Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 263 (2010) 95–110
www.elsevier.com/locate/entcs

A Formal Framework for Structural Reconfiguration of Components under Behavioural Adaptation

Antonio Cansado, Carlos Canal, Gwen Salaün, and Javier Cubo

Department of Computer Science, University of Málaga
Campus de Teatinos, 29071, Málaga, Spain
Emails: {acansado,canal,salaun,cubo}@lcc.uma.es

Abstract

A major asset of modern systems is to dynamically reconfigure systems 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 compensates 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 provides two contributions: (i) a formal framework that unifies behavioural adaptation and structural reconfiguration of components; this is used for statically reasoning whether it is possible to reconfigure a system. And (ii), two cases of reconfiguration in a client/server system in which the server is substituted by another one with a different behavioural interface, and the system keeps on working transparently from the client's point of view.

Keywords: Components, reconfiguration, behavioural adaptation, formal methods

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 difficult because components must communicate on compatible interfaces. It is even more difficult when the system must be able to reconfigure because components must provide reconfiguration capabilities. Here, we understand by reconfiguration the capacity of changing the component behaviour and/or component implementation at runtime [15]. For example, we are interested in upgrading a component, substituting a component by another, adding new components to a running system, and so on.

In general, components are black-box modules of software that come with behavioural specifications of their interfaces. Therefore, there is no access to the source

doi:10.1016/j.entcs.2010.05.006

code of components, but it is possible to use tool-assisted techniques to analyse the behaviour of a component assembly [5,8]. Some applications of this analysis are used in Software Adaptation [24] to work out behavioural mismatches in the components' interfaces. In [19], an adaptation contract defines rules on how mismatches can be worked out and a tool generates an adaptor that orchestrates the system execution and compensates incompatibilities existing in interfaces.

On the other hand, there is little support to analyse whether a reconfiguration is safe. 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 the reconfiguration holds. For example, 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. SOFA 2.0 [8] proposes *reconfiguration patterns* in order to avoid uncontrolled reconfigurations which lead to errors at runtime. This allows adding and removing components at runtime, passing references to components, *etc.*, under predefined structural patterns. There are other more general approaches that deal with distributed systems and software architectures [12,13], graph transformation [1,23] or metamodelling [11].

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 transaction 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.

Since substituting a component usually requires finding a perfect match, reconfiguration is usually limited to instances (or subtypes) of the same component. Our approach is to exploit behavioural 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. We build on the basis that components are provided with behavioural interfaces and the composition with adaptation contracts. Then, we show that in some cases the information found in the adaptation contract can be used to endow black-box components with reconfiguration capabilities.

This paper's contributions are: (i) we present a formal model that includes behavioural adaptation and structural reconfiguration of components. With this framework it is possible to verify properties of the complete system, including those involving reconfiguration of a component by another one with a different behavioural interface. And (ii), we present two examples of reconfiguration in which a server is substituted by another one with a different behavioural interfaces. We also show how can we build a fault-tolerant system by constraining the system's behaviour.

This paper is structured as follows: Firstly, Section 2 provides some background on formalisms that will be used throughout the paper. Then Section 3 introduces a client/server system that is used as running example. Section 4 provides the formal model that allows structural reconfiguration and behavioural adaptation. Section 5

provides applications of our approach for designing reconfigurable systems based on (non-reconfigurable) black-box components. Then, Section 6 presents related works on reconfiguration and behavioural adaptation and Section 7 concludes this paper.

2 Background

This work builds on previous work on hierarchical behavioural models [5] and Software Adaptation [9]. We recall in this section the main definitions that are used in this paper.

2.1 Networks of Synchronised Automata

We assume that component interfaces are equipped both with a signature (set of required and provided operations), and a protocol. We model the behaviour of a component as a Labelled Transition System (LTS). The LTS transitions encode the actions that a component can perform in a given state. In the following definitions, we frequently use indexed vectors: we note $\tilde{x}_I$ the vector $\langle ..., x_i, ... \rangle$ with $i \in I$, where I is a countable set.

Definition 2.1 [LTS]. A *Labelled Transition System (LTS)* 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 labels, $\rightarrow$ is the set of transitions : $\rightarrow \subseteq S \times L \times S$. We write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \rightarrow$.

Communication between components are represented using *actions* relative to the emission and reception of messages corresponding to operation calls, or internal actions performed by a component. Therefore, in our model, a *label* is either the internal action τ or a tuple (M, D) where M is the message name and D stands for the communication direction ($!$ for emission, and $?$ for reception).

Definition 2.2 [Network of LTSs]. Let Act be an action set. A *Net* is a tuple $\langle A_G, J, \tilde{O}_J, T \rangle$ where $A_G \subseteq Act$ is a set of global actions, J is a countable set of argument indexes, each index $j \in J$ is called a *hole*. O_j is the sort of the hole j with $O_j \subset Act$. The transducer T is an LTS $(S_T, s_{0\,T}, L_T, \rightarrow_T)$, and $L_T = \{ \overrightarrow{v} = \langle a_g.\tilde{\alpha}_I \rangle,\ a_g \in A_G,\ I \subseteq J \wedge \forall\, i \in I, \alpha_i \in O_i \}$

Synchronisation between components is specified using *Nets* [5]. *Nets* are inspired by *synchronisation vectors* [2], that we use to synchronise a number of processes and can describe dynamic configurations of processes, in which the possible synchronisations change with the state of the Net; they are *transducers* [18].

A transducer in the Net is encoded as an LTS whose labels are synchronisation vectors ($\overrightarrow{v}$), each one describing a particular synchronisation between the actions ($\tilde{\alpha}_I$) of different argument processes; the result of this synchronisation is seen as a global action a_g. Each state of the transducer T corresponds to a given configuration of the network in which a given set of synchronisations is possible. Some of those synchronisations can trigger a change of state in the transducer leading to a new configuration of the network; that is, it encodes a dynamic change on the configuration of the system.

Definition 2.3 [Sort]. The *Sort* is the set of actions that can be observed from outside the automaton. It is determined by its top-level node, L for an LTS, and A_G for a Net:

$$Sort(\langle S, s_0, L, \rightarrow\rangle) = L \qquad Sort(\langle A_G, J, \tilde{O}_J, T\rangle) = A_G$$

A Net is a generalised parallel operator. Complex systems are built by combining LTSs in a hierarchical manner using Nets at each level. There is a natural typing compatibility constraint for this construction, in term of the sorts of formal and actual parameters. The standard compatibility relation is Sort inclusion: a system *Sys* can be used as an actual argument of a Net at position j only if it agrees with the sort of the hole O_j, *i.e.* $Sort(Sys) \subseteq O_j$.

2.2 Specification of Adaptation Contracts

While building a new system by reusing existing components, behavioural interfaces do not always fit one another, and these interoperability issues have to be faced and worked out. Mismatches 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 [3,10].

Adaptors can be automatically generated based on an abstract description of how mismatch situations can be solved. This is given by an *adaptation contract*.

In this paper, the adaptation contract $\mathcal{AC}$ between components is specified using *vectors* [9]. Each action appearing in one vector is executed by one component and the overall result corresponds to a synchronisation between all the involved components. A vector may involve any number of components and does not require interactions to occur on the same names of actions. Moreover, a vector may synchronise actions performed by sub-processes in a hierarchical fashion.

For example, 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 $C2$. This does not mean that both actions have to take place simultaneously, nor one action just after the other; the adaptor will take into account their respective behaviour as specified in their LTS, accommodating the reception and sending of actions to the points in which the components are able to perform them (Fig. 1).

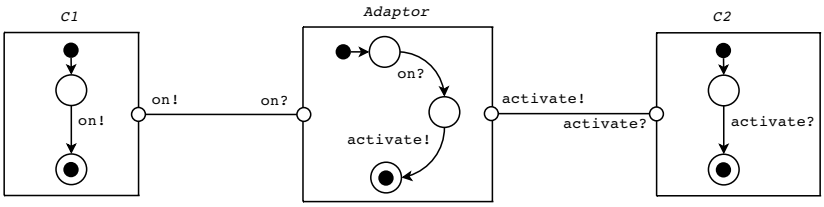


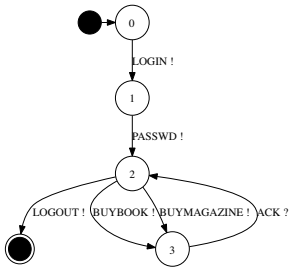
Fig. 1. Components $C1$ and $C2$ connected through an adaptor.

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 suppose none of the components have been designed with reconfiguration capabilities.

The client wants to buy books and magazines as shown in its behavioural interface in Fig. 2(a). The two servers A and B have behavioural interfaces depicted in Figs. 2(c) and 3(b) respectively. Server A can sell only one book per transaction [1]; on the other hand, server B can sell a bounded number of books and magazines.

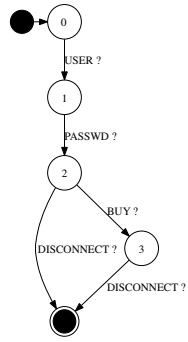
Initially, the client is connected to server A; we shall call this configuration c_A. The client and the server agree on an adaptation contract $\mathcal{AC}_{C,A}$ (see Fig. 2(b)). Naturally, under configuration c_A the client can only buy at most one book in each transaction but it is not allowed to buy magazines because this is not supported by server A. The latter is implicitly defined in the adaptation contract (Fig. 2(b)) because 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.



(a) LTS of Client C

$v_1 = \langle C : login!, A : user? \rangle$
$v_2 = \langle C : passwd!, A : passwd? \rangle$
$v_3 = \langle C : buyBook!, A : buy? \rangle$
$v_4 = \langle C : ack?, A : \epsilon \rangle$
$v_5 = \langle C : logout!, A : disconnect? \rangle$

(b) Adaptation Contract $\mathcal{AC}_{C,A}$



(c) LTS of Server A

Fig. 2. Configuration c_A.

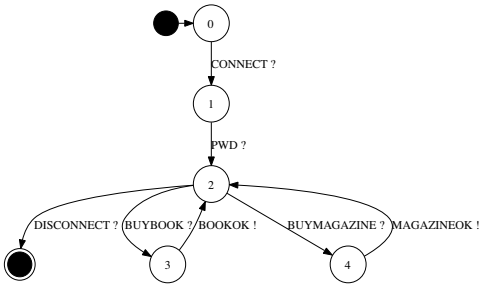
In an alternative configuration c_B the client is connected to server B whose protocol is depicted in Fig. 3(b). Similarly, the client and the server agree on an adaptation contract $\mathcal{AC}_{C,B}$ (see Fig. 3(a)). Under configuration c_B, the client can buy a bounded number of books and magazines.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

[1] A transaction starts in the LTS's initial state and ends in the final one. The definition of LTS in Section 2.1 does not include final states, though one can understand as final state a state with no outgoing transitions.

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 behavioural interfaces. Not only B provides additional functionality *w.r.t.* 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)).

$v_1 = \langle C : login!, B : connect? \rangle$
$v_2 = \langle C : passwd!, B : pwd? \rangle$
$v_3 = \langle C : buyBook!, B : buyBook? \rangle$
$v_4 = \langle C : ack?, B : bookOk! \rangle$
$v_5 = \langle C : buyMagazine!, B : buyMagazine? \rangle$
$v_6 = \langle C : ack?, B : magazineOk! \rangle$
$v_7 = \langle C : logout!, B : disconnect? \rangle$

(a) Adaptation Contract $\mathcal{AC}_{C,B}$



(b) LTS of Server B

Fig. 3. Configuration c_B.

4 Formal Model

This section provides the formal model that enables reconfiguration and behavioural adaptation. We define a *reconfiguration contract* to determine how the system may evolve in terms of structural changes. Then, we provide a formal model based on *Nets* for this kind of systems.

4.1 Reconfiguration Contract

A system architecture consists of a finite number of components. Each *configuration* is a subset of these components connected together by means of adaptation contracts.

Definition 4.1 [Configuration]. A configuration $c = \langle P, \mathcal{AC}, S^\star \rangle$ is a static structural configuration of a system. P is an indexed set of components. $\mathcal{AC}$ is an adaptation contract of components in P. $S^\star$ is a set of *reconfiguration states* defined upon P; these are states in which reconfiguration is allowed.

Changing a configuration by another is what we call a reconfiguration. This is specified in a *reconfiguration contract* which separates reconfiguration concerns from the business logic. Each configuration can be thought of a static system and the "dynamic" part is specified by a reconfiguration operation. The selection of reconfiguration states is left out-of-scope in this paper. We assume here that they

are given by the designer, though we show how some of them can be obtained in our examples.

Definition 4.2 [Reconfiguration Contract]. A reconfiguration contract $\mathcal{R} = \langle C, c_0, \rightarrow_{\mathcal{R}} \rangle$ is defined as:

C is a set of static configurations, where $c_i \in C$, $i \in \{0..n\}$, is a static configuration. $c_0 \in C$ is the initial configuration. $\rightarrow_{\mathcal{R}} \subseteq C \times R_{op} \times C$ is a set of reconfiguration operations, with reconfiguration operation $R_{op} \subseteq S_i^\star \times S_j^\star$, $S_i^\star \in c_i, S_j^\star \in c_j$.

From the definition above, reconfiguration can take place in the middle of a transaction, and the new configuration may have a new adaptation contract. This allows reconfiguring a component by another one that implements a different behavioural interface. Nevertheless, for guaranteeing consistency this can only happen at predefined states. A state of the source configuration ($s_i^\star$) defines when a configuration can be reconfigured. On the other hand, a state of the target configuration ($s_j^\star$) says what is the starting state in the target configuration to resume the execution.

Example

In the running example, there are two configurations:
$c_A = \langle \{C, A\}, \mathcal{AC}_{C,A}, S_A^\star \rangle$ and $c_B = \langle \{C, B\}, \mathcal{AC}_{C,B}, S_B^\star \rangle$.

The reconfiguration contract $\mathcal{R} = \langle C, c_A, \rightarrow_{\mathcal{R}} \rangle$ is given by:
$C = \{c_A, c_B\}$, and $\rightarrow_{\mathcal{R}} = \{c_A \xrightarrow{r} c_B\}$, with $r = (s_A^\star, s_B^\star)$.

r must specify the pairs of *reconfiguration states* on which reconfiguration can be performed. Since both servers have different behavioural interfaces, it is not straight-forward to determine how reconfiguration can take place after a transaction between the client and the server has started. Therefore, in the simplest scenario reconfiguration from c_A to c_B is only allowed at the initial states of the client and the server. This is specified as a unique reconfiguration state $s_i^\star \in S_i^\star$, $i \in \{A, B\}$ for each configuration; more precisely, $s_A^\star = \{C : s_0, A : s_0\}$ and $s_B^\star = \{C : s_0, B : s_0\}$. In Section 5 we will study how other pairs of reconfiguration states —apart from the initial states here— can be obtained.

4.2 Building Verifyable Systems

This section shows how to build Nets for the reconfiguration contract above. We have previously defined the system and now we generate a behavioural model of the complete system that can be fed into model-checking tools. There are two benefits in this approach.

Firstly, is it easier to verify properties related to reconfiguration if the complete system is modelled. In our running example, we can prove that the client may only buy magazines if the system is reconfigured towards a configuration c_B.

Secondly, a *Net* is close enough to the structure of a program that it should be possible to implement the *Net* in a component model framework such as *Fractal* [7]. This would provide us, at runtime, a component system with predictable behaviour under reconfiguration.

As the adaptor represents the interactions between the adapted components, we will use states of the adaptor to identify states in which reconfiguration will be applied. Let A_P be the adaptor LTS generated by $\mathcal{AC}$ [10], we add a reconfiguration action $r_{s_i^\star}?$ for each $s_i^\star \in S^\star$. This action leaves the adaptor in same state, defined as the state in which all components are in the state given by $s_i^\star$.

Definition 4.3 [Network of a Configuration]. A configuration $c = \langle P, \mathcal{AC}, S^\star \rangle$, defines a Net $c_{Net} = \langle A_G, J, \tilde{O}_J, T \rangle$, as:

Let A_P be the adaptor LTS generated by $\mathcal{AC}$, with a reconfiguration action $r_{s_i^\star}?$ for each $s_i^\star \in S^\star$. Each $a! \in Sort(p_i)$ defines an action $v = \{p_i : a!, A_P : a?\} \in A_G$. Each $a? \in Sort(p_i)$ defines an action $v = \{p_i : a?, A_P : a!\} \in A_G$. Each reconfiguration action $r_{s_i^\star}?$ defines an action $v = \{A_P : r_{s_i^\star}?\} \in A_G$. Each process p_i and A_P is an argument of J, with $Sort(p_i) \subseteq O_{p_i}$ and $Sort(A_P) \subseteq O_{A_P}$.

The Net transducer T is defined as: a unique state $s_T = s_{0_T} \in S_T$; and a transition $s_T \xrightarrow{v} s_T$ for each $v \in A_G$.

In the definition above, a *Net* closely represents a configuration; the root of the *Net*'s tree is the synchronisation operator given by the Net's transducer, and the *Net*'s leaves are the components p_i and A_P. The transducer actions are synchronisation vectors that relate actions between p_i and the adaptor; note that due to the adaptation process, these actions are exact dual operations. The transducer has additional transitions taking care of reconfiguration capabilities, though they are not used within a configuration.

We construct a system that allows reconfiguration between *configuration*s based on the *reconfiguration contract* and the construction of a network for a *configuration*.

Definition 4.4 [Network of Reconfiguration Contract]. A $\mathcal{R} = \langle C, c_A, \rightarrow_{\mathcal{R}} \rangle$ defines a Net $\mathcal{R}_{Net} = \langle A_G, J, \tilde{O}_J, T \rangle$, with:

Each configuration $c_i \in C$ defines a Net $c_{i_{Net}} \in J$. Each reconfiguration operation r defines a process P_r used as argument of J. This process will be in charge of initialising the target configuration. The global actions A_G are defined upon the union of the global actions of each configuration, and the reconfiguration transactions. Formally, $A_G = [\bigcup_{\forall c_i \in C} Sort(c_{i_{Net}})] \cup \{r : c_i \xrightarrow{r} c_j \in \rightarrow_{\mathcal{R}}, i \neq j\}$.

The sorts of each configuration $c_{i_{Net}}$ define $\tilde{O}_J$, *i.e.* $Sort(c_{i_{Net}}) = O_i$. The Net transducer $T{=}(S_T, s_{0_T}, L_T, \rightarrow_T)$ orchestrates the system execution. Each configuration *Net* $c_{i_{Net}}$ is represented by a state $s_i \in S_T$.

Actions within a configuration are transitions over the same configuration state, *i.e.*, there is a transition $s_i \xrightarrow{a_g} s_i$ for each $a_g \in Sort(c_{i_{Net}}) \setminus \bigcup r$.

For each reconfiguration operation $c_i \xrightarrow{r} c_j$ we add a reconfiguration state s_r in the transducer (see Fig. 4). In this state, only actions within P_r can be performed, *i.e.*, $s_r \xrightarrow{r_\alpha} s_r$ where r_α are actions in P_r. The reconfiguration starts with an action $r_{start} = r_{s_i^\star}?$ that changes the transducer state from s_i to s_r. Reconfiguration actions are performed (r_α), and finally the system changes to a configuration c_j in a state defined by r by performing the action $r_{end} = r_{s_j^\star}?$.

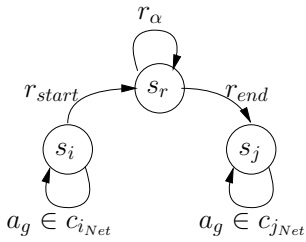


Fig. 4. Transducer representing a reconfiguration from configuration c_i to configuration c_j.

The reconfiguration contract is mapped into a Net $\mathcal{R}_{Net}$. The transducer of $\mathcal{R}_{Net}$ is an LTS that has each of the configurations as states, and transitions representing either a reconfiguration r or the allowance of a (non-reconfiguring) action a_g within a configuration. Additionally, each of the configurations c_i is mapped into a Net $c_{i_{Net}}$ that is used as an argument of the $\mathcal{R}_{Net}$'s holes. Hence, we build a tree of processes in which the root is $\mathcal{R}_{Net}$, the internal nodes are configurations, and leaves are LTSs of the components' behaviour.

Reconfigurations operations eventually move the system from a configuration to another (see Fig. 4). The role of the state s_r is to halt system execution and initialise the new configuration so that it starts at the target reconfiguration state.

Example

Back to the running example, the first step in creating the *Net* is to generate the adaptors $A_{C,A}$ (Fig. 5(a)) and $A_{C,B}$ (Fig. 5(b)) in the form of LTSs. This is done by the `Compositor` tool [14]. Based on the adaptation contracts, `Compositor` automatically generates an adaptor for each configuration that is guaranteed to orchestrate deadlock-free interactions between the client and a server.

The *Net* for c_A is created as follows [2]:

(i) We add a reconfiguration action $s^\star_A \xrightarrow{r_{s^\star_A}?} s^\star_A$ in the adaptor.

(ii) A_G has two actions for each vector in $\mathcal{AC}_{C,A}$, *i.e.*, $v_1 = \langle C : login!, A : user?\rangle$ generates the actions $v_1! = \langle C : login!, A_{C,A} : login?\rangle$ and $v_1? = \langle A_{C,A} : user!, A : user?\rangle$. A_G has the action $r_{s^\star_A}? = \langle A_{C,A} : r_{s^\star_A}?\rangle$.

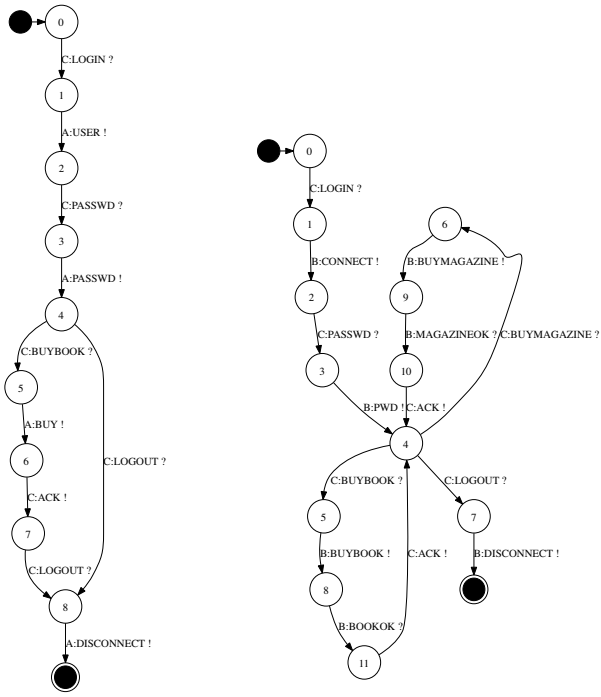
(iii) LTSs of the components and the adaptor are the *Net*'s holes: $J = \{C, A, A_{C,A}\}$; $\tilde{O}_J = \{Sort(C), Sort(A), Sort(A_{C,A})\}$.

(iv) T has a unique state s_T, with $s_T \xrightarrow{a_g} s_T$, $a_g \in A_G$.

No initialisation of server B is needed after the reconfiguration because the system can only be reconfigured in its initial state. Therefore, P_r is a dummy automaton with a unique state and no transition.

Finally, we create $\mathcal{R}_{Net}$: the transducer has states s_A, s_B, $s_{r_{A,B}}$. Non-reconfiguration transitions are: $s_A \xrightarrow{a_g} s_A, a_g \in c_{A_{Net}}$ and $s_B \xrightarrow{a_g} s_B, a_g \in c_{B_{Net}}$.

[2] For the sake of size, the *Net* of c_B is created in a similar way.



(a) Adaptor $A_{C,A}$ (b) Adaptor $A_{C,B}$

Fig. 5. Adaptors

Reconfiguration transitions are: $s_A \xrightarrow{r_{s^\star_A}?} s_{r_{A,B}}$ and $s_{r_{A,B}} \xrightarrow{r_{s^\star_B}?} s_B$.

In this example, reconfiguration is only allowed in the initial state. Nevertheless, our goal is to allow reconfiguration in arbitrary states in which the client and the server have already started a transaction. This is the goal of Section 5.

5 Contract-Aware Reconfiguration

We have shown in the previous section that the running example can be reconfigured at the initial stage of the transaction. Nevertheless, more interesting are scenarios in which reconfiguration can take place at arbitrary stages of the transaction. With that purpose, Section 5.1 defines a *test* that determines whether it is possible to reconfigure the component system at a certain stage of the transaction. Afterwards, Section 5.2 studies how to design a fault-tolerant system in which reconfiguration can take place between two different configurations back and forth.

5.1 History-Aware Reconfiguration

Now, suppose the previous example needs to be reconfigured during an ongoing transaction between C and A. This would be the case where a client connects to the

server A, logs in, and before disconnecting, A is substituted by B. Unfortunately, A and B do not provide such reconfiguration capabilities and it is not possible to substitute A by B without behavioural adaptation because they have different behavioural interfaces.

The client must not abort the ongoing transaction t. Therefore, if the execution trace of C (σ_C from now on) is valid in the new configuration c_B, then C could have been interacting with B from the very beginning. We proceed by initialising B under this assumption, *i.e.*, finding an execution trace σ_B such that the actions performed by C in t are feasible in c_B when B has performed σ_B.

As the adaptor orchestrates the execution between the two parties, we can use the state of the adaptor as the global system state. We say that $t = \langle \sigma_{p_1}, \sigma_{p_2} \rangle \rightsquigarrow_s A_{p_1,p_2}$ when actions performed by P_1 in σ_{p_1} and P_2 in σ_{p_2} lead the adaptor A_{p_1,p_2} to state s.

Definition 5.1 [Trace Compliant]. Let $t \rightsquigarrow_s A_{C,S}$, with $t = \langle \sigma_C, \sigma_S \rangle$ where σ_C and σ_S are traces of C and S respectively.

S' is *trace compliant* to S given t if there exists $t' = \langle \sigma_C, \sigma_{S'} \rangle \rightsquigarrow_{s'} A_{C,S'}$.

Not all components are *trace compliant* given t. However, if t' exists, the new configuration c_B can reach a (deadlock-free) state $s_1^\star$ that simulates the execution of c_A when the latter reaches the state $s_0^\star$. Therefore, it is possible to build a reconfigurable system that starts in a configuration c_A and may reconfigure towards a configuration c_B when the client has performed the actions in t. This reconfiguration does not affect the client in the sense it does not need to abort t nor to rollback. In fact, a new transaction t' is created such that the client continues working on transparently, though it is warned that the adaptation contract has changed; even so, its previous actions are valid and considered in the running transaction (t').

Trace compliance assures that the client's history is unchanged, though it says nothing about future actions. Therefore, it is possible to provide new functionality in the new configuration while maintaining consistency.

Example

In the running example, the login phase is similar in both configurations from the client's point of view. Therefore, it is easy to notice that both configurations are *trace compliant* given $\sigma_C = \{login!;\ passwd!\}$. This allows us to define a reconfiguration from the state $s_0^\star = \{C : s_2, A : s_2\}$ to the state $s_1^\star = \{C : s_2, B : s_2\}$. The initialisation required by B is the result of finding t', *i.e.*, $\sigma_B = \{connect?;\ pwd?\}$ (this specifies P_r). Therefore, meanwhile the client is logged in, the server can be substituted at runtime by another component with a different behavioural interface. Then, the client may log in server A, and after a reconfiguration to configuration c_B the client can buy several books and magazines.

On the other hand, c_B is not *trace compliant* to c_A given an arbitrary trace σ_C. For example, if the client has bought a magazine (or several books) in σ_C, then this trace cannot be simulated in c_A because A only supports transactions in which at one book is sold. Still, c_B is *trace compliant* to c_A for traces that do not include

buying several books or at least one magazine; we shall explore this scenario in Section 5.2.

5.2 Fault-Tolerant System

The previous example allowed reconfiguring the system from c_A to c_B, though reconfiguring the system from c_B to c_A is only possible on some of the traces performed by the client. We investigate here how to design a fault-tolerant system by constraining the behaviour of the previous system. We will guarantee that server A can always be substituted by B (and B by A likewise) transparently from the client's point of view.

One way of ensuring two components are mutually substitutable is by means of a bisimulation equivalence [22]. A more relaxed equivalence takes into account only the behaviour of the servers given the constraints imposed by the environment. In our case, these constraints are summarised within the adaptors because they express the allowed behaviour performed by both parties.

An even more relaxed condition checks whether the adaptors $A_{C,A}$ and $A_{C,B}$ are bisimilar [3] from the client's point of view. There are two options: either to ignore the actions performed by the servers, or to map the actions of one server into another (through yet another adaptation contract). Both solutions allow servers to implement different behavioural interfaces; we have chosen to explore the former.

Even under this relaxed condition it is difficult to find two servers that provide clients with the same functionalities. More likely is that some of the actions allowed to the client are common in both configurations. Hence, if the system were to work on this reduced behaviour it would be possible to provide reconfiguration capabilities as shown in Section 5.1. Our solution is to create new adaptors that are equivalent from the client's point of view. That is:

$$A^R_{C,A} \equiv_C A^R_{C,B} \textit{ iff } \begin{cases} \forall\, t = \langle \sigma_C, \sigma_A \rangle \rightsquigarrow_{s_0^\star} A^R_{C,A},\ \ \exists\, t' = \langle \sigma_C, \sigma_B \rangle \rightsquigarrow_{s_1^\star} A^R_{C,B} \\ \forall\, t' = \langle \sigma_C, \sigma_B \rangle \rightsquigarrow_{s_1^\star} A^R_{C,B},\ \ \exists\, t = \langle \sigma_C, \sigma_A \rangle \rightsquigarrow_{s_0^\star} A^R_{C,A} \end{cases}$$

These are adaptors that constrain the behaviours of the client and the servers such that we can perform reconfiguration at any moment (from the client's point of view). They are found as follows:

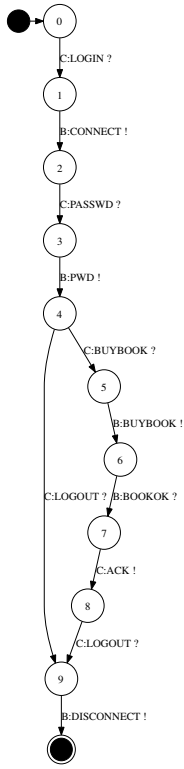
(i) Compute $C^R = \mathit{Product}((\mathit{Hide}(L_A)\ \mathit{in}\ A_{C,A}), (\mathit{Hide}(L_B)\ \mathit{in}\ A_{C,B}))$. If C^R is deadlock-free, proceed.

(ii) Compute $A^R_{C,A} = \mathit{Product}(C^R, A_{C,A})$ and $A^R_{C,B} = \mathit{Product}(C^R, A_{C,B})$.

[3] We leave open which kind of bisimulation is used; for example, strong, weak, or branching bisimulation.

Product synchronises matching labels of both parties into a new synchronised action. *Hide* makes a set of labels L be internal actions (τ).

C^R finds the behaviour that can be performed by the client in both adaptors. This is done by hiding the actions performed by the servers (L_A and L_B) and building the synchronised product. If there are no traces that can be performed by the client in both adaptors, there is a deadlock in C^R and it is not possible to build a fault-tolerant system. Otherwise, using C^R , we constrain each adaptor to this client behaviour which yields, by construction, trace compliant $A^R_{C,A}$ and $A^R_{C,B}$ given any transaction.



Example

In the running example, the constrained system allows the client to buy at most one book in each transaction. In fact, we find that $A^R_{C,A} \equiv A_{C,A}$ but $A^R_{C,B}$ allows only the traces in which the client buys a book or nothing at all (see Fig. on the right). In this scenario, either server A or B suits the client's actions, and thus reconfiguration is possible.

Any σ_C feasible in one of these new adaptors is feasible in the other. Therefore, we can apply the procedure from Section 5.1 for finding pairs of states on which reconfiguration can be applied. We build this way a system that can be reconfigured from one configuration to the other back and forth.

6 Related Work

Dynamic reconfiguration [15] is not a new topic and many solutions have already been proposed in the context of distributed systems and software architectures [12,13], graph transformation [1,23], software adaptation [20,19], metamodelling [11], or reconfiguration patterns [8]. 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 [20], the authors present some issues raised while dynamically reconfiguring behavioural 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 behavioural level is [21,19]. 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 [20,21,19], our goal is slightly different since we do not want to directly reconfigure adaptor behaviours, 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 [6,17,16] focus on existing programming languages and platforms, such as BPEL or SCA components, and suggest manual or at most semi-automated techniques for solving behavioural mismatch. In particular, the work presented in [16] deal with the monitoring and adaptation of BPEL services at run-time according to Quality of Services attributes. Their approach also proposes 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 behavioural issues.

7 Conclusions

This paper has presented a novel framework that supports the design of reconfigurable systems. The formal model defines reconfiguration as a transition from a (static) configuration to another. Each configuration specifies a component assembly with its own set of components and connections, and a *reconfiguration contract* defines *when* the configuration can be reconfigured 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 that, based on adaptation contracts, it is possible to substitute a component by another one that implements a different behavioural interface; this potentially includes mismatches in actions, ordering of actions, and functionality. Briefly, we build on the basis that for some cases it is

possible to find execution traces in which configurations are similar in some sense, and thus it is possible to simulate the execution of a system in another one with a different behavioural interface.

Perspectives. We have presented an initial step on reconfiguration of components with mismatching behavioural interfaces. The framework is expressive and suitable for our needs and we have presented a case-study of runtime configuration. There are many open questions, though, that we wish to continue working on:

We have shown that in a simple setting it is possible to find system states in which it is safe to perform reconfiguration. Nevertheless, for more general scenarios finding correspondences between states based on trace compliance is not enough. We believe that it is necessary to refine what are the necessary properties the new configuration must hold. For example, by endowing a reconfiguration contract with invariants under the form of temporal properties.

An alternative is to augment component LTSs with sub-transactions. In our client/server example, the server would specify that a sub-transaction starts when buying a product and ends when the acknowledgement has been sent to the client. Then, it would be possible to buy magazines from server B (which is not supported by server A), and then substitute B by A.

We also plan to address other scenarios in which a component is substituted by a component assembly. For example, where the server is substituted by a component implemented by a front-end component and a back-end component. Our formal model can formalise the behaviour of such systems, though there are new challenges on how to simulate actions of the former configuration in the new one.

Finally, we plan to integrate this framework within a component model such as *Fractal*. We believe that the model built on *Nets* can be implemented in the runtime platform of Fractal in order to provide components with *safe* reconfiguration capabilities.

Acknowledgement

This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Science and Innovation, and project P06-TIC-02250 funded by the Andalusian local Government.

References

[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] A. Arnold. *Finite transition systems. Semantics of communicating systems.* Prentice-Hall, 1994.

[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. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06*, volume 4294 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2006.

[7] 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.

[8] T. Bureš, P. Hnetynka, 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.

[9] J. Cámara, J. A. Martin, 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.

[10] C. Canal, P. Poizat, and G. Salaün. Model-Based Adaptation of Behavioural Mismatching Components. *IEEE Transactions on Software Engineering*, 34(4):546–563, 2008.

[11] 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.

[12] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[13] J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.

[14] R. Mateescu, P. Poizat, and G. Salaün. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. In *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 84–99, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] N. Medvidovic. ADLs and Dynamic Architecture Changes. In *SIGSOFT 96 Workshop*, pages 24–27. ACM, 1996.

[16] O. Moser, F. Rosenberg, and S. Dustdar. Non-Intrusive Monitoring and Adaptation for WS-BPEL. In *Proc. of WWW'08*, pages 815–824, 2008.

[17] 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.

[18] E. Najm, A. Lakas, A. Serouchni, E. Madelaine, and R. de Simone. ALTO: an interactive transformation tool for LOTOS and LOTOMATON. In *Proc. of Lotosphere Workshop and Seminar*, 1992.

[19] P. Poizat and G. Salaün. Adaptation of Open Component-Based Systems. In *Proc. of FMOODS'07*, volume 4468, pages 141–156. Springer, 2007.

[20] P. Poizat, G. Salaün, and M. Tivoli. On Dynamic Reconfiguration of Behavioural Adaptation. In *Proc. of WCAT'06*, pages 61–69, 2006.

[21] P. Poizat, G. Salaün, and M. Tivoli. An Adaptation-based Approach to Incrementally Build Component Systems. In *Proc. of FACS'06*, volume 182, pages 39–55, 2007.

[22] 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.

[23] 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.

[24] D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.