

Structural Reconfiguration of Systems under Behavioral Adaptation

Carlos Canal^a, Javier Cámara^b, Gwen Salaün^c

^a*Department of Computer Science, University of Málaga, Spain*

^b*Department of Informatics Engineering, University of Coimbra, Portugal*

^c*Grenoble INP, INRIA, France*

Abstract

A major asset of modern systems is to dynamically reconfigure themselves to cope with failures or component updates. Nevertheless, designing such systems with off-the-shelf components is hardly feasible: components are black-boxes that can only interact with others on compatible interfaces. Part of the problem is solved through Software Adaptation techniques, which compensate mismatches between interfaces. Our approach aims at using results of Software Adaptation in order to also provide reconfiguration capabilities to black-box components.

This paper first formalizes a framework that unifies behavioral adaptation and structural reconfiguration of components. This formalization is used for statically detecting whether it is possible to reconfigure a system. In a second part, we present five notions of reconfiguration: history-aware reconfiguration, future-aware reconfiguration, property-compliant reconfiguration, one-way reconfigurability, and full reconfigurability. For each of these notions, its relevant properties are presented, and they are illustrated on simple yet realistic examples.

Keywords: Components, dynamic reconfiguration, behavioral adaptation

1. Introduction

The success of Component-Based Software Development comes from creating complex systems by assembling smaller, simpler components. Nevertheless, building systems based on off-the-shelf components is a difficult task because these must communicate on compatible interfaces. The task becomes even more difficult when the system needs to reconfigure because in that case components must provide reconfiguration capabilities. Here, we understand by reconfiguration the capacity of changing the component behavior and/or implementation

Email addresses: canal@lcc.uma.es (Carlos Canal), jcmoreno@dei.uc.pt (Javier Cámara), gwen.salaun@inria.fr (Gwen Salaün)

at runtime [17]. For example, we are interested in upgrading or substituting a component by another one, adding new components to a running system, and so on.

Components are black-box modules of software that come with specifications of their interfaces. Therefore, we have no access to their source code, although it is possible to use tool-assisted techniques to analyze the behavior of a component assembly [5, 9]. Some applications of this analysis are used in Software Adaptation [26] to work out behavioral mismatch among component interfaces. In [20], an adaptation contract defines rules on how mismatch can be worked out and a tool generates an adaptor that orchestrates the system’s execution while compensating incompatibilities existing among interfaces.

On the contrary, there is little support to analyze whether a reconfiguration preserves certain properties. Enabling system reconfiguration requires designers to define (i) when a component can be reconfigured, (ii) which kind of reconfiguration is supported by the component, and (iii) which kind of properties are hold by reconfiguration operations; for instance, ensuring that some parts of the system can be reconfigured without system disruption. Our approach aims at providing a formal framework that helps answering these questions.

There are several related approaches in the literature. For instance, SOFA 2.0 [9] proposes *reconfiguration patterns* in order to avoid uncontrolled reconfigurations which lead to errors at runtime. This enables the addition and removal of components at runtime, passing references to components, *etc.*, under predefined structural patterns. Several other more general approaches dealing with distributed systems and software architectures [14, 15], graph transformation [1, 25] or metamodelling [13], also address reconfiguration issues. They will be discussed in Section 6.

Our goal is to reconfigure components that have not been designed with reconfiguration capabilities in mind. Moreover, we target reconfiguration of components that may be involved in an ongoing execution without stopping the system. This fits in a context where reconfiguration may be triggered at any moment and a component must be substituted at runtime.

We build on the basis that components are provided with both signature and behavioral interfaces, and their composition is described by means of an adaptation contract. The standard way for ensuring that a component can replace another one is by means of a bisimulation equivalence [24]. Thus, substituting a component requires finding a perfect match, and reconfiguration is usually limited to instances (or subtypes) of the same component. Instead, our approach aims to exploit behavioral adaptation to further allow reconfiguration. That is, we target reconfiguration scenarios in which both the former and the new component need some adaptation in order to allow substitution. Thus, bisimulation does not fit our purposes since the need for adaptation makes the components to behave differently in the configurations being considered.

This paper is structured as follows: First, Section 2 provides some background that will be used throughout the paper. Next, Section 3 introduces a client/server system that is used as running example through all the text. Then,

Section 4 provides the formal framework that unifies structural reconfiguration and behavioral adaptation. In a second part, Section 5 contains the core of our work. It presents five notions of reconfiguration compliance: (i) history-aware reconfiguration, (ii) future-aware reconfiguration, (iii) property-compliant reconfiguration, (iv) one-way reconfigurability, and (v) full reconfigurability. This section also presents proofs of properties of interest for each of the notions, and several related algorithms. These notions of compliance induce different reconfiguration scenarios that allow replacing a component by another one that may present a completely different interface, while ensuring the preservation of several interesting system properties. Each notion is illustrated on a simple yet realistic example. Finally, Section 6 presents related work on reconfiguration and behavioral adaptation, and Section 7 concludes this paper.

This article is a revised and extended version of our previous work presented in [12]:

- We propose a reconfiguration notion which preserves actions occurring in the future (*wrt.* the moment when we apply the reconfiguration). We also define a reconfiguration which is both history and future-compliant.
- We propose a less restrictive notion of reconfigurability based on the verification of temporal properties
- We give a formal characterization of all the reconfiguration notions, including theorems proving properties of interest, and (when suitable) algorithms for checking whether a reconfiguration satisfies a particular notion.

2. Background

This paper builds on our previous works on Software Adaptation, mainly [11, 10]. We recall in this section some of the concepts and definitions that are used in this paper.

2.1. Component interfaces

We assume that component interfaces are described by a signature and a protocol. The signature declares both the operations that the component provides, and those it requires from other components. A signature is represented by a set of actions L , relative to the emission and reception of messages corresponding to operation calls. An action is a tuple (M, D) where M is the operation name, and D stands for the communication direction ($!$ for emission, and $?$ for reception).

On the other hand, the protocol represents the behavior of the component *i.e.*, the order in which the operations in the signature are performed. We model the behavior of a component by means of a Labelled Transition System (LTS). The transitions in the LTS encode the actions that a component may perform in a given state.

Definition 1. [LTS]. A *Labelled Transition System* is a tuple $\langle S, s_0, L, \rightarrow \rangle$ where S is the set of states, $s_0 \in S$ is the initial state, L is the set of actions, \rightarrow is the transition relation: $\rightarrow \subseteq S \times L \times S$. We write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \rightarrow$.

The set L corresponds to the actions in the component's signature. Thus, we represent components just by the LTS that describe their behavior. Finally, we avoid non-determinism in LTSs. That is, we assume that for any $(s, \alpha, s') \in \rightarrow$, $\nexists s''$ s.t. $s'' \neq s'$ and $(s, \alpha, s'') \in \rightarrow$.

We will use traces for representing both the state of a component and the history of actions that it has performed up to a given point. Given an LTS $\langle S, s_0, L, \rightarrow \rangle$, a trace (usually denoted by σ, σ' , etc.) is a sequence $\langle \alpha_0 \dots \alpha_n \rangle$ of actions from L . ϵ represents the empty trace ($\epsilon = \langle \rangle$). We assume an operation of trace concatenation (written $\sigma \frown \sigma'$) defined in the usual way. We also assume an operation of complementary of a trace σ or a set of traces R (written $\bar{\sigma}$ and \bar{R} , respectively), defined by complementing all the actions in the trace or the set (*i.e.*, replacing $\alpha!$ by $\alpha?$ and *vice versa*). Finally, we will say that two traces σ and σ' are *disjoint* if they have no action in common, *i.e.*, if the sets of actions over which σ and σ' are defined are disjoint.

Then, we can define the traces of an LTS as the sequences of actions that can be observed according to its transition relation and starting from its initial state:

Definition 2. [Traces]. Let $p = \langle S, s_0, L, \rightarrow \rangle$ be an LTS. We define its traces $\Sigma_p = \{\epsilon\} \cup \{\sigma \mid \sigma = \alpha \frown \sigma' \text{ s.t. } \alpha \in L, (s_0, \alpha, s') \in \rightarrow, \text{ and } \sigma' \in \Sigma_{p'}\}$, where $p' = \langle S, s', L, \rightarrow \rangle$.

The definition above considers the (possibly infinite) set of traces of an LTS, including any partial trace that can be derived from it. In some situations, we will be just interested in *maximal* traces, *i.e.*, those which are not contained in any other trace.

Definition 3. [Maximal Traces]. Let p be an LTS. We define its maximal traces $\Sigma_p^* = \Sigma_p \setminus \{\sigma \mid \sigma \in \Sigma_p \text{ s.t. } \exists \sigma', \sigma' \neq \epsilon, \text{ and } \sigma \frown \sigma' \in \Sigma_p\}$ where ' \setminus ' stands for set subtraction.

Synchronization between a group of components, each one represented by an LTS, is defined by means of their synchronous product.

Definition 4. [Synchronous Product]. The synchronous product of n LTS $p_i = \langle S_i, s_{0_i}, L_i, \rightarrow_i \rangle$, $i \in \{1, \dots, n\}$, is the LTS $p_1 \parallel \dots \parallel p_n = \langle S, s_0, L, \rightarrow \rangle$ such that:

- $S = S_1 \times \dots \times S_n$,
- $s_0 = (s_{0_1}, \dots, s_{0_n})$,
- $L = L_1 \cup \{-\} \times \dots \times L_n \cup \{-\}$,

- \rightarrow is defined as follows:
 $\forall (s_1, \dots, s_n) \in S, \forall i, j \in \{1, \dots, n\}, i < j$ such that $\exists (s_i, \alpha, s'_i) \in \rightarrow_i$,
 $\exists (s_j, \bar{\alpha}, s'_j) \in \rightarrow_j$, then
 $(x_1, \dots, x_n) \in S$ and $((s_1, \dots, s_n), (l_1, \dots, l_n), (x_1, \dots, x_n)) \in \rightarrow$,
where $\forall k \in \{1, \dots, n\}$: $\begin{cases} l_k = \alpha, x_k = s'_i & \text{if } k = i \\ l_k = \bar{\alpha}, x_k = s'_j & \text{if } k = j \\ l_k = -, x_k = s_k & \text{otherwise} \end{cases}$

where the \times operator stands for the cartesian product.

The states in the product correspond to tuples of states of the components (called substates). For instance, a state (s_1, \dots, s_n) denotes that component p_1 is in state s_1 , ..., component p_n is in state s_n . Initially all components are in their initial state (*i.e.*, s_{0_i} for p_i), which means that the initial state of the product is $(s_{0_1}, \dots, s_{0_n})$. The computation of the transitions expresses that, given some composite state (s_1, \dots, s_n) in the product, there is some transition outgoing from this state iff there are two components, p_i and p_j , that may perform from states s_i and s_j in their LTS a complementary action (*i.e.*, $(s_i, \alpha, s'_i) \in \rightarrow_i$, $(s_j, \bar{\alpha}, s'_j) \in \rightarrow_j$), one sending a message and the other one receiving it), while the other components do not perform any actions (denoted $-$). The resulting target state of the transition corresponds to its same source state, except for the substates of components p_i and p_j , which are now s'_i and s'_j , respectively. Transitions in the product are labelled with actions from the components' action sets (extended with $-$), one from each component.

2.2. Adaptation contracts and adaptors

While building a new system by reusing existing components, behavioral interfaces do not always fit one another, and these interoperability issues have to be faced and worked out. Mismatch may be caused by different message names, a message without counterpart (or with several ones) in the partner, etc. The presence of mismatch results in a deadlocking execution of several components [3, 11].

Adaptors can be automatically generated based on an abstract description of how mismatch situations can be solved [11]. This is given by an *adaptation contract*. The adaptation contract is specified by a set of *correspondence vectors* (or *vectors* for short).

Definition 5. [Vector]. A correspondence vector for a set of components $\{\langle S_i, s_{0_i}, L_i, \rightarrow_i \rangle\}_{i \in \{1, \dots, n\}}$, is a tuple $\langle e_1, \dots, e_n \rangle$ with $e_i \in L_i \cup \{-\}$, $-$ meaning that a component does not participate in a synchronization.

Vectors express correspondences between messages, like bindings between ports or connectors in architectural descriptions. Each vector establishes a correspondence among actions of the different components involved in the adaptation. Each action appearing in one vector is executed by one component and the overall result corresponds to a generalized synchronization (performed in several consecutive steps) between all the components involved. A vector may involve

any number of components and does not require interactions to occur on the same names of actions. Vectors also allow representing component actions that have no counterpart. These actions will be mimicked when required, in order to make the components in the system progress. This way, our adaptation model can be applied to both closed and open systems.

Definition 6. [Adaptation Contract]. *An adaptation contract for a set of components $\{\langle S_i, s_{0_i}, L_i, \rightarrow_i \rangle\}_{i \in \{1, \dots, n\}}$, is a set of correspondence vectors defined over the action sets L_i of these components.*

From a set of LTS $P = \{p_1, \dots, p_n\}$ representing a number of components we want to adapt to each other, and an adaptation contract \mathcal{AC} , we can build an adaptor A_P (see Algorithms 1 and 3 in [11] for details) that solves the interaction mismatch among the components, taking into account the correspondences between actions described in the adaptation contract.

The adaptor is given by an LTS which, put into a non deadlock-free system, renders it deadlock-free [11]. The behavior of the adapted system is given by the synchronous product of the adaptor and the LTS of the components ($A_P \parallel p_1 \parallel \dots \parallel p_n$). In order to avoid any direct synchronization between the components being adapted, we assume that the sets of actions L_1, \dots, L_n of these components are disjoint. In order to guarantee this, for any component p_i in P we prefix its action names with the name of the component (e.g. $p_i : \alpha!$, $p_i : \beta?$). This way, all the messages exchanged will pass through the adaptor, which can be seen as a coordinator or component-in-the-middle for the components being adapted.

Example. Let us consider two components, $C1$ and $C2$. A vector $v = \langle C1 : on!, C2 : activate? \rangle$ denotes that the action $on!$ performed by component $C1$ corresponds to action $activate?$ performed by component $C2$. \square

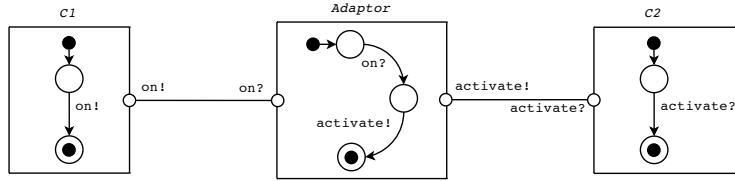


Figure 1: Components $C1$ and $C2$ connected through an adaptor.

The adaptor synchronizes with components using the same name of actions but the reversed directions, e.g., in Fig. 1 we may observe the communication between $on!$ in $C1$ and $on?$ in the adaptor. Furthermore, when a vector includes more than one action, the adaptor always starts the set of interactions formalized in the vector with the receptions (which correspond to emissions on component interfaces), and next handles the emissions.

3. Running Example

This section presents the running example used in the following sections. It consists of a client/server system in which the server may be substituted by an alternative server component. This can be needed in case of server failure, or simply for a change in the client’s context or network connection that made unreachable the original server. We assume that none of the components have been designed with reconfiguration capabilities.

The client wants to buy books and magazines as shown in its behavioral interface in Fig. 2(a). The two servers A and B have behavioral interfaces depicted in Figs. 2(c) and 3(b) respectively. Server A can sell only one book; on the other hand, server B can sell any number of books and magazines, eventually disconnecting.

Initially, the client is connected to server A ; we shall call this configuration c_A . The correspondence between actions on the client and the server is given by an adaptation contract $\mathcal{AC}_{C,A}$ (see Fig. 2(b)). Under configuration c_A the client is able to buy at most one book, but it is not allowed to buy magazines because this is not supported by server A . The latter is implicitly defined in the adaptation contract (Fig. 2(b)) as there is no vector allowing the client to perform the action *buyMagazine!*. Finally, server A does not send the acknowledgement *ack?* (see v_4 in Fig. 2(b)) expected by the client; this must also be worked out by the adaptor.

In an alternative configuration c_B the client is connected to server B whose protocol is depicted in Fig. 3(b). Similarly, the correspondence between actions is given by $\mathcal{AC}_{C,B}$ (see Fig. 3(a)). Under configuration c_B , the client can buy a number of books and magazines. In Fig. 3(a), we see that vector v_5 allows the client to buy magazines. Moreover, server B sends a different acknowledgment for each product (see v_4 and v_6 in Fig. 3(a)).

We shall study reconfiguration from c_A to c_B which substitutes A by B . It is worth noting that A and B do not have the same behavioral interfaces. Not only B provides additional functionality *wrt.* A , but also B does not have the same names for the actions (and potentially the ordering of actions may be different as well). For instance, v_1 of $\mathcal{AC}_{C,A}$ (see Fig. 2(b)) says that the *login!* action of the client relates to *user?* of server A . On the other hand, this *login!* action must be related to *connect?* of server B (see v_1 of $\mathcal{AC}_{C,B}$ in Fig. 3(a)).

Following the methodology for behavioral adaptation presented in [11], adaptors LTS can be automatically generated for configurations c_A and c_B (see adaptors $A_{C,A}$ and $A_{C,B}$ in Fig. 4). This is done by the **Compositor** tool [16]. Based on the adaptation contracts, **Compositor** automatically generates an adaptor for each configuration. Each adaptor is guaranteed by construction to orchestrate deadlock-free interactions between the client and the corresponding server, and also to fulfill the correspondences of actions described in the adaptation contract of each configuration.

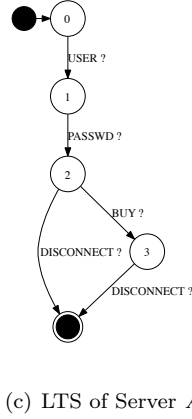
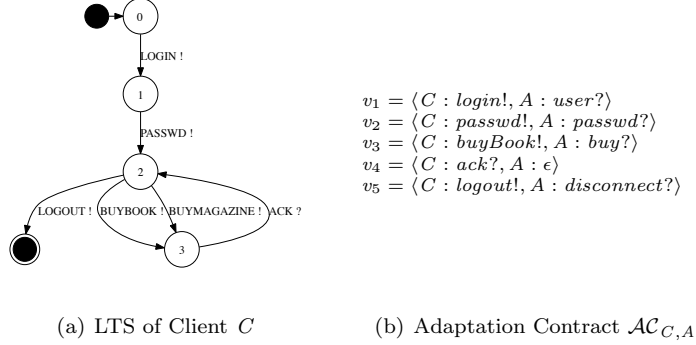


Figure 2: Configuration c_A .

4. Formal Model

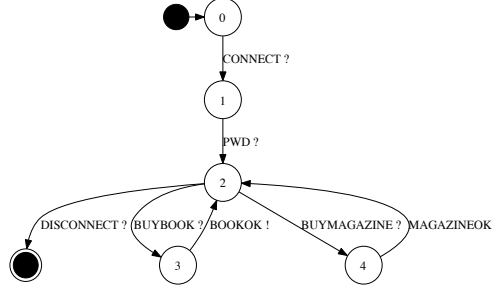
This section provides the formal model that enables both reconfiguration and behavioral adaptation. We first define a configuration as a set of components interacting by means of an adaptor, and then *reconfiguration contracts* are introduced in order to determine how a system may evolve in terms of structural changes.

4.1. Configurations

A system architecture consists of a finite number of components. The architecture may present different configurations. Each *configuration* consists of a subset of these components connected together by means of an adaptor.

Definition 7. [Configuration]. A configuration of an architecture is a tuple $\langle P, \mathcal{AC}, A_P \rangle$, where $P = \{p_1, \dots, p_n\}$ is a subset of the components of the architecture. Each component $p_i \in P$ is represented by an LTS $\langle S_i, s_{0_i}, L_i, \rightarrow_i \rangle$. \mathcal{AC}

$v_1 = \langle C : \text{login!}, B : \text{connect?} \rangle$
 $v_2 = \langle C : \text{passwd!}, B : \text{pwd?} \rangle$
 $v_3 = \langle C : \text{buyBook!}, B : \text{buyBook?} \rangle$
 $v_4 = \langle C : \text{ack?}, B : \text{bookOk!} \rangle$
 $v_5 = \langle C : \text{buyMagazine!}, B : \text{buyMagazine?} \rangle$
 $v_6 = \langle C : \text{ack?}, B : \text{magazineOk!} \rangle$
 $v_7 = \langle C : \text{logout!}, B : \text{disconnect?} \rangle$
 (a) Adaptation Contract $\mathcal{AC}_{C,B}$



(b) LTS of Server B

Figure 3: Configuration c_B .

is an adaptation contract for the components in P , $A_P = \langle S_A, s_{0_A}, L_A, \rightarrow_A \rangle$ is an adaptor, generated from \mathcal{AC} and P by means of Algorithms 1 or 3 in [11]. A configuration $\langle P, \mathcal{AC}, A_P \rangle$ is characterized by an LTS $c = \langle S_c, s_{0_c}, L_c, \rightarrow_c \rangle$ obtained by computing the synchronous product of all the components $p_i \in P$ and the adaptor A_P , i.e., $c = A_P \parallel p_1 \parallel \dots \parallel p_n$.

Let us now focus on the traces of such a configuration. From Definition 4, the actions of c are of the form $(\alpha_{A_P}, \alpha_1, \dots, \alpha_n)$, where α_{A_P} is an action from the adaptor A_P , and each α_i ($i = 1, \dots, n$) is an action from $L_i \cup \{-\}$. However, as the adaptor orchestrates the interaction between the components in a configuration, and any communication passes through it (we may recall that all the action sets L_i of the components are disjoint), each action of the configuration will consist on a synchronization between the adaptor A_P and exactly one of the components in P , i.e., $\exists! \alpha_i$ ($i = 1, \dots, n$) such that $\alpha_i \neq \{-\}$ and $\alpha_i = \overline{\alpha_{A_P}}$. Hence, we will only pay attention to the (complemented) actions of the adaptor for representing the traces of a configuration.

Definition 8. [Traces of a configuration]. Let $\langle P, \mathcal{AC}, A_P \rangle$ be a configuration. Let $c = \langle S_c, s_{0_c}, L_c, \rightarrow_c \rangle$ be the LTS characterizing it. We define its traces $\Sigma_c = \{\epsilon\} \cup \{\sigma \mid \sigma = \alpha \frown \sigma' \text{ s.t. } \exists (s_{0_c}, (\bar{\alpha}, \dots, \alpha, \dots), s') \in \rightarrow_c \text{ and } \sigma' \in \Sigma_{c'}\}$, where $\bar{\alpha}$ stands for the complementary action of α (i.e., $\alpha?$ for $\alpha!$ and vice versa), and $c' = \langle S_c, s', L_c, \rightarrow_c \rangle$.

Given a configuration $\langle P, \mathcal{AC}, A_P \rangle$, the LTS c characterizing it, and a trace

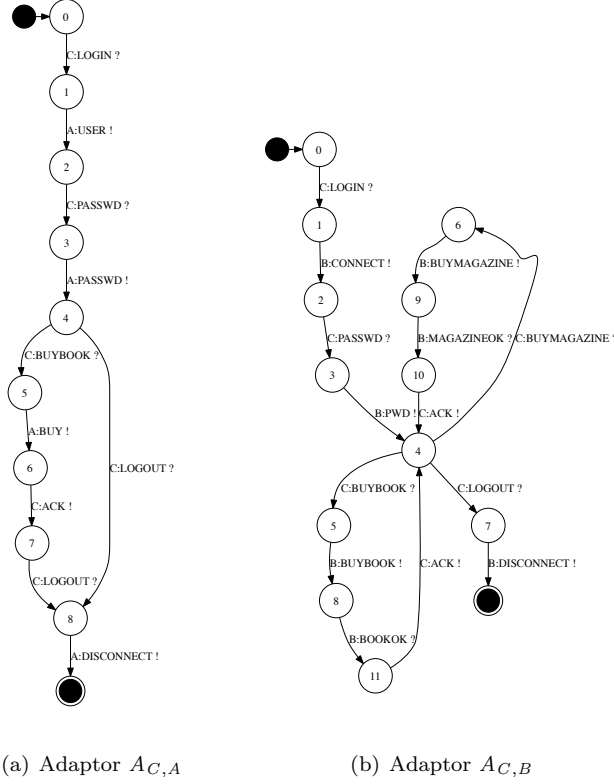


Figure 4: Adaptors for configurations c_A and c_B .

$\sigma \in \Sigma_c$, we can derive the actions performed in σ by each of the components $p \in P$ in the configuration. For that we have to project σ over the actions of p .

Definition 9. [Projection]. Let σ be a trace of a certain configuration. Let $p = \langle S, s_0, L, \rightarrow \rangle$ be a component in that configuration. The projection of σ over p (denoted $\sigma \downarrow_p$) is defined as:

$$\begin{aligned} \epsilon \downarrow_p &= \epsilon \\ (\alpha \frown \sigma') \downarrow_p &= \begin{cases} \alpha \frown \sigma' \downarrow_p & \text{if } \alpha \in L \\ \sigma' \downarrow_p & \text{if } \alpha \notin L \end{cases} \end{aligned}$$

The definition of projection can be extended to a set of traces $\{\sigma_i\}_I$. We write $\{\sigma_i\}_I \downarrow_p$ for $\{\sigma_i \downarrow_p\}_I$.

We will also need an operator over traces that hides the actions performed by a given component, leaving intact the rest of the actions in the trace.

Definition 10. [Hiding]. Let σ be a trace of a certain configuration. Let $p = \langle S, s_0, L, \rightarrow \rangle$ be a component in that configuration. The hiding of p in σ (denoted $\sigma \setminus_p$) is defined as:

$$\begin{aligned} \epsilon \setminus_p &= \epsilon \\ (\alpha \frown \sigma') \setminus_p &= \begin{cases} \alpha \frown \sigma' \setminus_p & \text{if } \alpha \notin L \\ \sigma' \setminus_p & \text{if } \alpha \in L \end{cases} \end{aligned}$$

From the definitions of projection and hiding above, some properties can be trivially inferred. In particular, we will use the following in the sequel: $(\sigma \frown \sigma') \downarrow_p = \sigma \downarrow_p \frown \sigma' \downarrow_p$, $(\sigma \frown \sigma') \setminus_p = \sigma \setminus_p \frown \sigma' \setminus_p$, and if $p \neq p'$ (and thus $L_p \cap L_{p'} = \emptyset$) then $(\sigma \setminus_p) \downarrow_{p'} = \sigma \downarrow_{p'}$.

Finally, we will also make use of an operator for trace interleaving:

Definition 11. [Trace Interleaving]. *Let σ and σ' be two traces. Their interleaving is defined as follows:*

$$\sigma \parallel \sigma' = \begin{cases} \{\sigma\} & \text{if } \sigma' = \epsilon \\ \{\sigma'\} & \text{if } \sigma = \epsilon \\ \{\sigma_1 \mid \sigma_1 = \alpha \frown (\sigma_{tail} \parallel \sigma')\} \cup \{\sigma_2 \mid \sigma_2 = \alpha' \frown (\sigma \parallel \sigma'_{tail})\} \\ & \text{with } \sigma = \alpha \frown \sigma_{tail} \text{ and } \sigma' = \alpha' \frown \sigma'_{tail}, \text{ otherwise} \end{cases}$$

Interleaving can be generalized to a set of traces $\{\sigma_i\}_I$. In that case we write $\parallel \{\sigma_i\}_I$.

Lemma 12. *Let $\{p_i = \langle S_i, s_{0_i}, L_i, \rightarrow_i \rangle\}_P$ $p_i \in P$ be a set of LTS whose actions are disjoint (i.e., $\forall p_i, p_j \in P$ ($p_i \neq p_j$), $L_i \cap L_j = \emptyset$). Let $\{\sigma_{p_i}\}_P$ $p_i \in P$ a set of traces such that each $\sigma_{p_i} \in \Sigma_{p_i}$. Then for any trace $\sigma \in \parallel \{\sigma_{p_i}\}_P$ $p_i \in P$, we have that $\forall p_i \in P$, $\sigma \downarrow_{p_i} = \sigma_{p_i}$.*

Proof. Since $\forall p_i, p_j \in P$ ($p_i \neq p_j$), $L_i \cap L_j = \emptyset$, it can be easily derived from Definitions 9 and 11. \square

4.2. Reconfiguration contracts

Replacing a configuration by another one is what we call a reconfiguration. Reconfigurations are specified in a *reconfiguration contract* which separates reconfiguration concerns from the business logic. Each configuration can be thought of as a static view of the architecture, while its dynamic view is specified by a reconfiguration contract.

Definition 13. [Reconfiguration Contract]. *Let \mathcal{C} be the set of configurations of an architecture. Let $\{\langle S_i, s_{0_i}, L_i, \rightarrow_i \rangle\}_{\mathcal{C}}$, $i \in \mathcal{C}$ be the set of LTS characterizing the configurations in \mathcal{C} . Let $\mathcal{S} = \bigcup_{\mathcal{C}} S_i$. A reconfiguration contract \mathcal{R} is a tuple $\langle \mathcal{C}, c_0, \rightarrow_{\mathcal{R}} \rangle$ where $c_0 \in \mathcal{C}$ is the initial configuration, and $\rightarrow_{\mathcal{R}} \subseteq \mathcal{C} \times \mathcal{S} \times \mathcal{C} \times \mathcal{S}$ is a set of reconfiguration operations, where $c_i : s_i \rightarrow c_j : s_j \in \mathcal{R}$ implies $c_i, c_j \in \mathcal{C}$, $s_i \in S_i$, and $s_j \in S_j$ —the states of the LTS characterizing c_i and c_j respectively. A reconfiguration operation $c_i : s_i \rightarrow c_j : s_j$ indicates that the architecture can be reconfigured from state s_i in c_i —which is called the source reconfiguration state—to state s_j in c_j —which is called the target reconfiguration state.*

Reconfiguration can take place in the middle of the execution of a configuration, and the new configuration may use a different adaptation contract. This allows the replacement of a component by another one that implements a different behavioral interface. The source reconfiguration state s_i defines when a configuration c_i can be reconfigured. On the other hand, the target reconfiguration state s_j indicates the starting state in the target configuration c_j to resume the execution. We will assume for the time being, that source and target reconfiguration states are known. In Section 5, we show how they can be obtained.

Example. In our running example, there are two configurations, c_A and c_B , where $c_A = \langle \{C, A\}, \mathcal{AC}_{C,A}, A_{C,A} \rangle$ and $c_B = \langle \{C, B\}, \mathcal{AC}_{C,B}, A_{C,B} \rangle$. The reconfiguration contract $\mathcal{R} = \langle \{c_A, c_B\}, c_A, \rightarrow_{\mathcal{R}} \rangle$ must indicate the reconfiguration states in which reconfiguration can be performed. However, as the servers A and B have different behavioral interfaces, it is not straight-forward to determine how reconfiguration can take place after the interaction between the client C and the server A has started. Therefore, the simplest reconfiguration scenario from c_A to c_B is defined at the initial states of the client and the server A . This is specified as a unique reconfiguration operation $(A_{C,A} : s_0, C : s_0, A : s_0) \rightarrow (A_{C,B} : s_0, C : s_0, B : s_0)$. In Section 5 we will study how other pairs of reconfiguration states—apart from the initial state here—can be obtained. \square

5. Contract-Aware Reconfiguration

In the preceding section, we have shown that systems can be reconfigured at the initial stage of their execution. Nevertheless, there are more interesting scenarios in which reconfiguration can take place. In this Section, we will introduce several notions of *reconfiguration compliance*, and prove some of their properties of interest. We will also show how to determine the reconfiguration states, and the actions that must be performed for achieving reconfiguration. To that purpose, Section 5.1 defines a notion of compliance that determines whether it is possible to reconfigure an architecture at an arbitrary stage paying attention to the interactions that took place in the system prior to reconfiguration. Section 5.2 explores how to define a reconfiguration compliance that is aware of future actions in the source configuration. Section 5.3 presents the less restrictive notion of property-compliant reconfiguration, which allows fine-grained control over the specification of the behavior that the architecture must preserve after reconfiguration, specified using temporal logic. Section 5.4 combines both history and future-aware reconfiguration, resulting in a definition of the conditions for one-way reconfigurability, from the source to the target configuration. Finally, Section 5.5 studies how to design a fully reconfigurable system architecture in which reconfiguration can take place back and forth between different configurations.

5.1. History-Aware Reconfiguration

First, we define the conditions for performing reconfiguration being aware of the previous history of the architecture. We call it *history-aware reconfiguration*.

Let $c = \langle P, \mathcal{AC}, A_c \rangle$ and $c' = \langle P', \mathcal{AC}', A_{c'} \rangle$ be two configurations of a given architecture. Suppose that c and c' differ in that a certain component $p_i \in P$ is replaced by a component $p_j \in P'$ (and thus the adaptor A_c generated from \mathcal{AC} for configuration c is replaced by $A_{c'}$ generated from \mathcal{AC}' for c'). Assume that the architecture is currently under configuration c and that the trace performed so far is σ_c .

Let us consider a component p such that $p \in P \cap P'$, i.e., $p \neq p_i, p \neq p_j$. This component p is not directly affected by the reconfiguration from c to c' , since it remains in c' . Thus, p should not be obliged to abort nor to rollback its current execution, represented by $\sigma_c \downarrow_p$, so it can keep on running unaware of the reconfiguration.

Therefore, for history-aware reconfiguration we have to ensure that for each component p in both the source and the target configurations, the execution trace $\sigma_c \downarrow_p$ already performed by p under configuration c is also contained in the traces of c' . Let us formalize this notion of history-aware reconfiguration with the following definition.

Definition 14. [History Compliance]. Let $\sigma_c \in \Sigma_c$ be a trace executed under a certain configuration $c = \langle P, \mathcal{AC}, A_c \rangle$. Let $c' = \langle P', \mathcal{AC}', A_{c'} \rangle$ be a configuration such that $\exists p_i \in P \exists p_j \in P', p_i \neq p_j$ and $P \setminus \{p_i\} = P' \setminus \{p_j\}$, in which a component p_i in c has been replaced by another component p_j in c' . Configuration c' is history-compliant to configuration c given σ_c (written $c' \triangleleft_{\sigma_c} c$) iff there exists $\sigma_{c'} \in \Sigma_{c'}$ such that $\forall p \in P \cap P'$ we have that $\sigma_{c'} \downarrow_p = \sigma_c \downarrow_p$.

Based on Definition 14, we will define history-aware reconfiguration operations in the reconfiguration contract of the architecture as follows. For each $\sigma_c \in \Sigma_c$ such that $c' \triangleleft_{\sigma_c} c$, assume that $\sigma_{c'} \in \Sigma_{c'}$ is the trace whose existence Definition 14 refers to. Let s_c be the state of configuration c after performing σ_c , and $s_{c'}$ the state of configuration c' after performing $\sigma_{c'}$. Then, we add $c: s_c \rightarrow c': s_{c'}$ to the reconfiguration contract of the architecture, allowing it to reconfigure from the state s_c in the configuration c to $s_{c'}$ in c' .

The following theorem proves that after performing a history-aware reconfiguration operation from c to c' all components p present in both configurations remain in the same state. Hence, they need not to abort nor to rollback their current traces $\sigma_c \downarrow_p$, and they do not require initialization in c' . Instead, they are able to go on working unaware of the reconfiguration, and their previous actions are contained in the new execution trace $\sigma_{c'}$.

Theorem 15. Let c and c' be two configurations of a given architecture, and let $\sigma_c \in \Sigma_c$ such that $c' \triangleleft_{\sigma_c} c$. Let $\sigma_{c'} \in \Sigma_{c'}$ be a trace under the conditions of Definition 14. Let $s_c = (s_{A_c}, s_1, \dots, s_i, \dots, s_n)$ and $s_{c'} = (s_{A_{c'}}, s'_1, \dots, s'_j, \dots, s'_n)$ be the states of the LTS characterizing c and c' after performing σ_c and $\sigma_{c'}$ respectively. Then, $\forall k$ s.t. $p_k \in P \cap P'$, we have $s_k = s'_k$.

Proof. It is trivial, since Definition 14 requires that $\forall p_k \in P \cap P' \sigma_{c'} \downarrow_{p_k} = \sigma_c \downarrow_{p_k}$. \square

As a result of Theorem 15, for performing the reconfiguration, we only need to initialize the new component p_j and the adaptor $A_{c'}$ in c' , in order to make them arrive to states $s_{A_{c'}}$ and s_j , respectively. For initializing the adaptor $A_{c'}$, as the traces of a configuration are the complement of those of its adaptor (see Definition 8), we only need to feed $A_{c'}$ with $\overline{\sigma_{c'}}$. On the other hand, as the actions of the components of a configuration are disjoint, $\sigma_{c'} \downarrow_{p_j}$ gives us the trace for initializing p_j .

Definition 14 above establishes the conditions for a new configuration c' being aware of the previous history of the architecture. However, it does not give us the trace $\sigma_{c'}$ that must be used for initializing the new configuration. Finding that trace is the purpose of the following theorem:

Theorem 16. *Let c and c' be two configurations of a given architecture as described in Definition 14. Let $\sigma_c \in \Sigma_c$, and let σ be a trace such that:*

- $\forall p \in P \cap P', \sigma_c \downarrow_p$ and σ are disjoint, and
- $\| (\cup_{p \in P \cap P'} \{\sigma_c \downarrow_p\} \cup \{\sigma\}) \cap \Sigma_{c'} \neq \emptyset$

then $c' \triangleleft_{\sigma_c} c$ and $\sigma \in \Sigma_{p_j}$.

Proof. From the second condition of the theorem, let us consider a trace $\sigma_{c'}$ such that $\sigma_{c'} \in \Sigma_{c'}$ and $\sigma_{c'} \in \| (\cup_{p \in P \cap P'} \{\sigma_c \downarrow_p\} \cup \{\sigma\})$. Since all the components in P' have disjoint action sets, and also σ and all the $\sigma_c \downarrow_p$ are disjoint (by the first condition), then for all $p \in P \cap P'$ from Lemma 12 we have that $\sigma_{c'} \downarrow_p = \sigma_c \downarrow_p$. Hence, $c' \triangleleft_{\sigma_c} c$. Finally, as $\sigma_{c'} \in \Sigma_{c'}$ and p_j is the only component in P' which is not in $P \cap P'$, again from Lemma 12 we have that $\sigma_{c'} \downarrow_{p_j} = \sigma$ and $\sigma \in \Sigma_{p_j}$. \square

Theorem 16 gives us a way to compute the trace $\sigma_{c'}$ that fulfills the conditions of Definition 14 and the trace σ required for initializing the component p_j . Algorithm 1 below builds the initialization traces pair $(\sigma, \sigma_{c'})$ for the component p_j and the adaptor $A_{c'}$ incrementally. At each step, the algorithm tries to make the system advance with elements from the traces in Σ (initially $\{\sigma_c \downarrow_p\}_{p \in P \cap P'}$). If this is not possible, it checks if an action in L_{p_j} can make the system advance. If there is only a single action meeting the condition, the algorithm extends the initialization traces with it and goes to the next iteration. If more than one action meet the condition, the algorithm tries to build recursively the potential remaining parts of the initialization traces, $(\sigma^*, \sigma_{c'}^*)$, starting from the current state of the adaptor (*current*), and the traces currently in Σ . If no initialization traces are found, the algorithm returns (ϵ, ϵ) . The Algorithm stops when all traces in Σ are empty (success), or the system cannot further advance with elements of Σ or L_{p_j} . Initial call to the algorithm is made as $h_traces(p_j, A_{c'}, \{\sigma_c \downarrow_p\}_{p \in P \cap P'}, s_{0_{A_{c'}}})$. It is worth mentioning that

the initialization traces pair $(\sigma, \sigma_{c'})$ may be not unique. In that case, the stop condition of the algorithm guarantees that one of them will be returned.

Algorithm 1 h_traces

Computes initialization traces σ and $\sigma_{c'}$ for $p_j \in P'$ and $A_{c'}$, respectively.

inputs Component $p_j = \langle S_{p_j}, s_{0_{p_j}}, L_{p_j}, \rightarrow_{p_j} \rangle$, Adaptor $A_{c'} = \langle S_{A_{c'}}, s_{0_{A_{c'}}}, L_{A_{c'}}, \rightarrow_{A_{c'}} \rangle$,

Traces Σ , current state for adaptor *current*

output Traces $\sigma, \sigma_{c'}$

```

1:  $(\sigma, \sigma_{c'}) := (\epsilon, \epsilon)$ 
2: while  $\exists \sigma_k \in \Sigma : \sigma_k \neq \epsilon$  do
3:   if  $\exists \sigma_l = \{p_l : \alpha_0 \ p_l : \alpha_1 \dots p_l : \alpha_n\} \in \Sigma : (current, \bar{\alpha}_0, s') \in \rightarrow_{A_{c'}}$  then
4:      $\sigma_{c'} := \sigma_{c'} \frown \{p_l : \bar{\alpha}_0\}$ 
5:      $current := s'$ 
6:      $\sigma_l := \{p_l : \alpha_1 \dots p_l : \alpha_n\}$ 
7:   else
8:      $L_{\rightarrow} = \{\alpha \in L_{p_j} \mid (current, \bar{\alpha}, s') \in \rightarrow_{A_{c'}}\}$ 
9:     if  $|L_{\rightarrow}| = 1$  then
10:       $\sigma_{c'} := \sigma_{c'} \frown \{p_j : \bar{\alpha}\}$ 
11:       $\sigma := \sigma \frown \{p_j : \alpha\}$ 
12:       $current := s'$ 
13:     else if  $(|L_{\rightarrow}| > 1) \wedge (\exists (\sigma^*, \sigma_{c'}^*) = h\_traces(p_j, A_{c'}, \Sigma, current) : (\sigma^*, \sigma_{c'}^*) \neq (\epsilon, \epsilon))$ 
14:       then
15:         return  $(\sigma \frown \sigma^*, \sigma_{c'} \frown \sigma_{c'}^*)$ 
16:       else
17:         return  $(\epsilon, \epsilon)$ 
18:       end if
19:     end if
20: end while
21: return  $(\sigma, \sigma_{c'})$ 

```

Once we have found σ'_c and σ , for performing the reconfiguration from c to c' , we will initialize $A_{c'}$ and p_j with $\bar{\sigma}_{c'}$ and σ respectively, while the rest of the components $p \in P \cap P'$ in the configuration remain in their current states (*i.e.*, each of them having performed the trace $\sigma_c \downarrow_p$ under configuration c). This way we have reconfigured the architecture from c to c' being aware of its previous history.

Example. In the running example, it is easy to find out situations for which configuration c_B is *history-compliant* to c_A ¹. For instance, let us suppose a scenario where a client connects to server A , logs in, and before disconnecting, A needs to be substituted by B . Unfortunately, A and B do not provide such reconfiguration capabilities and it is not possible to directly replace one by another without adaptation because they have different behavioral interfaces.

Let us suppose that the trace performed so far under configuration c_A is:

$\sigma_{c_A} = \langle c:\text{login! } a:\text{user? } c:\text{passwd! } a:\text{passwd? } \rangle$

Hence, the trace performed by the client C is:

$\sigma_{c_A} \downarrow_C = \langle c:\text{login! } c:\text{passwd! } \rangle$

¹In fact, it can be found that $c_B \triangleleft_{\sigma} c_A$ for any trace σ of configuration c_A .

In order to keep the client unaware of the reconfiguration, it must not abort its ongoing execution. Only if the trace $\sigma_{c_A} \downarrow_C$ is valid in the new configuration c_B , C could continue its execution as if it has been interacting with B from the very beginning.

Using Algorithm 1, from B , $\sigma_{c_A} \downarrow_C$ and the adaptor $A_{C,B}$ we can obtain a trace σ_{c_B} that makes $c_B \triangleleft_{\sigma_{c_A}} c_A$:

$$\sigma_{c_B} = \langle c:\text{login! } b:\text{connect? } c:\text{passwd! } \rangle$$

Hence, configuration c_B is history-compliant to configuration c_A given the trace σ_{c_A} . This allows us to define a history-aware reconfiguration operation from the state $s_A = (A_{C,A} : s_4, C : s_2, A : s_2)$ in c_A to the state $s_B = (A_{C,B} : s_3, C : s_2, B : s_1)$ in c_B , where state numbers refer to Figs. 2, 3, and 4.

The trace $\overline{\sigma_{c_B}}$ indicates the initialization required for the adaptor $A_{C,B}$ in the target configuration c_B . With respect to the server B , it must be initialized using the trace $\sigma_{c_B} \downarrow_B$:

$$\sigma_{c_B} \downarrow_B = \langle b:\text{connect! } \rangle$$

After these initializations all the components in the target configuration c_B are ready to resume working as if reconfiguration had never taken place. While the client was kept logged in, the original server A has been substituted at runtime by another component B with a different behavioral interface.

This example shows how the client may initially log to server A , and after a reconfiguration to c_B it is logged to server B , where differently from configuration c_A it would be able to buy several books and magazines.

On the contrary, c_A is not *history-compliant* to c_B given an arbitrary trace σ_{c_B} . Consider the case that the client has bought a magazine (it will be the same for several books) under configuration c_B , as for instance:

$$\sigma_{c_B} \downarrow_C = \langle c:\text{login! } c:\text{passwd! } c:\text{buyMagazine! } \rangle$$

this trace cannot be performed under configuration c_A because A only does not allow execution traces in which magazines are sold. In this case, Algorithm 1 would provide no results. Still, c_A is *history-compliant* to c_B for traces that do not include buying magazines nor more than one book, such as for instance:

$$\sigma'_{c_B} = \langle c:\text{login! } b:\text{connect? } c:\text{passwd! } b:\text{pwd? } c:\text{buybook! } \rangle$$

as its projection over the client C :

$$\sigma'_{c_B} \downarrow_C = \langle c:\text{login! } c:\text{passwd! } c:\text{buybook! } \rangle$$

is a trace that can be performed by the client under configuration c_A . We shall explore this scenario of reconfiguration back from c_B to c_A in Section 5.5. \square

History compliance requires that the history of the architecture remains unchanged. In such a way, the components not directly involved in a reconfiguration are able to continue working on from their current states, even if the reconfiguration was caused by a failure in the component being replaced. However, history compliance says nothing about future actions. Therefore, it may be possible to provide more (or less) functionality in the target configuration. The next section deals with the future behavior of an architecture after reconfiguration.

5.2. Future-aware reconfiguration

In the preceding section, we have studied reconfigurations which are aware of past actions in the architecture. Now, we will explore how to define a notion of compliance that takes into account future actions in the source configuration. We call this *future-aware* reconfiguration.

Suppose that $c = \langle P, \mathcal{AP}, A_c \rangle$ and $c' = \langle P', \mathcal{AP}', A_{c'} \rangle$ are two configurations of a given architecture. Suppose that the difference between c and c' is that component $p_i \in P$ is replaced by component $p_j \in P'$. Assume that the trace performed so far under configuration c is σ_c .

Let us consider again a component $p \in P \cap P'$ (*i.e.*, any of the components which are not directly affected by the reconfiguration). Differently from history-aware configuration above, we shall focus now on the future actions of these components that can be performed in both configurations.

In order to determine if c' is future-compliant with c we shall check if any possible trace of c that continues σ_c is also possible under configuration c' (conveniently hiding the actions performed by the components p_i and p_j , which are not in both configurations). This requirement would ensure that any component $p \in P \cap P'$ mentioned above could go on interacting under configuration c' as it would have done under configuration c .

Let us formalize this notion of future-aware reconfiguration by the definition of *future-compliance* below.

Definition 17. [Future Compliance]. *Let $\sigma_c \in \Sigma_c$ be a trace executed under a certain configuration $c = \langle P, \mathcal{AP}, A_c \rangle$. Let $c' = \langle P', \mathcal{AP}', A_{c'} \rangle$ be a configuration such that $\exists p_i \in P \exists p_j \in P', p_i \neq p_j$ and $P \setminus \{p_i\} = P' \setminus \{p_j\}$, in which a component p_i in c has been replaced by another component p_j in c' . Configuration c' is future-compliant to configuration c given σ_c (written $c' \triangleright_{\sigma_c} c$) iff there exists $\sigma_{c'} \in \Sigma_{c'}$ such that $\forall \sigma_c^*. \sigma_c \frown \sigma_c^* \in \Sigma_c^*, \exists \sigma_{c'}^*. \sigma_{c'} \frown \sigma_{c'}^* \in \Sigma_{c'}^*$ and $\sigma_c^* \setminus p_i = \sigma_{c'}^* \setminus p_j$.*

Based on Definition 17, we will define future-aware reconfiguration operations in the reconfiguration contract of the architecture as follows. For each $\sigma_c \in \Sigma_c$ such that $c' \triangleright_{\sigma_c} c$, assume that $\sigma_{c'} \in \Sigma_{c'}$ is the trace whose existence Definition 17 refers to. Let s_c be the state of configuration c after performing σ_c , and $s_{c'}$ the state of configuration c' after performing $\sigma_{c'}$. Then, we add $c : s_c \rightarrow c' : s_{c'}$ to the reconfiguration contract of the architecture, allowing it to reconfigure from the state s_c in the configuration c to $s_{c'}$ in c' .

Future compliance ensures certain interesting properties in the architecture. In particular, that after the point of reconfiguration, the behavior in the configuration c of the components $p \in P \cap P'$ not directly affected by the reconfiguration can be *simulated* by the new configuration c' , as shown by the following theorem:

Theorem 18. *Let c and c' be two configurations of a given architecture, and let $\sigma_c \in \Sigma_c$ such that $c' \triangleright_{\sigma_c} c$. Let $\sigma_{c'} \in \Sigma_{c'}$ be a trace under the conditions of Definition 17. Then, $\forall p \in P \cap P'$ and $\forall \sigma_c^* \text{ s.t. } \sigma_c \frown \sigma_c^* \in \Sigma_c^*$ we have that $(\sigma_{c'} \frown \sigma_c^*) \downarrow_p \in \Sigma_{c'}^* \downarrow_p$.*

Proof. From Definition 17 we have that $\forall \sigma_c^* \text{ s.t. } \sigma_c \frown \sigma_c^* \in \Sigma_c^* \exists \sigma_{c'}^* \text{ s.t. } \sigma_{c'} \frown \sigma_{c'}^* \in \Sigma_{c'}^*$. In particular, if we project the trace $\sigma_{c'} \frown \sigma_{c'}^*$ over any component $p \in P \cap P'$, and attending to Definition 9, we have that $(\sigma_{c'} \frown \sigma_{c'}^*) \downarrow_p = \sigma_{c'} \downarrow_p \frown \sigma_{c'}^* \downarrow_p \in \Sigma_{c'}^* \downarrow_p$ (*). On the other hand, from Definition 17, we also have that for σ_c^* and $\sigma_{c'}^*$, $\sigma_c^* \setminus p_i = \sigma_{c'}^* \setminus p_j$. Projecting again over all $p \in P \cap P'$, we have that $(\sigma_c^* \setminus p_i) \downarrow_p = (\sigma_{c'}^* \setminus p_j) \downarrow_p$, and as $p \neq p_j$ and $p \neq p_i$, from Definitions 9 and 10, we have that $\sigma_c^* \downarrow_p = \sigma_{c'}^* \downarrow_p$. Hence, recalling (*) we have that $\sigma_{c'} \downarrow_p \frown \sigma_c^* \downarrow_p \in \Sigma_{c'}^* \downarrow_p$, and thus $(\sigma_{c'} \frown \sigma_c^*) \downarrow_p \in \Sigma_{c'}^* \downarrow_p$. \square

From Theorem 18, all the components $p \in P \cap P'$ will not require to rollback nor to compensate the interactions performed so far under configuration c , and in fact they can continue working on as if they still were under configuration c , although the architecture has been reconfigured and the adaptation contract has changed from \mathcal{A} to \mathcal{A}' . Similarly as we have done in Section 5.1, for the new components in the target configuration c' , the adaptor $A_{P'}$ must be initialized using the trace $\overline{\sigma_{c'}}$, while p_j must be initialized using $\sigma_{c'} \downarrow_{p_j}$.

Future compliance defines the conditions for performing a reconfiguration ensuring that the future behavior of the components remaining in the target configuration is not affected by the reconfiguration: they can continue working as if reconfiguration had never taken place. However, the *past* actions in c' (represented by the trace $\sigma_{c'}$) may be completely different from those in c (represented by σ_c), and in fact it may happen that some of the past actions in the source configuration are not available in the target, which prevents us from finding a straightforward algorithm for computing the required trace σ_c' as we have done in Algorithm 1 for history-aware reconfiguration. Instead, we should explore all the states in configuration c' in order to find out if any of them satisfies the conditions of Definition 17. However, this limitation of future compliance will be overcome in Section 5.4, where we introduce one-way reconfiguration.

Example. Coming back to our running example, it is again easy to find out situations in which reconfiguration is *future-compliant*. For instance, let us suppose a scenario where a client is initially connected to server A , where it has accomplished the login phase. We will assume that the trace executed so far under configuration c_A is:

$$\sigma_{c_A} = \langle c:\text{login! } a:\text{user? } c:\text{passwd! } a:\text{passwd? } \rangle$$

Hence, hiding in σ_{c_A} the server A being replaced, we have:

$$\sigma_{c_A} \setminus A = \sigma_{c_A} \downarrow_C = \langle c:\text{login! } c:\text{passwd! } \rangle$$

Suppose that at this point, we need to change from configuration c_A to configuration c_B and that we would like to ensure future compliance in the reconfiguration. In order to do that, we need to find a trace σ_{c_B} of the target configuration such that the conditions of Definition 17 concerning possible future traces under configuration c_A are fulfilled. Since the behavior of server A is rather restrictive, these traces are basically:

$$\sigma_{c_A}^{*1} = \langle c:\text{buybook! } a:\text{buy? } c:\text{ack? } c:\text{logout! } a:\text{disconnect? } \rangle$$

and

$$\sigma_{c_A}^{*2} = \langle \text{c:logout! a:disconnect?} \rangle$$

representing the scenarios in which the client buys a book then disconnects ($\sigma_{c_A}^{*1}$), or directly disconnects ($\sigma_{c_A}^{*2}$). In both traces we hide the server A , obtaining the behavior of the client C , which is the only component in common in both configurations.

$$\sigma_{c_A}^{*1} \setminus A = \sigma_{c_A}^{*1} \downarrow_C = \langle \text{c:buybook! c:ack? c:logout!} \rangle$$

and

$$\sigma_{c_A}^{*2} \setminus A = \sigma_{c_A}^{*2} \downarrow_C = \langle \text{c:logout!} \rangle$$

Now we have to find a trace σ_{c_B} that makes configuration c_B arrive to a point in which both $\sigma_{c_A}^{*1} \setminus A$ and $\sigma_{c_A}^{*2} \setminus A$ are possible under this reconfiguration. It is not difficult to find out that for instance:

$$\sigma_{c_B} = \langle \text{c:login! b:connect? c:passwd!} \rangle$$

may be such a trace. Hence, configuration c_B is future-compliant to configuration c_A given σ_{c_A} ($c_B \triangleright_{\sigma_{c_A}} c_A$). This allows us to define a future-aware reconfiguration operation from the state $s_A = (A_{C,A} : s_4, C : s_2, A : s_2)$ in c_A to $s_B = (A_{C,B} : s_3, C : s_2, B : s_1)$ in c_B , where state numbers refer to Figs. 2, 3, and 4. Thus, the adaptor $A_{C,B}$ in the target configuration c_B has to be initialized with the trace:

$$\overline{\sigma_{c_B}} = \langle \text{c:login? b:connect! c:passwd?} \rangle$$

and the new server B in the target is initialized with:

$$\sigma_{c_B} \downarrow_B = \sigma_{c_A} \setminus A = \langle \text{b:connect?} \rangle$$

Notice that once both components are initialized as indicated, the following action in c_B will be b:pwd? , representing that the password is sent from the adaptor $A_{C,B}$ to the server B . Then, the client C will go on interacting with the server B (through the adaptor $A_{C,B}$) by any of the traces $\sigma_{c_A}^{*1} \downarrow_B$ or $\sigma_{c_A}^{*2} \downarrow_B$, as it still was under configuration c_A . Hence, the reconfiguration from c_A to c_B can be done keeping the client C unaware of it. \square

Future compliance defines the basic requirements for component replacement without affecting the future behavior of the rest of the components in a configuration. The components remaining in the target configuration will not be aware of the replacement as the new component provides at least the same behavior that the one being replaced. However, it may be difficult to find a replacement component that ensures future compliance. For this reason, the next section will explore more relaxed notions of compliance, suitable for scenarios in which it is only required that certain properties hold after reconfiguration.

5.3. Property-aware Reconfiguration

The notion of future-compliance introduced in the preceding section may be too restrictive in some situations, especially in scenarios where a component is substituted by another one with a more restricted functionality. In these cases, preserving part of the source configuration behavior (represented by a certain property ϕ) may suffice for the operation of the system. To deal with these situations we present in this section the notion of property-aware reconfiguration, which allows a finer-grained control over the specification of the behavior

that the architecture must preserve after reconfiguration. Namely, we introduce two different notions of property-aware reconfiguration: (i) existential property compliance, which requires the existence of traces under the target configuration preserving a certain property; and (ii) universal property compliance, in which a global property must be satisfied by every possible trace once reconfiguration takes place.

Properties are expressed as next-free LTL formulas over actions in execution traces, *i.e.*, atomic propositions correspond to actions and therefore we assume that the execution of an action $\alpha!$ is synonymous to the atomic proposition $\alpha!$ in a temporal logic formula. Hence, given a finite set of atomic propositions \mathcal{P} , formulas are constructed inductively as: (i) Every $\phi \in \mathcal{P}$ is a formula; (ii) given the formulas ϕ and ψ : $\phi \rightarrow \psi$, $\phi \wedge \psi$, $\phi \vee \psi$, and $\neg\phi$ are also formulas; and (iii) given the formulas ϕ and ψ : $\phi U \psi$ is also a formula. The following abbreviations are used: Eventually ($\diamond\phi = \text{TRUE } U \phi$) and Always ($\Box\phi = \neg\diamond\neg\phi$).

An interpretation of an LTL formula is an infinite word $w = x_0x_1 \dots x_n$ over $2^{\mathcal{P}}$, where at some time point $i \in N$ a proposition ϕ is true iff $\phi \in x_i$. We express as w_i the suffix of w starting at i . The semantics of next-free LTL is defined as:

Propositions For $\phi \in \mathcal{P}$, $w \models \phi$ iff $\phi \in x_0$.

Boolean operators Given the formulas ϕ and ψ :

- $w \models \neg\phi$ iff not $w \models \phi$
- $w \models \phi \wedge \psi$ iff $w \models \phi$ and $w \models \psi$
- $w \models \phi \vee \psi$ iff $w \models \phi$ or $w \models \psi$
- $w \models \phi \rightarrow \psi$ iff not ($w \models \phi$ and not $w \models \psi$)

Temporal operators $w \models \phi U \psi$ iff there exists $i \in N$ such that $w_i \models \psi$ and for all $0 \leq j < i$, $w_j \models \phi$.

We now define existential property compliance as follows:

Definition 19. [Existential Property Compliance]. Let $\sigma_c \in \Sigma_c$ be a trace executed under a certain configuration $c = \langle P, \mathcal{AP}, A_c \rangle$. Let $c' = \langle P', \mathcal{AP}', A_{c'} \rangle$ be a configuration such that $\exists p_i \in P \exists p_j \in P'$, $p_i \neq p_j$ and $P \setminus \{p_i\} = P' \setminus \{p_j\}$, in which a component p_i in c has been replaced by another component p_j in c' . Let ϕ be a next-free LTL formula built on actions of components from $P \cap P'$. Configuration c' is existentially-property-compliant with respect to configuration c given σ_c , and ϕ (written $c' \triangleright_{\sigma_c, \phi}^{\exists} c$) iff:

1. $\exists \sigma_c^* \text{ s.t. } \sigma_c \frown \sigma_c^* \in \Sigma_c^* \text{ and } \sigma_c \frown \sigma_c^* \models \phi$
2. $\exists \sigma_{c'}, \sigma_{c'}^* \text{ s.t. } \sigma_{c'} \frown \sigma_{c'}^* \in \Sigma_{c'}^* \text{ and } \sigma_c \frown \sigma_{c'}^* \models \phi$

Existential property compliance requires first that ϕ holds at least for a maximal trace (prefixed by σ_c) under the source configuration, and then that there exists at least one maximal trace in the target configuration such that

ϕ holds for the combined trace performed before and after the reconfiguration ($\sigma_c \frown \sigma_{c'}^*$). Note that the stated property ϕ refers only to actions of the components in common. In fact, if ϕ depended on actions from p_i or p_j , in general it may not hold either before or after reconfiguration, since p_i and p_j are not present in both the source and the target configurations.

As we have mentioned, existential property compliance relaxes the conditions of future compliance, since it only requires that a certain property is satisfied after reconfiguration (instead of requiring that any possible continuation trace is also possible in the target configuration). The following theorem formalizes this intuition.

Theorem 20. *Let c and c' be two configurations of a given architecture, and let $\sigma_c \in \Sigma_c$. If $\exists \sigma_c^*$ s.t. $\sigma_c \frown \sigma_c^* \in \Sigma_c^*$ and $\sigma_c \frown \sigma_c^* \models \phi$, then $c' \triangleright_{\sigma_c} c \implies c' \triangleright_{\sigma_c, \phi}^{\exists} c$.*

Proof. We have that $\exists \sigma_c^*$ s.t. $\sigma_c \frown \sigma_c^* \in \Sigma_c^*$ and $\sigma_c \frown \sigma_c^* \models \phi$. As ϕ is not built on actions from p_i , we have that $\sigma_c \frown \sigma_c^* \setminus_{p_i} \models \phi$. On the other hand, from $c' \triangleright_{\sigma_c} c$ we have that $\exists \sigma_{c'} \in \Sigma_{c'}$ such that $\forall \sigma_c^* \in \Sigma_c^*$ s.t. $\sigma_c \frown \sigma_c^* \in \Sigma_c^*$ $\exists \sigma_{c'}^* \in \Sigma_{c'}^*$ s.t. $\sigma_{c'} \frown \sigma_{c'}^* \in \Sigma_{c'}^*$ and $\sigma_c^* \setminus_{p_i} = \sigma_{c'}^* \setminus_{p_j}$. Thus, $\sigma_c \frown \sigma_{c'}^* \setminus_{p_j} \models \phi$. Finally, as ϕ is not built on actions from p_j , $\sigma_c \frown \sigma_{c'}^* \models \phi$. Hence, $c' \triangleright_{\sigma_c, \phi}^{\exists} c$. \square

Let us now consider a universally quantified version of property compliance. It is defined as follows:

Definition 21. [Universal Property Compliance]. *Let $\sigma_c \in \Sigma_c$ be a trace executed under a certain configuration $c = \langle P, \mathcal{AP}, A_c \rangle$. Let $c' = \langle P', \mathcal{AP}', A_{c'} \rangle$ be a configuration such that $\exists p_i \in P \exists p_j \in P'$, $p_i \neq p_j$ and $P \setminus \{p_i\} = P' \setminus \{p_j\}$, in which a component p_i in c has been replaced by another component p_j in c' . Let ϕ be a next-free LTL formula built on actions of components from $P \cap P'$. Configuration c' is universally-property-compliant with respect to configuration c given σ_c , and ϕ (written $c' \triangleright_{\sigma_c, \phi}^{\forall} c$) iff:*

1. $\forall \sigma_c^* \in \Sigma_c^*$ s.t. $\sigma_c \frown \sigma_c^* \in \Sigma_c^*$, we have that $\sigma_c \frown \sigma_c^* \models \phi$
2. $\exists \sigma_{c'} \in \Sigma_{c'}$ such that $\forall \sigma_{c'}^* \in \Sigma_{c'}^*$ s.t. $\sigma_{c'} \frown \sigma_{c'}^* \in \Sigma_{c'}^*$, we have that $\sigma_c \frown \sigma_{c'}^* \models \phi$

Universal property compliance firstly requires that ϕ holds for all maximal traces which continue from the current state of execution in the source configuration c . Secondly, it also requires that there exists a trace $\sigma_{c'}$ under configuration c' such that ϕ holds for all traces that combine the execution trace σ_c already performed under configuration c with any possible maximal continuation trace $\sigma_{c'}^*$ under configuration c' .

Based on Definition 19 and 21, and given a certain property ϕ we want a trace or all the traces in the architecture to satisfy, we will define property-aware reconfiguration operations in the reconfiguration contract of the architecture as follows. For each $\sigma_c \in \Sigma_c$ such that $c' \triangleright_{\sigma_c, \phi}^{\exists} c$, or $c' \triangleright_{\sigma_c, \phi}^{\forall} c$, assume that $\sigma_{c'} \in \Sigma_{c'}$ is the trace whose existence Definitions 19 or 21 refer to. Let s_c be the state of configuration c after performing σ_c , and s'_c the state of configuration

c' after performing $\sigma_{c'}$. Then, we add $c : s_c \rightarrow c' : s_{c'}$ to the reconfiguration contract of the architecture, allowing it to reconfigure from the state s_c in the configuration c to $s_{c'}$ in c' . Similarly to future-compliance, the components $p \in P \cap P'$ will not require to rollback nor to compensate the interactions performed so far under configuration c , while the considered property ϕ holds (either for one or for all continuation traces in c'), even though the architecture has been reconfigured and the adaptation contract has changed from \mathcal{AC} to \mathcal{AC}' . Like in the scenarios described in previous sections, the adaptor $A_{P'}$ must be initialized using the trace $\bar{\sigma}_{c'}$, while p_j must be initialized using $\sigma_{c'} \downarrow p_j$.

Example. Returning to our client-server running example, let us now consider that the initial configuration is c_B , and in the current scenario the client is connected to server B , after having completed the login phase. Assume that the trace executed so far is:

$$\sigma_{c_B} = \langle c:\text{login! } b:\text{connect? } c:\text{passwd! } b:\text{pwd? } \rangle$$

Under the current configuration, the client can buy a number of books and magazines. However, assume that the designer specifies that under some operation conditions it may be acceptable for the system to provide a reduced functionality where it is only possible to buy books. A property $\psi = \diamond c:\text{buyBook!}$ can be specified to determine the kind of reconfigurations which are allowed, according to the aforescribed scenario. Given the requirements, we want to determine whether c_A is existentially-property compliant with respect to property ψ from the current state of the execution ($c_A \triangleright_{\sigma_{c_B}, \psi}^{\exists} c_B$).

At this point, reconfiguring the architecture from configuration c_B to c_A restricts the behavior by allowing the client to buy only books. However, this reduced functionality still fulfills the requirements of property-aware reconfiguration, since there are potential execution traces in c_A that satisfy ψ . Let us first check that ψ holds at least for one maximal trace in c_B starting with σ_{c_B} . Obviously, in c_B there are continuations $\sigma_{c_B}^*$ of σ_{c_B} such that $\sigma_{c_B} \frown \sigma_{c_B}^* \models \psi$: all those in which there is at least an occurrence of the action $c:\text{buyBook!}$. If we consider, for instance:

$$\sigma_{c_B}^{*1} = \langle c:\text{buyBook! } b:\text{buyBook? } b:\text{bookOk! } c:\text{ack? } c:\text{logout! } b:\text{disconnect? } \rangle$$

we have that:

$$\sigma_{c_B} \frown \sigma_{c_B}^{*1} = \langle c:\text{login! } b:\text{connect? } c:\text{passwd! } b:\text{pwd? } c:\text{buyBook! } b:\text{buyBook? } b:\text{bookOk! } c:\text{ack? } c:\text{logout! } b:\text{disconnect? } \rangle \models \psi$$

Now, given σ_{c_B} , let us identify traces σ_{c_A} and $\sigma_{c_A}^{*1}$ in c_A under the conditions of Definition 19. We respectively have:

$$\sigma_{c_A} = \langle c:\text{login! } a:\text{user? } c:\text{passwd! } a:\text{passwd? } \rangle, \text{ and}$$

$$\sigma_{c_A}^{*1} = \langle c:\text{buyBook! } a:\text{buy? } c:\text{ack? } c:\text{logout! } a:\text{disconnect? } \rangle$$

such that $\sigma_{c_A} \frown \sigma_{c_A}^{*1} \in \Sigma_{c_A}^*$ and also

$$\sigma_{c_B} \frown \sigma_{c_A}^{*1} = \langle c:\text{login! } b:\text{connect? } c:\text{passwd! } b:\text{pwd? } c:\text{buyBook! } a:\text{buy? } c:\text{ack? } c:\text{logout! } a:\text{disconnect? } \rangle \models \psi$$

Within the same scenario, let us assume now that the designer wants to make sure that the client always disconnects at the end of the session. In order to guarantee this property across configurations, we define the formula

$\chi = \Box(c:\text{login!} \rightarrow \Diamond c:\text{logout!})$. In this case, we want to preserve the property in all maximal traces in the new configuration, so we need to check if c_A is universally-property compliant to c_B with respect to property χ from the current state of the execution ($c_A \triangleright_{\sigma_{c_B}, \chi}^{\forall} c_B$). Observing the example, we can determine that all maximal traces prefixed by trace σ_{c_B} satisfy the property, since the client can always disconnect the session in configuration c_B . In particular, we may identify the trace fragments:

$$\begin{aligned} \sigma_{c_B}^{*l1} &= \langle c:\text{buyBook! } b:\text{buyBook? } b:\text{bookOk! } c:\text{ack? } \rangle \\ \sigma_{c_B}^{*l2} &= \langle c:\text{buyMagazine! } b:\text{buyMagazine? } b:\text{magazineOk! } c::\text{ack? } \rangle \\ \sigma_{c_B}^{*pf} &= \langle c:\text{logout! } b:\text{disconnect? } \rangle \end{aligned}$$

So that the set of maximal traces in c_B can be given by the expression $\sigma_{c_B} \frown \{\sigma_{c_B}^{*l1} \mid \sigma_{c_B}^{*l2}\} \frown \sigma_{c_B}^{*pf}$, where $\{\dots\}$ indicates repetition and \mid indicates choice. Hence, all traces in $\Sigma_{c_B}^*$ correspond to σ_{c_B} followed by a number of combined repetitions of the traces that correspond to the two different loops in the adaptor for configuration c_B (zero or more times), and ending with the postfix trace $\sigma_{c_B}^{*pf}$, that always contains $c:\text{logout!}$. Therefore, we can guarantee the satisfaction of the first condition for universal property compliance, stating that $\forall \sigma \in \Sigma_{c_B}^*, \sigma \models \chi$.

If we now consider the target configuration c_A , we can identify the traces:

$$\begin{aligned} \sigma_{c_A}^{*1} &= \langle c:\text{buyBook! } a:\text{buy? } c:\text{ack? } c:\text{logout! } a:\text{disconnect? } \rangle \\ \sigma_{c_A}^{*2} &= \langle c:\text{logout! } a:\text{disconnect? } \rangle \end{aligned}$$

In this case, $\Sigma_{c_A}^* = \{\sigma_{c_A} \frown \sigma_{c_A}^{*1}, \sigma_{c_A} \frown \sigma_{c_A}^{*2}\}$. All traces after reconfiguration in c_A contain $c:\text{logout!}$. Hence, $\forall \sigma_{c_A} \frown \sigma_{c_A}^* \in \Sigma_{c_A}^*, \sigma_{c_B} \frown \sigma_{c_A}^* \models \phi$. This satisfies the second condition for universal property compliance, therefore we can state that $c_A \triangleright_{\sigma_{c_B}, \phi}^{\forall} c_B$. \square

Both universal and existential property compliance can be combined with different properties by the designer in order to have a fine-grained control of the reconfigurations which are allowed in the architecture. However, it is worth mentioning that the part of the behavior to be preserved has to be carefully considered when specifying properties for reconfiguration, since subtle changes in the formulas or missing terms may lead to the specification of unsatisfiable reconfigurations. The notions of property compliance defined in this section enable the designer to specify precisely which are the properties related—to the behavior of the system before and after reconfiguration—that must hold in case reconfiguration takes place.

5.4. One-way Reconfiguration

In Sections 5.1 and 5.2 we have introduced two notions of reconfigurability that are aware of either the past or the future actions under the source configuration. In this section, we will combine both notions, resulting in a definition of the conditions for one-way reconfigurability, from a source to a target configuration.

Definition 22. [Compliance]. Let $\sigma_c \in \Sigma_c$ be a trace executed under a certain configuration $c = \langle P, \mathcal{AP}, A_c \rangle$. Let $c' = \langle P', \mathcal{AP}', A_{c'} \rangle$ be a configuration such that $\exists p_i \in P \exists p_j \in P', p_i \neq p_j$ and $P \setminus \{p_i\} = P' \setminus \{p_j\}$, in which a component $p_i \in P$ has been replaced by another component $p_j \in P'$. Configuration c' is compliant to configuration c given σ_c (written $c' \triangleleft_{\sigma_c} c$) iff there exists $\sigma_{c'} \in \Sigma_{c'}$ such that $\forall p \in P \cap P'$ we have that:

- $\sigma_{c'} \downarrow_p = \sigma_c \downarrow_p$, and
- $\forall \sigma_c^* \text{ s.t. } \sigma_c \frown \sigma_c^* \in \Sigma_c^* \exists \sigma_{c'}^* \text{ s.t. } \sigma_{c'} \frown \sigma_{c'}^* \in \Sigma_{c'}^*$, and $\sigma_c^* \setminus_{p_i} = \sigma_{c'}^* \setminus_{p_j}$.

Theorem 23. Let c and c' be two configurations of a given architecture. Let $\sigma_c \in \Sigma_c$. If $c' \triangleleft_{\sigma_c} c$ then $c' \triangleleft_{\sigma_c} c$, and $c' \triangleright_{\sigma_c} c$.

Proof. The proof is immediate from Definitions 14,17 and 22. □

As a result of the theorem above, compliance ensures that given a trace $\sigma_c \in \Sigma_c$, it is possible to move from the source configuration c and to the target configuration c' , taking into account all actions (past and future) under the source configuration. In order to check compliance, Algorithm 1 can be used for finding whether there is a trace $\sigma_{c'} \in \Sigma_{c'}$ that makes $c' \triangleleft_{\sigma_c} c$. In that case, we will then check if that trace satisfies also the second condition of Definition 22.

Example. It is trivial to find that the trace:

$\sigma_{c_A} = \langle c:\text{login! } a:\text{user? } c:\text{passwd! } a:\text{passwd? } \rangle$

used in the examples in Sections 5.1 and 5.2 satisfies also Definition 22, with:

$\sigma_{c_B} = \langle c:\text{login! } b:\text{connect? } c:\text{passwd! } \rangle$

Hence, $c_B \triangleleft_{\sigma_{c_A}} c_A$. Then, from Theorems 15 and 18 reconfiguration from c_A to c_B will take place keeping the client C unaware of it. □

The notion of compliance defined in this section makes reconfiguration consistent with both the history and the future of the system when moving from the source to the target configuration. This notion will be useful for instance when we are obliged to reconfigure due to a failure in one of the components in a configuration. In the next section we will explore a notion of full compliance that deals with scenarios in which we move back and forth between two alternate configurations.

5.5. Full reconfigurability

In the previous sections we have shown several scenarios of how to reconfigure the architecture in our running example from configuration c_A to c_B , although reconfiguration from c_B to c_A is only possible on some very specific traces performed by the client, and when we allow future actions to be different under the two configurations, as in history-aware reconfiguration. We investigate here how to design a fully reconfigurable system, in which reconfiguration can take place in both directions at any moment. In our running example, consider for instance that both servers A and B were repeatedly failing, and thus we

should often switch between two (or more) configurations. In order to allow these alternating reconfiguration operations, we will constrain the behavior of the adaptors in both configurations. Let us first formalize *full compliance* by the definition below.

Definition 24. [Full Compliance]. *Let $c = \langle P, \mathcal{AP}, A_c \rangle$ and $c' = \langle P', \mathcal{AP}', A_{c'} \rangle$ be two different configurations of a given architecture such that $\exists p_i \in P \exists p_j \in P', p_i \neq p_j$ and $P \setminus \{p_i\} = P' \setminus \{p_j\}$, which differ in that component p_i in configuration c has been replaced by component p_j in c' . Configurations c' and c are fully compliant (written $c' \diamond c$) iff:*

- $\forall \sigma_c \in \Sigma_c^* \exists \sigma_{c'} \in \Sigma_{c'}^* \text{ s.t. } \forall p \in P \cap P' \sigma_c \downarrow_p = \sigma_{c'} \downarrow_p$, and
- $\forall \sigma_{c'} \in \Sigma_{c'}^* \exists \sigma_c \in \Sigma_c^* \text{ s.t. } \forall p \in P \cap P' \sigma_{c'} \downarrow_p = \sigma_c \downarrow_p$.

The definition above requires that for the components in common, their traces are the same under both configurations. In that case, for any given state in the source configuration (let us say, s_c), to which we arrive after performing a trace σ_c , we can find a counterpart reconfiguration state the target configuration (let us call it $s_{c'}$), to which we arrive after performing a trace $\sigma_{c'}$ (and *vice versa*). Therefore, we will define fully-compliant reconfiguration operations $c: s_c \rightarrow c': s_{c'}$, and $c': s_{c'} \rightarrow c: s_c$. As we will show, these reconfiguration operations back and forth c and c' allow reconfiguration of the architecture at any execution state.

Full compliance is the most restrictive definition of reconfiguration we have given so far, since it implies one-way compliance from c to c' and *vice versa* for any trace of the architecture, as shown in the following Theorem:

Theorem 25. *Let c and c' be two configurations of a given architecture. If $c' \diamond c$, then $\forall \sigma_c, \sigma_{c'} \text{ s.t. } \sigma_c \in \Sigma_c, \sigma_{c'} \in \Sigma_{c'} \text{ we have } c' \triangleleft_{\sigma_c} c$, and $c' \triangleleft_{\sigma_{c'}} c$.*

Proof. Since Definition 24 is symmetric we will just prove one of the implications, namely $c' \diamond c \implies \forall \sigma_c \in \Sigma_c c' \triangleleft_{\sigma_c} c$.

Let us consider a trace $\sigma_c \in \Sigma_c$. Since $c' \diamond c$, we have (in particular for this trace σ_c) that $\exists \sigma_{c'} \in \Sigma_{c'}$ such that $\forall p \in P \cap P' \sigma_c \downarrow_p = \sigma_{c'} \downarrow_p$, which is the first condition of the Definition 22. Let us now consider any continuation σ_c^* of σ_c such that $\sigma_c \frown \sigma_c^* \in \Sigma_c^*$. Again, since the trace is among those of configuration c , we will find a continuation $\sigma_{c'}^*$ of $\sigma_{c'}$ such that $\sigma_{c'} \frown \sigma_{c'}^* \in \Sigma_{c'}^*$ and $\forall p \in P \cap P', \sigma_c^* \downarrow_p = \sigma_{c'}^* \downarrow_p$. If we make the interleaving of all these projections, we have that $\|_{P \cap P'} (\sigma_c^* \downarrow_p) = \|_{P \cap P'} (\sigma_{c'}^* \downarrow_p)$. Since both $p_i \notin P \cap P'$ and $p_j \notin P \cap P'$ those interleavings will not be affected by hiding: $\|_{P \cap P'} (\sigma_c^* \downarrow_p) = (\|_{P \cap P'} (\sigma_c^* \downarrow_p)) \setminus_{p_i}$ and $\|_{P \cap P'} (\sigma_{c'}^* \downarrow_p) = (\|_{P \cap P'} (\sigma_{c'}^* \downarrow_p)) \setminus_{p_j}$. Finally, consider any trace contained in one of those interleavings; it is contained in the second one, too. Hence, $\sigma_c^* \setminus_{p_i} = \sigma_{c'}^* \setminus_{p_j}$, which is the second condition of the Definition 22, and $c' \triangleleft_{\sigma_c} c$. \square

Definition 24 above imposes tight conditions for two configurations being fully compliant. However, it is still possible to feature an architecture with

full reconfigurability. For that, we need to create restricted versions of the adaptors that are equivalent from the point of view of the components shared in both configurations. These restricted adaptors constrain the behavior of the components in the architecture so that it is possible to perform reconfiguration at any moment.

Definition 26. [Restricted adaptor]. Consider a configuration $c = \langle P, \mathcal{AC}, A_c \rangle$ of a given architecture. Let $p \in P$ be a component in c , and R be a set of traces in $\Sigma_c \setminus p$. Assume that $A_c = \langle S, s_0, L_{A_c}, \rightarrow_{A_c} \rangle$. The restricted adaptor $A_c^{R,p}$ is defined as: $A_c^{R,p} = \langle S, s_0, L_{A_c}, \rightarrow_{A_c^{R,p}} \rangle$, such that $\rightarrow_{A_c^{R,p}} \subseteq \rightarrow_{A_c}$ and $\Sigma_{A_c^{R,p} \setminus p} = \bar{R}$.

Definition 26 indicates how to restrict the behavior of an adaptor given a set of traces R and a component p of the configuration: some of the transitions (s, α, s') in the transition relation of the original adaptor A_c are removed when they allow an action α that would lead to a trace that is not contained in \bar{R} ² and α is not among the actions of the component p (i.e., transitions labelled with actions of p are not removed). The behavior allowed by the restricted adaptor is a subset of that of the original one. The use of restricted adaptors is shown in the following Theorem:

Theorem 27. Let $c = \langle P, \mathcal{AP}, A_c \rangle$ and $c' = \langle P', \mathcal{AP}', A_{c'} \rangle$ be two different configurations of a given architecture such that $\exists p_i \in P \exists p_j \in P', p_i \neq p_j$ and $P \setminus \{p_i\} = P' \setminus \{p_j\}$. Let $R = \Sigma_c^* \setminus p_i \cap \Sigma_{c'}^* \setminus p_j$. Let $c^{R,p_i} = \langle P, \mathcal{AP}, A_c^{R,p_i} \rangle$ and $c^{R,p_j} = \langle P', \mathcal{AP}', A_{c'}^{R,p_j} \rangle$ be the result of replacing in c and c' the adaptors A_c and $A_{c'}$ by A_c^{R,p_i} and $A_{c'}^{R,p_j}$, respectively. Then, we have that $c^{R,p_i} \diamond c^{R,p_j}$.

Proof. First, let us consider the traces in $R = \Sigma_c^* \setminus p_i \cap \Sigma_{c'}^* \setminus p_j$. We have that $R \subseteq \Sigma_c^* \setminus p_i$ and $R \subseteq \Sigma_{c'}^* \setminus p_j$ (i.e., R contains the traces that are present in both c and c' from the point of view of the components $p \in P \cap P'$, for which we hide the actions of the components p_i and p_j which are not in $P \cap P'$). If we then compute the restricted versions of the adaptors A_c^{R,p_i} and $A_{c'}^{R,p_j}$, attending to Definition 26, we have that $\Sigma_{A_c^{R,p_i} \setminus p_i} = \Sigma_{A_{c'}^{R,p_j} \setminus p_j}$. Then, as an adaptor mediates all the interactions among the components in its configuration, if we consider the configurations c^{R,p_i} and c^{R,p_j} in which A_c and $A_{c'}$ are replaced by A_c^{R,p_i} and $A_{c'}^{R,p_j}$, respectively, we have that $\Sigma_{c^{R,p_i} \setminus p_i} = \Sigma_{c^{R,p_j} \setminus p_j}$. Since $p_i, p_j \notin P \cap P'$, from that we have $\forall p \in P \cap P' \Sigma_{c^{R,p_i} \setminus p_i} \downarrow_p = \Sigma_{c^{R,p_j} \setminus p_j} \downarrow_p$ which (considering in particular the maximal traces) ensures the conditions of the Definition 24. Hence, $c^{R,p_i} \diamond c^{R,p_j}$ \square

R represents the behavior that can be performed by the components in common in both configurations. It is obtained by hiding in the maximal traces

²Note that the traces in R are negated since the actions of the adaptor are always complementary to those of the components in P .

of c (resp. c') the actions performed by the component p_i (resp. p_j) and computing the intersection of these two sets. This yields the set of maximal traces that are shared in both configurations for the components in common. If R is empty, it is not possible to build a fully reconfigurable architecture (there is no shared behavior in the configurations considered). Otherwise, using R we restrict each adaptor to this shared behavior which yields, by construction, that configurations c and c' —using the restricted adaptors A_c^{R,p_i} and $A_{c'}^{R,p_j}$, respectively—are now fully compliant.

Algorithm 2 computes the restricted adaptor for a particular configuration c with respect to a set of shared maximal traces R and a given component p' (either p_i or p_j , the components not in $P \cap P'$) in c . In particular, the algorithm begins by aggregating all traces in R into an LTS P_R that contains the behavior shared in configurations c and c' . The transition relation \rightarrow_R in this LTS is defined with respect to states of the configuration c . In order to ensure termination, the algorithm assumes that the set of shared traces R is finite. If R is infinite (that would be the case, for instance, if the LTS describing the components in both configurations contained loops, yielding infinite sets of arbitrarily long traces) there are several approaches to deal with this problem (see for instance Biermann’s algorithm [6] or Angluin’s L^* [2], which aim at synthesizing finite state machines from finite subsets of their input-output behavior). These works can be applied to our case in the first part of Algorithm 2 in order to obtain the LTS P_R from a finite subset of an infinite set of shared traces R .

Finally, the algorithm uses the information obtained in the previous step to compute the transition relation for the new adaptor $\rightarrow_{A_c^{R,p'}}$, by only including transitions allowed in the shared behavior in P_R , or involving actions of the component p' (either p_i or p_j) which is not shared between configurations (line 15). The information in the LTS characterizing configuration c is used to relate the behavior in the adaptor with the shared behavior among configurations built into P_R .

Let us now formalize the functions that we use in Algorithm 2. Function $states_G$ returns the tuple of states in all components in a configuration (except for the adaptor) associated to a given state of the adaptor A_c :

$$states_G(s_{A_c}, c = \langle S_c, s_{0_c}, L_c, \rightarrow_c \rangle) = (s_1, \dots, s_n), \text{ such that } s_c = (s_{A_c}, s_1, \dots, s_n) \in S_c$$

Function $succ_G$ returns the successor of a state of the configuration c after the execution of an action α in a component p , $p \neq p'$:

$$succ_G((s_1, \dots, s_p, \dots, s_n), \alpha) = (s_1, \dots, s'_p, \dots, s_n), \text{ such that } (s_p, \alpha, s'_p) \in \rightarrow_p$$

Example. Coming back to our running example, we can compute $\Sigma_{c_A}^* \setminus_A$ and $\Sigma_{c_B}^* \setminus_B$ by hiding in both configurations the actions corresponding to the servers A and B , respectively. These will be the traces performed by the client in each of the configurations. Their intersection R gives us the client traces that are in common in both configurations:

$$R = \left\{ \begin{array}{l} \langle c : login! \ c : passwd! \ c : logout! \rangle, \\ \langle c : login! \ c : passwd! \ c : buybook! \ c : ack? \ c : logout! \rangle \end{array} \right\}$$

Algorithm 2 *restrict_adaptor*

Computes the restricted adaptor for a configuration c with respect to a finite set R of shared traces for more than one configuration.

inputs Component $p' = \langle S_{p'}, s_{0p'}, L_{p'}, \rightarrow_{p'} \rangle$, components $p_k = \langle S_k, s_{0k}, L_k, \rightarrow_k \rangle$, *s.t.* $p_k \in P \cap P'$, Adaptor $A_c = \langle S_{A_c}, s_{0A_c}, L_{A_c}, \rightarrow_{A_c} \rangle$, Shared maximal trace set R

output Restricted adaptor $A_c^{R,p'}$

```
1:  $\rightarrow_R := \emptyset$ 
2: for all  $\sigma = \{\alpha_1 \dots \alpha_m\} \in R$  do
3:    $current := (s_{0_1}, \dots, s_{0_n})$ 
4:   for all  $\alpha_l, l \in \{1, \dots, m\}$  do
5:      $t := (current, \bar{\alpha}_l, succ_G(current, \alpha_l))$ 
6:     if  $t \notin \rightarrow_R$  then
7:        $\rightarrow_R := \rightarrow_R \cup \{t\}$ 
8:        $current := succ_G(current, \alpha_l)$ 
9:     end if
10:  end for
11: end for
12:  $P_R = \langle (s_{0_1}, \dots, s_{0_n}), S_1 \times \dots \times S_n, L_1 \cup \dots \cup L_n, \rightarrow_R \rangle$ 
13:  $\rightarrow_{A_c^{R,p'}} := \emptyset$ 
14: for all  $(q, \alpha, q') \in \rightarrow_{A_c}$  do
15:   if  $(states_G(q), \alpha, states_G(q')) \in \rightarrow_R \vee \alpha \in L_{p'}$  then
16:      $\rightarrow_{A_c^{R,p'}} := \rightarrow_{A_c^{R,p'}} \cup \{(q, \alpha, q')\}$ 
17:   end if
18: end for
19:  $S_{A_c^{R,p'}} := \{s_A \in S_{A_c} \mid \exists (q, \alpha, q') \in \rightarrow_{A_c^{R,p'}} : s_A = q \vee s_A = q'\}$ 
20:  $L_{A_c^{R,p'}} := \{\alpha_A \in L_{A_c} \mid \exists (q, \alpha, q') \in \rightarrow_{A_c^{R,p'}} : \alpha_A = \alpha\}$ 
21: return  $\langle S_{A_c^{R,p'}}, s_{0A}, L_{A_c^{R,p'}}, \rightarrow_{A_c^{R,p'}} \rangle$ 
```

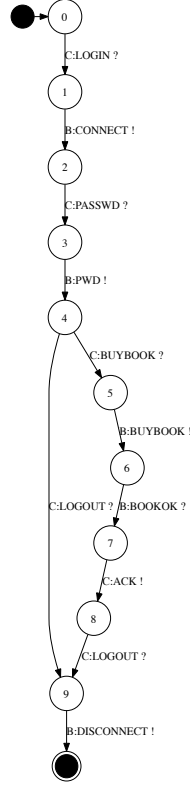


Figure 5: Restricted reconfiguration adaptor $A_{C,B}^{R,B}$.

which describes a system in which clients are allowed to buy at most one book (but no magazines). These traces are used to restrict the adaptors $A_{C,A}$ and $A_{C,B}$ to allow only the behavior considered. In fact, we find that $A_{C,A}^{R,A} \equiv A_{C,A}$, since all the client's behavior admitted by the server A is also contained in R , while $A_{C,B}^{R,B}$ (shown in Fig. 5) constrains the original adaptor $A_{C,B}$ allowing only the traces in which the client buys a book or nothing at all. In this scenario, any client trace that can be performed in one of the configurations is also feasible in the other one. This way we guarantee that server A can always be substituted by B (and B by A likewise) while keeping the client unaware of the substitution, building a system that can switch from one configuration to the other back and forth. \square

Full compliance is the most restrictive of the notions of compliance defined in this work. In fact, it implies restricting the functionality of the system in both

configurations in order to ensure Compliance in both directions. This notion would be useful for instance in scenarios in which repeated network failures oblige us to move back and forth between two alternate configurations, while we want to maintain the rest of the components in the system unaware of the repeated reconfiguration operations.

In this section we have introduced different notions of reconfiguration compliance. Each compliance states the requirements for defining reconfiguration operations that satisfy it. When all the operations in the reconfiguration contract of an architecture satisfy a given notion of compliance, we say that the architecture satisfies that particular compliance. In that case, successive reconfigurations of the architecture as defined in the reconfiguration contract ensure that the properties of the notion of compliance considered are preserved. This allows us to build reconfigurable architectures that exhibit history-awareness, future-awareness, property-awareness, or full reconfigurability.

6. Related Work

Dynamic reconfiguration [17] is not a new topic and many solutions have already been proposed in the context of distributed systems and software architectures [14, 15], graph transformation [1, 25], software adaptation [21, 20], metamodelling [13], or reconfiguration patterns [9]. On the other hand, Software Adaptation is a recent solution to build component-based systems accessed and reused through their public interfaces. Adaptation is known as the only way to compose black-box components with mismatching interfaces. However, only few works have focused so far on the reconfiguration of systems whose correct execution is ensured using adaptor components. In the rest of this section, we focus on approaches that tackled reconfiguration aspects for systems developed using adaptation techniques.

First of all, in [21], the authors present some issues raised while dynamically reconfiguring behavioral adaptors. In particular, they present an example in which a pair of reconfigurations is successively applied to an adaptor due to the upgrade of a component in which some actions have been first removed and next added. No solution is proposed in this work to automate or support the adaptor reconfiguration when some changes occur in the system.

Most of the current adaptation proposals may be considered as global, since they proceed by computing global adaptors for closed systems made up of a predefined and fixed set of components. However, this is not satisfactory when the system may evolve, with components entering or leaving it at any time, *e.g.*, for pervasive computing. To enable adaptation on such systems, an incremental approach should be considered, by which the adaptation is dynamically reconfigured depending on the components present in the system. One of the first attempts in this direction is [4], whose proposal for incremental software construction by means of refinement allows for simple signature adaptation. However, to our knowledge the only proposal addressing incremental adaptation at the behavioral level is [22, 20]. In these papers, the authors present a

solution to build step by step a system consisting of several components which need some adaptations. To do so, they propose some techniques to (i) generate an adaptor for each new component added to the system, and (ii) reconfigure the system (components and adaptors) when a component is removed.

Compared to [21, 22, 20], our goal is slightly different since we do not want to directly reconfigure adaptor behaviors, but we want to substitute both a component and its adaptor by another couple component-adaptor while preserving some properties of the system such as trace compliance.

Some recent approaches found in the literature [7, 19, 18] focus on existing programming languages and platforms, such as BPEL or SCA components, and suggest manual or at most semi-automated techniques for solving behavioral mismatch. In particular, the work presented in [18] deals with the monitoring and adaptation of BPEL services at run-time according to Quality of Services attributes. Their approach also proposes the replacement of partner services based on various strategies either syntactic or semantic. Although replaceability ideas presented in this paper are close to our reconfiguration problem, they mainly deal with QoS characteristics whereas our focus is on behavioral issues.

7. Conclusions and Future Work

This paper has presented a framework that supports the design of reconfigurable systems. The formal model defines reconfiguration as a transition from a (static) configuration to another one. Each configuration specifies a set of components interacting by means of an adaptor, and a *reconfiguration contract* defines *when* the configuration can be changed to a new one and *which* is the starting state in the new configuration in order to resume the execution.

We have integrated Software Adaptation in the framework in order to further enable reconfiguration. We have shown the conditions for a reconfiguration of the system consisting in the substitution of a component by another one that implements a different behavioral interface; this potentially includes mismatch in actions as well as in their ordering and functionality. This way, substitution ensures several interesting properties of the system, related to the components not being replaced. We build on the basis that for some cases it is possible to find sets of execution traces for different configurations which are similar from the point of view of system parts non-substituted across configurations. Thus, it is possible to simulate the execution of a system in another one where one or more components may be substituted by others with a different behavioral interface.

From a merely practical perspective, we cannot justify the re-enactment of all previous interactions of the system in order to initialize the replacement component during a reconfiguration. To tackle this issue, we believe that a notion of transaction should be defined over the LTS representing a configuration. When a configuration completes a transaction, it does not need to be re-enacted if for instance, component failure forces a reconfiguration of the system. Only transactions which have not been fully completed would need to be re-enacted within the target configuration. Transactions have not been addressed yet in our

proposal, although we consider them an interesting line of research for future work.

The reconfiguration model presented in this paper assumes a centralized adaptor. Although we believe that this is a reasonable abstraction for design, the adaptor may become a bottleneck if we consider a distributed deployment of the system. Some of our previous works (see for instance [23]) address the distribution of adaptors across different locations in a distributed setting. This approach can also be applied to the scenarios described in this paper, although we have preferred to focus on the definitions strictly related to reconfigurability, just assuming that the adaptor is centralized.

The framework that we have presented is expressive and suitable for our needs. Although in this paper we have focused on formalizing different notions of reconfiguration and their properties, as a future perspective we plan to integrate this framework within the *Fractal* component model [8]. We believe that the reconfiguration model built on *Nets* presented in [12] can be implemented in the runtime platform of Fractal in order to provide components with reconfiguration capabilities that satisfy the notions of compliance defined in this work.

Acknowledgements. This work has been partially supported by projects TIN2008-05932 funded by the Spanish Ministry of Science and Innovation (MICINN), and P07-TIC-3184 funded by the Andalusian Regional Government and the European Regional Development Fund (ERDF). We would also like to thank Antonio Cansado and Javier Cubo for their collaboration in the early stages of this work.

- [1] N. Aguirre and T. Maibaum. A Logical Basis for the Specification of Reconfigurable Component-Based Systems. In *Proc. of FASE'03*, volume 2621 of *LNCS*, pages 37–51. Springer, 2003.
- [2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
- [3] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-based Systems. In *Proc. of ICSE'07*, pages 784–787. IEEE Computer Society, 2007.
- [4] R. J. Back. Incremental Software Construction with Refinement Diagrams. Technical Report 660, Turku Center for Computer Science, 2005.
- [5] T. Barros, R. Ameur-Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed Fractal components. *Annals of Telecommunications*, 64(1):25–43, 2009.
- [6] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C-21(6):592–587, 1972.

- [7] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 27–39. Springer, 2006.
- [8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.
- [9] T. Bureš, P. Hnetyňka, and F. Plášil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.
- [10] J. Cámara, J. A. Martín, G. Salaün, J. Cubo, M. Ouederni, C. Canal, and E. Pimentel. Itaca: An integrated toolbox for the automatic composition and adaptation of web services. In *Proc. of ICSE'09*, pages 627–630. IEEE Computer Society, 2009.
- [11] C. Canal, P. Poizat, and G. Salaün. Model-Based Adaptation of Behavioural Mismatching Components. *IEEE Transactions on Software Engineering*, 34(4):546–563, 2008.
- [12] A. Cansado, G. Salaün, C. Canal, and J. Cubo. A formal framework for structural reconfiguration of components under behavioural adaptation. *Electronic Notes in Theoretical Computer Science (ENTCS) series*, 263:95–110, 2010.
- [13] A. Ketfi and N. Belkhatir. A Metamodel-Based Approach for the Dynamic Reconfiguration of Component-Based Software. In *Proc. of ICSR'04*, volume 3107 of *LNCS*, pages 264–273. Springer, 2004.
- [14] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.
- [15] J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.
- [16] R. Mateescu, P. Poizat, and G. Salaün. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. In *Proc. of ICSOC'08*, volume 5364 of *LNCS*, pages 84–99. Springer, 2008.
- [17] N. Medvidovic. ADLs and Dynamic Architecture Changes. In *SIGSOFT 96 Workshop*, pages 24–27. ACM, 1996.
- [18] O. Moser, F. Rosenberg, and S. Dustdar. Non-Intrusive Monitoring and Adaptation for WS-BPEL. In *Proc. of WWW'08*, pages 815–824, 2008.

- [19] H. R. Motahari-Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-Automated Adaptation of Service Interactions. In *Proc. of WWW'07*, pages 993–1002, 2007.
- [20] P. Poizat and G. Salaün. Adaptation of Open Component-Based Systems. In *Proc. of FMOODS'07*, volume 4468 of *LNCS*, pages 141–156. Springer, 2007.
- [21] P. Poizat, G. Salaün, and M. Tivoli. On Dynamic Reconfiguration of Behavioural Adaptation. In *Proc. of WCAT'06*, pages 61–69, 2006.
- [22] P. Poizat, G. Salaün, and M. Tivoli. An Adaptation-based Approach to Incrementally Build Component Systems. *Electronic Notes in Theoretical Computer Science (ENTCS) series*, 182:39–55, 2007.
- [23] G. Salaün. Generation of service wrapper protocols from choreography specifications. In A. Cerone and S. Gruner, editors, *SEFM*, pages 313–322. IEEE Computer Society, 2008.
- [24] I. Černá, P. Vařeková, and B. Zimmerova. Component substitutability via equivalencies of component-interaction automata. *Electronic Notes in Theoretical Computer Science (ENTCS) series*, 182:39–55, 2007.
- [25] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proc. of ESEC / SIGSOFT FSE 2001*, pages 21–32. ACM, 2001.
- [26] D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.