars. Finally, Section 5 shows correctness results, and 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.

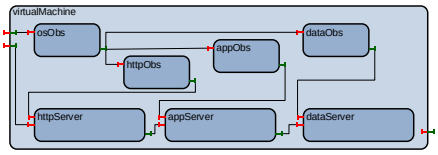


Fig. 1: Managed Virtual Machine with Three-tier Application Compoments

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.

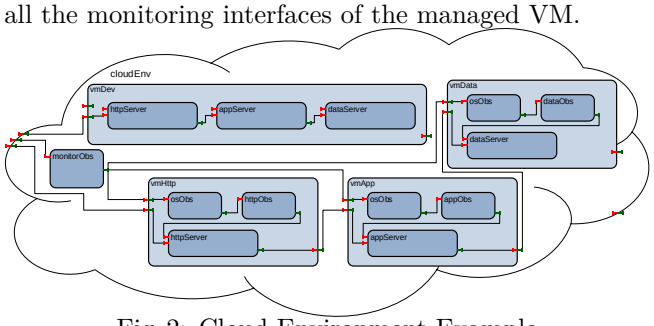


Fig. 2: Cloud Environment Example

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 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.

3 Component-based Model and Semantics

3.1 Configurations and Reconfigurations

Component models can be very heteregeneous. 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 [13,7]. 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 [13], 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
  - $Container : Interfaces \uplus Parameters \to Components$ *is a* total function *giving the component which supplies the considered interface or the component of a considered parameter;*
  - $ContainerType : Interfaces \uplus Parameters \to Types$ *is a* total function *that associates a type to each (required or provided) interface and to each parameter;*
  - $Contingency : RequiredInts \to \{mandatory, optional\}$ *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*[4]*;*
  - $Binding : ProvidedInts \to RequiredInts$ *is a* partial function *which binds together a provided interface and a required one;*
  - $Delegate : Interfaces \to Interfaces$ *is a* partial function *to express delegation links;*
  - $State : Components \to \{started, stopped\}$ *is a* total function *giving the status of instantiated components;*
  - $Value : Parameters \to \{t | t \in PType\}$ *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 [14]. The interpretation of functions, relations, and predicates over $Elem$ is done according to basic definitions in [14] and Def. 1. The interested reader is referred to [7].

Let $\mathcal{C} = \{c, c_1, c_2, \ldots\}$ be a set of configurations. An *interpretation* function $l : \mathcal{C} \to CP$ 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 [7]. Intuitively,

[4] 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.

- 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**).

Table 1: Consistency Constraints

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

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

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

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

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

$$\begin{matrix}\forall ip \in ProvidedInts, \\ \forall ir \in RequiredInts, \forall id \in Interfaces\end{matrix} \cdot \left(Binding(ip) = ir \Rightarrow \begin{matrix} Delegate(ip) \neq id \\ \wedge Delegate(ir) \neq id \end{matrix} \right) \quad \textbf{(CC.6)}$$

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

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

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

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

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

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

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`$(c)$.

3.2 Operational Semantics

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 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), 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)	$\forall p \in Parameters, (p, comp_2) \notin Container$ (5)

Inspired by the predicate-based semantics of programming language constructs [15], we consider a reconfiguration operation *ope*, and two configurations c and c' such that the transition between c and c' is performed using *ope*. Then, given R, some conditions on the configuration of the system under scrutiny, the notation $wp(ope, R)$ denotes, as in [8], the *weakest precondition* for the configuration c such that activation of *ope* is guaranteed to lead to c' satisfying the postcondition R. More formally, in our case, if $l(c) \Rightarrow wp(ope, R)$ then $l(c') \Rightarrow R$. Therefore, the weakest precondition $wp(add, R_{add})$ is the conjunction of preconditions (2) to (5).

Inspired by [8] 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 [8], 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.

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*>

If a guarded reconfiguration set is made of more than one guarded reconfiguration, they are separated by the [] operator[5]. 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

[5] As in [8], the order in which guarded reconfigurations appear is semantically irrelevant.

run is the name of a generic action representing all the running operations[6] 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 $c \stackrel{ope}{\rightarrow} c'$ 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 $\forall i \geq 0.\ \exists\ ope_i \in \mathcal{R}_{run}.(c_i \stackrel{ope_i}{\rightarrow} c_{i+1})$. 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 $reach(\mathcal{C}) = \{reach(c) \mid c \in \mathcal{C}\}$.

Proposition 1 (Preservation of consistency). *Given* $\mathcal{C}^0 \subseteq \mathcal{C}$, $\texttt{consistent}(\mathcal{C}^0)$ *implies* $\texttt{consistent}(reach(\mathcal{C}^0))$.

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 $\texttt{consistent}(c)$ and the preconditions of $add(,)$ hold on c: Then, the transition $c \stackrel{add}{\rightarrow} c'$ leads to configuration c' such that $\texttt{consistent}(c')$, i.e., that the postconditions of $add(,)$ satisfy the consistency constraints of Table 1 too; formally, $(l(c) \Rightarrow CC \wedge wp(add, R_{add})) \wedge (c \stackrel{add}{\rightarrow} c') \Rightarrow (l(c') \Rightarrow CC \wedge R_{add})$. 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));

(**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

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

$add(,)$ operation, or not. In the latter case the constraint (**CC.10**) trivially holds on c'. In the former case, since $\texttt{consistent}(c)$, 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 $ope_0, ope_1, \ldots, ope_{n-1}$ and a set of intermediate configurations $\mathcal{C}' = \{c_1, c_2, \ldots, c_{n-1}\}$[7] such that $c_0 \overset{ope_0}{\rightarrow} c_1$, $c_1 \overset{ope_1}{\rightarrow} c_2$, $\ldots$, $c_{n-1} \overset{ope_{n-1}}{\rightarrow} c$, 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., $gl_n = ope_0; ope_1; \ldots; ope_n$, 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 [8], $CC_0 = wp(gl_n, CC_{n+1}) = wp(gl_n, wp(ope_{n+1}, CC_{n+2}))$.

We can also prove (see Appendix A) that guarded reconfigurations having a non primitive 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 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 ($G \rightarrow S_0; S_1; \ldots; S_n$) 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 Implementation with *GROOVE*

This section describes how our model has been implemented within the *GROOVE* graph transformation tool [9]. This implementation is then used to experiment with our case study example.

[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., $c_1, c_3, \ldots$, would not belong to $\mathcal{C}$ and $\mathcal{C}' \not\subset \mathcal{C}$.

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 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.

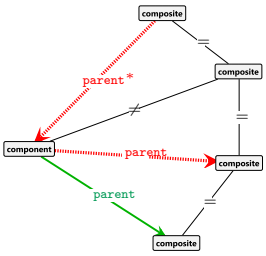
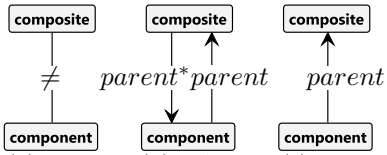


Fig. 3: *add* primitive operation in *GROOVE*

Graphs are transformed by 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 transitive relation *parent* 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 transformations 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 is displayed Fig. 4.



(a) LHS (b) NAC (c) RHS

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

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

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. 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.

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	$2^1 = 2$	$2^0 = 1$

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 5 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.

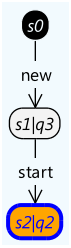


Fig. 5: Bare OS ($ic = 0$)

Similarly, for a component-based system representing a managed application server ($ic = 5$) the graph transition system is displayed in Fig. 6. In addition to the primitive reconfiguration operation used as transition labels, there is a label "chk_present_appServerPC" which represents an assertion that 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.

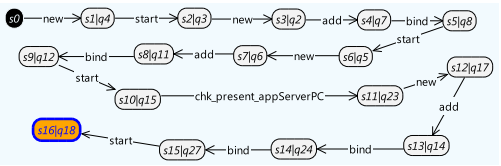


Fig. 6: Managed Application Server ($ic = 5$)

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 fixed as illustrated Fig. 7. The evolution is first performed in a deterministic way from state `s0` to `s8`. State `s16`, 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 `s23` to `s41`, to obtain a managed LAMP server ($ic = 11$). Let us notice that the evolution between `s8` and `s16` is non-deterministic. We have two shortest paths (`s8` → `s10` → `s16` and `s8` → `s11` → `s16`) that can easily be discovered using a breadth-first exploration.

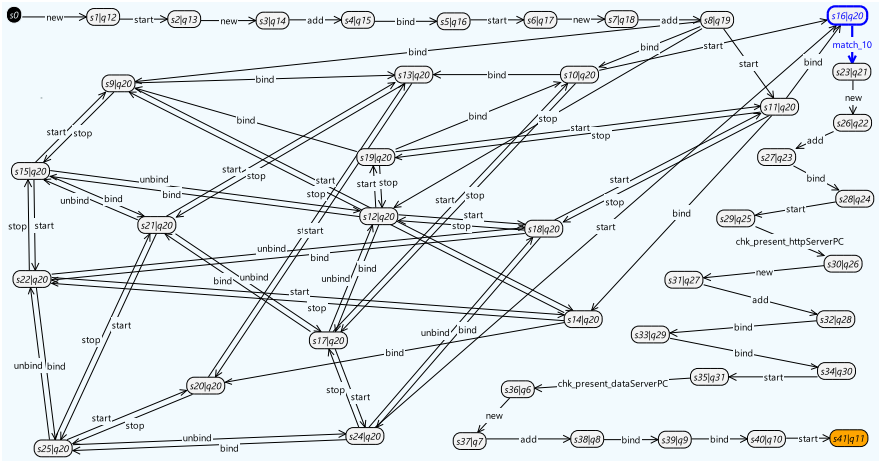


Fig. 7: Managed LAMP Server ($ic = 11$)

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. 7 has 26 states and 66 transitions. We can notice that, in our implementation, the order of primitive reconfiguration operations is fully determined[8] for $0 \leq ic \leq 5$ and $8 \leq ic \leq 9$.

Table 5: Number of states and transitions per install code

Unmanaged			**Managed**		
ic	states	transitions	ic	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

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 shows, as illustrated Fig. 7, that 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

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 simply

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

by enriching the configurations with more precise memory states and the effect of these actions upon memory.

5.1 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*. Formally, all the actions $ope \in \mathcal{R}_{run}$ are interpreted as mappings $\overline{ope}$ from $GM \times LM$ into itself. Additionaly, there are some actions specific to the implementation, $\mathcal{R}_{imp}$, as *manage* in Sect. 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 $\mathcal{I}_{\mathcal{R}_{run}} = \{\mathcal{I}, \mathcal{I}_{GROOVE}, ...\}$ 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. [7] 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 $State$[9], $v_1, \ldots, v_n \in LM$ 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 the $\mathcal{I}_{GROOVE}$ in Sect. 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 (Fig. 4a and 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 parameter, 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. In the *GROOVE* implementation, however, without this NAC $(comp_1, comp_2) \notin Parent$,

[9] Viewed as a relation.

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

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-determinstic 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* $S_{\mathcal{I}} = \langle \mathcal{C}_{\mathcal{I}}, \mathcal{C}^0_{\mathcal{I}}, \mathcal{R}_{run_{\mathcal{I}}}, \rightarrow_{\mathcal{I}}, l_{\mathcal{I}} \rangle$ *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,* $\mathcal{R}_{run_{\mathcal{I}}} = \{\overline{ope} \mid ope \in \mathcal{R}_{run} \cup \mathcal{R}_{imp}\}$, $\rightarrow_{\mathcal{I}} \subseteq \mathcal{C}_{\mathcal{I}} \times \mathcal{R}_{run_{\mathcal{I}}} \times \mathcal{C}_{\mathcal{I}}$ *is the reconfiguration relation obeying graph rules, and* $l_{\mathcal{I}} : \mathcal{C}_{\mathcal{I}} \rightarrow CP$ *is a total interpretation function.*

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 $\mathcal{R}_{run_{\mathcal{I}}} = \{\overline{ope} \mid ope \in \mathcal{R}_{run} \cup \mathcal{R}_{imp}\}$ 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. 6, 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 $c \stackrel{ope}{\Rightarrow} c'$ when there are $n, m \geq 0$ such that $c \stackrel{\tau^n ope \tau^m}{\longrightarrow} c'$.

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* $\mathcal{R}$, $(c_1, c_2) \in \sqsubseteq_\tau$ *implies whenever* $c_1 \stackrel{ope}{\Rightarrow}_1 c_1'$, *then there exists* $c_2' \in \mathcal{C}_2$ *such that* $c_2 \stackrel{ope}{\Rightarrow}_2 c_2'$ *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 $\forall c_1^0 \in \mathcal{C}_1^0 \exists c_2^0 \in \mathcal{C}_2^0 . (c_1^0, c_2^0) \in \sqsubseteq_\tau$.

Let us consider interpreted reconfiguration operations in $\mathcal{R}_{run_{\mathcal{I}}}$ 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). $S_{\mathcal{I}} \sqsubseteq_\tau S$.

Proof (sketch). We first establish, as we did above for the *add* operation, that any primitive reconfiguration operation of the implementation has stronger preconditions than its counterpart in the specification model. This way, we

can prove[11] that guarded reconfigurations composed of primitive statements, $G \to \overline{s}$, with $\overline{s} \in \mathcal{R}_{run_{\mathcal{I}}} \backslash \mathcal{R}_{imp}$ have stronger preconditions than the corresponding statement $s \in \mathcal{R}_{run}$.

Let us consider a sequence of guarded reconfigurations $G_0 \to \overline{s}_0, G_1 \to \overline{s}_1, \ldots, G_n \to \overline{s}_n$, while ignoring τ's covering operations in $\mathcal{R}_{imp}$. For all i s.t. $0 \leq i \leq n$, $\overline{s}_i$ has stronger preconditions than s_i; therefore, there exists a guard G'_i, s.t. $G_i \Rightarrow G'_i$ and $G'_i \to s_i$, as illustrated below.

$$\begin{array}{cc} S_{\mathcal{I}} & G_0 \to \overline{s}_0, G_1 \to \overline{s}_1, \ldots, G_n \to \overline{s}_n \\ & \Downarrow \qquad \Downarrow \qquad\qquad \Downarrow \\ S & G'_0 \to s_0, G'_1 \to s_1, \ldots, G'_n \to s_n \end{array}$$

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 $\overline{ope}$, which is always eventually taken, and we are done. □

This result shows that the specification model is a correct approximation of the more realistic interpreted model. As the reachability properties are compatible with $\sqsubseteq_\tau$, this leads us, consequently, to:

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

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

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

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. We show that the *GROOVE* framework can help bridge that gap. In [5] 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,

[11] Using hypothesis on weakest preconditions defined in [8].

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 [5,10,12,17]), but a truly *dynamic* reconfiguration.

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 mean 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 [17,18,19], 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 described in [5] for centralised system and [6] for decentralised or distributed systems.

6.2 Conclusion

Inspired by [8], we proposed (cf. Table 3) 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.

We also, as a practical contribution, implemented our model using the *GROOVE* graph transformation tool [9], 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 reconfiguration 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*; possibles 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 grammars 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 grammars for the application of adaptation policies at runtime.

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) 1–32
2. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Software Engineering, 1999. Proceedings of the 1999 International Conference on. (1999) 411–420
3. Dormoy, J., Kouchnarenko, O., Lanoix, A.: Runtime verification of temporal patterns for dynamic reconfigurations of components. In Arbab, F., Ölveczky, P., eds.: FACS. Volume 7253 of LNCS. Springer Berlin Heidelberg (2012) 115—132
4. Trentelman, K., Huisman, M.: Extending JML specifications with temporal logic. In: Algebraic Methodology and Software Technology. Springer (2002) 334–348
5. 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. Volume 8348 of LNCS. Springer (2014) 234–253
6. Kouchnarenko, O., Weber, J.F.: Decentralised Evaluation of Temporal Patterns over Component-based Systems at Runtime. In Lanese, I., Madelaine, E., eds.: FACS. Volume 8997 of LNCS. Springer (2015) to appear. Long version available at: http://hal.archives-ouvertes.fr/hal-01044639.
7. Lanoix, A., Dormoy, J., Kouchnarenko, O.: Combining proof and model-checking to validate reconfigurable architectures. ENTCS **279** (2011) 43–57
8. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18** (1975) 453–457
9. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. Int. J. on Software Tools for Technology Transfer **14** (2012) 15–40
10. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The fractal component model and its support in java. Softw., Pract. Exper. **36** (2006) 1257–1284
11. Schneider, S., Treharne, H.: Csp theorems for communicating b machines. Formal Asp. Comput. **17** (2005) 390–422
12. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.B.: A component-based middleware platform for reconfigurable service-oriented architectures. Software: Practice and Experience **42** (2012) 559–583
13. Dormoy, J., Kouchnarenko, O., Lanoix, A.: Using temporal logic for dynamic reconfigurations of components. In Barbosa, L., Lumpe, M., eds.: FACS. Volume 6921 of LNCS. Springer Berlin Heidelberg (2012) 200–217
14. Hamilton, A.G.: Logic for mathematicians. Cambridge University Press, Cambridge (1978)
15. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12** (1969) 576–580
16. Milner, R.: Communication and concurrency. Prentice-Hall, Inc. (1989)
17. Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V., 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) 245–255
18. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. Software Engineering, IEEE Transactions on **16** (1990) 1293–1306

19. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. Software Engineering, IEEE Transactions on **33** (2007) 856–868

A Proof of Proposition 1 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 $B'_1 \to S_1 [\!] \dots [\!] B'_n \to S_n$, with B'_i being a boolean and $Si = ope^i_0; ope^i_1; \dots; ope^i_{n_i}$, where n_i represent the number of primitive statements $(ope^i_0, ope^i_1, \dots, ope^i_{n_i})$ of the guarded list S_i, and pre^i_j (resp. pre^i_{j+1}) represents the precondition (postcondition) of ope^i_j, for $0 < i \leq n$ and $0 \leq j \leq n_i$ (resp. $0 < j \leq n_i + 1$).

Since pre^i_0 is the precondition of S_i and we suppose the configuration before the application of S_i to be consistent, we rewrite grs as $B_1 \to S_1 [\!] \dots [\!] B_n \to S_n$, with $B_i = B'_i \wedge CC \wedge pre^i_0$. 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** $B_1 \to S_1 [\!] \dots [\!] B_n \to S_n$ **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, $CC \wedge pre^i_0 \Rightarrow wp(S_i, CC \wedge pre^i_{n_i+1})$; then, by definition $B_i \Rightarrow wp(S_i, CC \wedge pre^i_{n_i+1}) \Rightarrow wp(S_i, CC)$. 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, $CC \wedge \bigwedge\limits_{i \in I_\top} pre^i_{n_i+1}$, for the alternative construct by considering that each term of the conjunction $\bigwedge\limits_{i \in I_\top} pre^i_{n_i+1}$ is part of the postcondition of a guarded list eligible for execution.

Then, $wp(IF, CC \wedge \bigwedge\limits_{j \in I_\top} pre^j_{n_j+1}) = BB \wedge \forall i \in I : B_i \Rightarrow wp(S_i, CC \wedge pre^i_{n_i+1})$ because, by definition, $\forall i \in I_\top, B_i = \top$.

Therefore:

$$\begin{aligned} BB \wedge CC \wedge \bigwedge_{i \in I_\top} pre^i_0 &\Rightarrow BB \wedge \bigwedge_{i \in I_\top} wp(S_i, CC \wedge pre^i_{n_i+1}) \\ &\Rightarrow BB \wedge (\forall i \in I : B_i \Rightarrow wp(S_i, CC \wedge pre^i_{n_i+1})) \\ &\Rightarrow wp(IF, CC \wedge \bigwedge_{j \in I_\top} pre^j_{n_j+1}) \end{aligned}$$

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

A.2 Repetitive Construct

Let DO denote **do** $B_1 \to S_1 [\!] \ldots [\!] B_n \to S_n$ **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 $l_0, l_1, \ldots, l_k$, such that, for $0 \leq j \leq k$, $1 \leq l_j \leq n$ and $S_{l_0}; S_{l_1}; \ldots; S_{l_k}$, 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 $CC \wedge pre^{l_k}_{n_{l_k}+1}$ is a valid postcondition and, since $CC \wedge pre^{l_k}_{n_{l_k}+1} \Rightarrow CC \wedge \bigvee\limits_{i \in I} pre^{i}_{n_i+1}$, we have $wp(DO, CC \wedge pre^{l_k}_{n_{l_k}+1}) \Rightarrow wp(DO, CC \wedge \bigvee\limits_{i \in I} pre^{i}_{n_i+1})$.

We established before that, for S_i being a sequence of primitive statement, $CC \wedge pre^i_0 \Rightarrow wp(S_i, CC \wedge pre^i_{n_i+1})$; then $CC \wedge pre^{l_0}_0 \Rightarrow wp(S_{l_0}; S_{l_1}; \ldots; S_{l_k}, CC \wedge pre^{l_k}_{n_{l_k}+1})$.

Therefore:

$$
\begin{aligned}
CC \wedge \bigwedge_{i \in I} pre^i_0 &\Rightarrow CC \wedge pre^{l_0}_0 \\
&\Rightarrow wp(S_{l_0}; S_{l_1}; \ldots; S_{l_k}, CC \wedge pre^{l_k}_{n_{l_k}+1}) \\
&\qquad\qquad \text{for any valid sequence } S_{l_0}; S_{l_1}; \ldots; S_{l_k} \\
&\Rightarrow wp(DO, CC \wedge pre^{l_k}_{n_{l_k}+1}) \\
&\Rightarrow wp(DO, CC \wedge \bigvee_{i \in I} pre^i_{n_i+1})
\end{aligned}
$$

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.