Component Reconfiguration in Presence of Mismatch

Carlos Canal and Antonio Cansado
Department of Computer Science, University of Málaga, Spain
E-mail: canal@lcc.uma.es

Keywords: component substitution, dynamic reconfiguration, software adaptation

Received: February 20, 2010

This paper discusses how to reconfigure systems in which components present mismatch in both their signature and behavioural interfaces. We are interested in performing component substitution without stopping the system, though we assume components are not designed with reconfiguration capabilities in mind. We also consider that components may need to be adapted before interacting with the system. In this work we identify the basic requirements for achieving runtime component substitution, and define several different interchangeability notions that are adequate to component substitution under behavioural adaptation. Our approach is illustrated with a case-study of a client/server system where the server needs to be substituted by a new one. Classic equivalence and compatibility notions fail to find a new server because the only one available implements a different interface. We show how our interchangeability notions could be used in order to let the system keep on working.

Povzetek: Opisano je preoblikovanje sistemov, ko se zgodi neskladje.

1 Introduction

Software reuse is of great interest because it reduces costs and speeds up development time. Indeed, a vast number of software components are already available through the Internet, and many research and development efforts are being invested in devising techniques for combining them safely and efficiently. In particular, Software Adaptation promotes the use of adaptors in order to compensate mismatch among component interfaces. In fact, this is the only known way to adapt off-the-shelf components since designers usually only have access to their public interfaces. Without adaptation, components could not be put together or their execution could lead to deadlocking scenarios [2, 9].

Still, one of the most challenging issues in Software Adaptation is that systems need to adapt to environmental changes, server upgrades or failures, or even the availability of a new component more suitable to be used in the system. Indeed, the need for finding a new component to be integrated in the system may be either reactive or proactive. The reactive case is caused by the system itself. For instance as a consequence of connection loss or failure of one its components, thus creating a hole in the system that must be filled for its correct functioning. The proactive case would be caused by the availability of a new component that is suspected to be a good candidate for being integrated in the system, replacing some of its current components. In both cases, we have first to detect the need for reconfiguration by using runtime monitoring techniques both on the system and on its environment. Then, the interface of the candidate components —and its compatibility with the rest of the system— must be evaluated, attending not only to its signature interface (names of services, operations, messages, etc.), but also to its behavioural interface (the order in which the elements in the signature interface are expected to be used) and the QoS features provided/expected by the component and the system.

When dealing with this kind of dynamic reconfiguration [14], component substitution must be applied without stopping the complete system, and trying to affect minimally its performance, in particular the functioning of those of its parts that are not directly involved in the reconfiguration. That means that components must collaborate to support reconfiguration capabilities. In fact, it is important to determine when the system can be reconfigured and which kind of properties the system holds after reconfiguration.

Few works have studied the interplay of behavioural adaptation and reconfiguration so far. In most approaches to reconfiguration, substituting a component by another one requires the new component to implement the same functionality as the former one. This means that substitution is usually limited to instances (or subtypes) of a given component. However, it is possible that a component cannot substitute another one, but an adapted version can.

This paper identifies some basic requirements for runtime component substitution and we describe the operations required to achieve this reconfiguration. We also define several different interchangeability notions that are well fitted for component substitution under behavioural adaptation. The paper is structured as follows: Firstly, Section 2 provides some background on behavioural interfaces and adaptation. Then, Section 3 introduces a client/server system that is used as running example through all this document. Section 4 presents our reconfiguration model, for describing systems as a collection of static architectural views (configurations), and reconfiguration operations for moving from one configuration to another one; it also shows how reconfiguration states can be defined at certain points of system execution, and how new components must be initialised for arriving to these states. Next, Section 5 defines different notions of substitutability that we believe are adequate for component replacement under behavioural adaptation. Then, Section 6 outlines the platform that we plan to implement for validating our results. Finally, Section 7 presents related works on reconfiguration and behavioural adaptation, and Section 8 concludes the paper.

This paper builds on our previous work in the field. It is an extension of our position paper [10], developing the ideas presented there, adding many explanations and more detailed examples. In presents also our model for dynamic reconfiguration, and the notions of substitutability discussed in [10] are formally defined here.

2 Background

We assume that component interfaces are equipped both with a signature (set of required and provided operations), and a protocol. For the 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. For reasons of space we omit the signature interface when it can be easily inferred from the corresponding protocol.

Definition 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 or alphabet, $\rightarrow$ 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).

LTSs are adequate as far as user-friendliness and development of formal algorithms are concerned. However, higher-level behavioural languages such as process algebras can be used to define behavioural interfaces in a more concise way. We can use for that purpose the part of the CCS notation restricted to sequential processes, which can be translated into LTS models: $P ::= 0|a?P|a!P|\tau.P|P1 + P2|P/L|A$, where 0 denotes a do-nothing process; $a?P$ a process which receives a and then behaves as P; $a!P$ a process which sends a and then behaves as P; $\tau.P$ a process which performs an internal action τ and then becomes P; $P1 + P2$ a process which may act either as $P1$ or $P2$; P/L is the process P after hiding the names in L, preventing any communication on those names; and A denotes the call to a process defined by an agent definition equation $A = P$. Additionally, we will use the parallel operator $||$ for representing the composition of components —represented by CCS processes— allowing the synchronisation of their input and output actions.

In this paper we will use LTSs or CCS expressions indistinctly for representing components and adaptors. Both could be easily obtained for standard notations such as WS-BPEL or WWF.

2.1 Specification of adaptation contracts

Adaptors can be automatically generated based on an abstract description of how mismatch can be solved. This is given by an *adaptation contract* ($\mathscr{AC}$). In this paper, the adaptation contract between components is specified using *vectors* [8]. Each action appearing in a vector is executed by one of the components, and the overall result corresponds to a loose synchronisation between all of them. A vector may involve any number of components and does not require interactions to occur on the same names of actions. For distinguishing between actions with the same name occurring on different components, we prefix actions with component names.

For example, $\langle C1.on!, C2.activate?\rangle$ is a vector denoting 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; for the transmission of $C1$'s action *on*! to $C2$ as *activate*?, the adaptor will take into account the behaviour of these components 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).

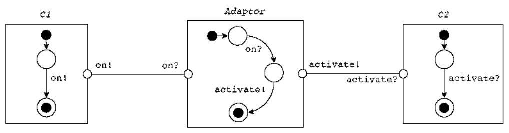


Figure 1: Components $C1$ and $C2$ connected through an adaptor.

2.2 Adaptor generation

Thus, previously to the reconfiguration of the system by the integration of a new component, we will likely need to adapt the component for solving the problems of compatibility detected in the component discovery phase. This will be accomplished by generating an adaptor, that will play the role of wrapper or component in-the-middle, filtering the interactions between the component and the system and ensuring both a correct functioning of the system (verifying for instance the absence of deadlocks or other user defined properties) and the safety of the composition (*i.e.*, that the component is behaving as stated on its interface). In previous works we have developed a methodology for behavioural adaptation (see [9], where our approach for generating adaptors is presented). Following this methodology, both contract specification and adaptor generation are tool supported [8].

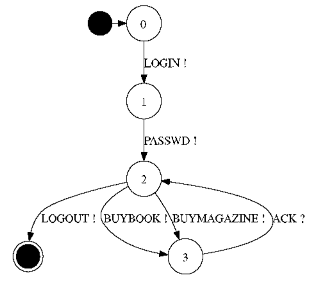
3 Running example

This section presents the running example used throughout the paper. It consists of a client/server system in which the server may be substituted by an alternative server component. This may be necessary in case of server failure, or simply for a change in the client's context or network connection that made unreachable the original server. Suppose that the client wants to buy books and magazines as shown in its behavioural interface in Fig. 2(a). The server A can sell only one book per transaction (see Fig. 2(c)); on the other hand, the server B can sell a bounded number of books and magazines (see Fig. 3(b)). In both cases, sales are represented by a pair of actions (one order and its acknowledgement), and with these two actions we abstract all the details of payment and shipment.

Initially, the client is connected to the server A; we shall call this configuration c_A. The client and the server agree on an adaptation contract $\mathscr{AC}_{C,A}$ (see Fig. 2(b)), which establishes action correspondences between the client and the server A. Obviously, under configuration c_A the client can buy at most one book in each transaction but it is not allowed to buy magazines because this is not supported by the server A. The latter is implicitly defined in the adaptation contract (Fig. 2(b)) since there is no vector allowing the client to perform the action *buyMagazine*!. Finally, the server A does not send the acknowledgement *ack*? expected by the client; this must be worked out by the adaptor, too (see v_4 in Fig. 2(b)).

In an alternative configuration c_B the client is connected to the server B whose protocol is depicted in Fig. 3(b). Similarly, the client and the server agree on an adaptation contract $\mathscr{AC}_{C,B}$ (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, the server B sends a different acknowledgement for each product (see v_4 and v_6 in Fig. 3(a)).

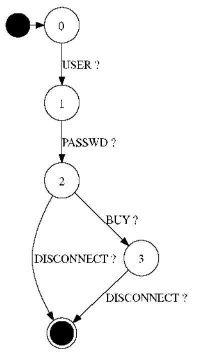
Following the methodology for behavioural adaptation presented in [9], adaptors can be automatically generated for configurations c_A and c_B (see adaptors $A_{C,A}$ and $A_{C,B}$ in Fig. 4). These adaptors ensure by construction that the interaction between the client and servers A or B will take place without deadlock and fulfilling the correspondences of actions described in the corresponding adaptation contracts [9].



(a) LTS of Client C

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

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

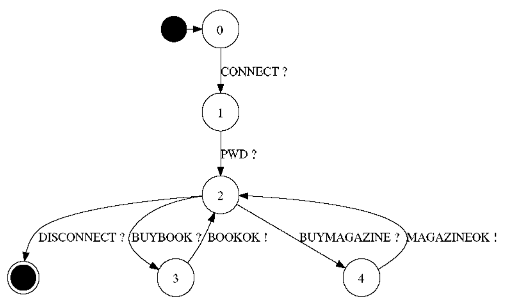


(c) LTS of Server A

Figure 2: Configuration c_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 $\mathscr{AC}_{C,B}$



(b) LTS of Server B

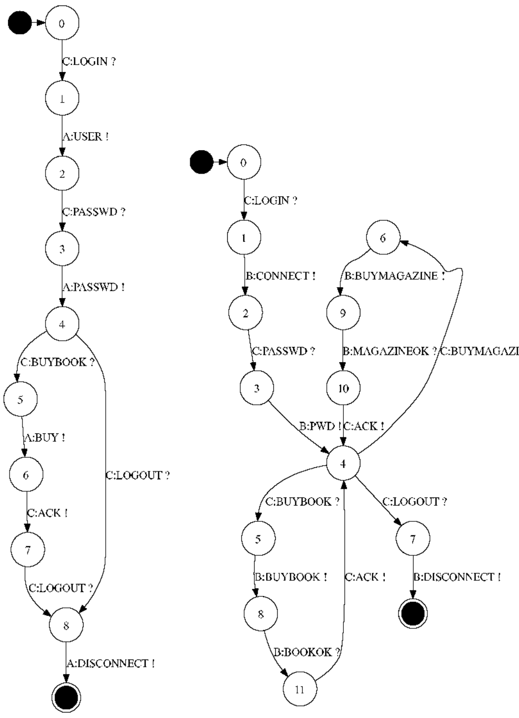
Figure 3: Configuration c_B.

4 Reconfiguration model

This section presents the model that enables both reconfiguration and behavioural adaptation. We define a *reconfiguration contract* to determine how the system may evolve in terms of structural changes.

First, 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 2. [Configuration]. *A configuration* $c = \langle P, \mathscr{AC}, S^{\star} \rangle$ *is a static structural representation*



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

Figure 4: Adaptors

of a system's architecture. P is an indexed set of components. $\mathscr{AC}$ is an adaptation contract of components in P. $S^\star$ is a set of reconfiguration states *defined upon P; these are the states in which reconfiguration is allowed.*

Changing a configuration by another is what we call a reconfiguration. A reconfiguration is specified in a *reconfiguration contract* which separates reconfiguration concerns from the business logic of the system. This way, each configuration can be thought of as a static view of the system, while its dynamic view is specified by a reconfiguration contract.

Definition 3. [Reconfiguration Contract]. *A reconfiguration contract $\mathscr{R} = \langle C, c_0, \rightarrow_{\mathscr{R}} \rangle$ is defined as:*

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

From the definition above, reconfiguration can only take place at predefined states, for guaranteeing system consistency. One certain state of the source configuration ($s_i^\star$) defines when an architecture can be reconfigured. On the other hand, one state of the target configuration ($s_j^\star$) says what is the starting state in the target configuration to resume the execution. Notice also that the target configuration may require a new adaptation contract (allowing replacing a component by another one that implements a different behavioural interface).

Example. In our running example, there are two configurations:
$c_A = \langle \{C,A\}, \mathscr{AC}_{C,A}, S_A^\star \rangle$, and
$c_B = \langle \{C,B\}, \mathscr{AC}_{C,B}, S_B^\star \rangle$.
The reconfiguration contract $\mathscr{R} = \langle C, c_A, \rightarrow_{\mathscr{R}} \rangle$ is given by:
$C = \{c_A, c_B\}$, and $\rightarrow_{\mathscr{R}} = \{c_A \xrightarrow{r} c_B\}$, with
$r = (s_A^\star, s_B^\star)$.
Hence, r specifies 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.
In the simplest scenario, we may consider that reconfiguration from c_A to c_B and back is only allowed at the initial states of the client and the server. This is specified as a unique reconfiguration state $s_i^0 \in S_i^\star, i \in \{A,B\}$ for each configuration, (where $s_A^0 = \{C.s_0, A.s_0\}$ and $s_B^0 = \{C.s_0, B.s_0\}$), and a pair of reconfiguration operations $c_A \xrightarrow{r_{A,B}} c_B$ and $c_B \xrightarrow{r_{B,A}} c_A$, with $r_{A,B} = \{s_A^0, s_B^0\}$ and $r_{B,A} = \{s_B^0, s_A^0\}$ (subindexes in states always refer to state numbers as depicted in Figs. 2 and 3).

In the next section, we will study how other pairs of reconfiguration states —apart from the initial states here— can be obtained.

4.1 Reconfiguration at runtime

In the previous section we have presented our reconfiguration model considering that reconfiguration could be only performed at the initial state of the system (i.e. at static time). Now we will generalise our working scenario allowing reconfiguration to occur when the interactions have already started and the components are in intermediate states (i.e. at dynamic time).

Interactions already performed with the component being substituted cannot be merely ignored; they must be either reproduced up to an equivalent state with the new component, transparently to the rest of the system, or rolled back and compensated when the reproduction of the state is not possible. Both fault-tolerance algorithms, exception handling and roll-back techniques must be developed to this effect, and compensation procedures must be defined when the initiated interactions cannot be correctly finished.

Example. In our running example, if the login phase has already been performed with the system in configuration c_A, and then we need to move to configuration c_B, the server B should be initialised such that the client does not need to re-log in the system. Suppose that the client C has performed a trace $\{login!, passwd!\}$: Then, the initialisation trace for the server B would be $\{connect?, pwd?\}$. Once the server B is initialised as indicated, the system can be reconfigured in order to use the new component. The substitution of the server A by the server B does not affect the client C in the sense it does not need to re-log in the system. In fact,

the client continues working on transparently, though it is warned that the adaptation contract has changed.
This way, we have implicitly defined two new reconfiguration states: $s_A^2 = \{C.s_2, A.s_2\}$ for configuration c_A and $s_B^2 = \{C.s_2, B.s_2\}$ for c_B, and one reconfiguration operation $c_A \xrightarrow{r_2} c_B$, with $r_2 = \{s_A^2, s_B^2\}$.

In the next section, we will present several notions of substitutability that will help us defining additional reconfiguration states and operations for our system.

5 Notions of substitutability

One of the key elements in allowing safe reconfiguration is to determine whether a certain component can be easily replaced by another one.

A relation of equivalence —such as bisimulation ($\sim$) in CCS— cannot be used for these purposes. Indeed, since there is mismatch among the interfaces of the components, a test based on bisimulation would immediately reject servers A and B as equivalent ($A \not\sim B$). Even if we accommodated name mismatch between both servers by using the adaptation contracts $\mathscr{AC}_{C,A}$ and $\mathscr{AC}_{C,B}$ for building name substitutions σ_A, σ_B according to the correspondence of names described in those contracts, the renamed components $A\sigma_A$ and $B\sigma_B$ would still remain not bisimilar, due to behavioural mismatch between them. Thus, we need to define a notion of substitutability adequate for our purposes, indicating whether the replacement of the server A by the server B (or *vice versa*) is suitable in a certain system willing to perform this reconfiguration.

5.1 Contextual equivalence

As we have seen, an equivalence relation like bisimulation is not well suited for our purposes since it takes into account all visible actions possibly performed by the components and ignores the context in which those components operate, and how this context affects them. A proof of equivalence would yield whether two components are interchangeable *in any system*, while we just need to prove if they can be exchanged in a *given* system.

Hence, we need to take into account the influence of the context and to ignore the actions performed by the former and novel components such that the rest of the system continues working transparently. This allows both former and novel components to have different behavioural interfaces as far as their adapted versions provide the same functionality from the point of view of the context. For representing how the context affects the behaviour of a component, we will use the adaptor generated for this component within this context, as described in section 2.

Definition 4. [Contextual interchangeability]. *Two components A and B are interchangeable in the context of another component C iff:*

$$(A||A_{C,A})/L_A \sim (A||A_{C,B})/L_B$$

where $A_{C,A}$, (resp. $A_{C,B}$) is the adaptor necessary for making A (resp. B) interact successfully with C, and L_A (resp. L_B) is the alphabet or set of labels used by A (resp. B) in its communications.

In the definition above, for checking contextual interchangeability we just have to compose the components A and B involved, together with the corresponding adaptors generated for interacting with the context C, and to hide the labels (L_A or L_B) through which the components and their adaptors communicate. The resulting processes represent the components as seen from the point of view of the context C. If they are equivalent —which can be easily checked with CCS tools like the Concurrency Workbench (CWB)—, they can be freely substituted one by another. Any action performed by one of them in the context of C can be exactly reproduced by the other one.

Example. In our running example, consider now a client $C2$ that buys exactly one book in each transaction:
$C2 = login!\ passwd!\ buybook!\ ack?\ logout!\ 0$
$C2$ can interact with server A or B indistinctly. Therefore, the client $C2$ (here playing the role of the context) enforces a behaviour that makes both servers A and B equivalent in the sense above. Hence, we should be able to build a system that is able to reconfigure at any point from the server A to the server B (or from the server B to the server A). Similarly, it is easy to find out that servers A and B are not equivalent in the context of the client C, as originally defined in Fig. 2(a).

5.2 Minimal Disruption

Contextual interchangeability requires that once adapted the components being considered are undistinguishable from the point of view of the context they interact with. A more relaxed notion of substitutability is what we call minimal disruption. Here, only the future actions performed by the environment are taken into account as far as the current system execution —but not *any* possible previous interaction— can be simulated in the new configuration. This is useful when the new configuration has an incompatible behaviour up to a certain point and a compatible one afterwards, but for some specific trace —the current execution— the incompatible part of the behaviour works fine.

Before defining minimal disruption, we have to show how can we enforce a certain component to execute a given trace. This is the purpose of the Definition 5 below.

Definition 5. [Trace-enforcing processes]. *Let $t = \{\langle a_0, \bar{a}_0\rangle, \ldots \langle a_n, \bar{a}_n\rangle\}$ be a trace of actions pairs, where each $\bar{a}_i$ states for the complementary action of a_i (i.e. if a_i is $a!$ then $\bar{a}_i$ is $a?$ and vice versa). Then, we define $Left(t)$ and $Right(t)$ as the processes:*
$Left(t) = a_0 \ldots a_n\ 0$

$Right(t) = \bar{a}_0 \ldots \bar{a}_n\, 0$
obtained by the sequential composition of the left (resp. right) actions from each pair $\rangle a_i, \bar{a}_i\rangle$ in t.

Definition 6. [Minimal disruption]. *Two components A and B are minimal disrupting replaceable in the context of another component C, and given a trace of actions t, iff there exist A_t, B_t, C_t such that:*

- $A_t \sim Right(t)||(A||A_{C,A})/L_A$,
- $B_t \sim Right(t)||(B||A_{C,B})/L_B$,
- $C_t \sim Left(t)||C$, *and*
- *A_t and B_t are equivalent in the context of C_t.*

where as in Definition 4 $A_{C,A}$, (resp. $A_{C,B}$) is the adaptor necessary for making A (resp. B) interact successfully with C.

Hence, for finding out if two components A, B are interchangeable up to minimal disruption in a certain context C, and given a trace t already executed in that system, we just have to make A, B (composed with their corresponding adaptors), and C execute the corresponding part (left or right actions) of the trace, and then prove if the future behaviour of these components is equivalent from the point of view of the context. Again, all this can be easily checked with the CWB. In that case, we can freely perform the substitution of A by B (or the other way round) at the execution point defined by the trace.

Notice that the trace t used in this notion of minimal disruption shows us how to define a reconfiguration state for each configuration ($\langle C.s_i, A.s_j\rangle$ for c_A, and $\langle C.s_i, B.s_k\rangle$ for c_B) which denote the states in which A, B and C are after being enforced to reproduce the trace t, and the corresponding reconfiguration operations (from c_A to c_B and *vice versa*).

Example. In our running example, consider now that we are in configuration c_B —where the client C is interacting with the server B (adapted through $A_{C,B}$)— and they have already executed the trace $\{\langle C.login!, B.connect?\rangle$, $\langle C.passwd!, B.pwd?\rangle$, $\langle C.buybook!, B.buybook?\rangle\}$. If at that point we have to replace the server B with a fresh version of this server (let us call it B') due to server breakdown or connection failure, we have to initialise the new server B' (still adapted through $B_{C,B}$), with the process $connect!\ pwd!\ buybook!\ 0$. Then, the reconfigured system would be able to go on normally.

5.3 History-aware interchangeability

When dealing with component upgrade it is more useful to define a notion of substitutability that we could name as history-aware. Only the current execution needs to be simulated in the new configuration; future actions are allowed to be different. After reconfiguration, the environment may access the new services provided by the new component, or be denied to others that cannot be handled in the new configuration.

Definition 7. [History awareness]. *Two components A and B are history-awareness interchangeable in the context of another component C, and given a trace of actions t, iff there exists A_t, B_t, C_t such that:*

- $A_t \sim Right(t)||(A||A_{C,A})/L_A$,
- $B_t \sim Right(t)||(B||A_{C,B})/L_B$,
- $C_t \sim Left(t)||C$.

where all the processes involved in the definition above are the same as indicated in Definitions 5 and 6.

As we can see, history-aware interchangeability is a precondition for minimal disruption. However, we have preferred to present the notions in this order, from the finest grained to more relaxed notions.

Example. Consider in our running example that we initially are in configuration c_A, with the client logged to the server A (adapted through $A_{C,A}$) and that they have already executed the trace $\{\langle C.login!, A.user?\rangle$, $\langle C.passwd!, A.passwd?\rangle$, $\langle C.buybook!, A.buy?\rangle\}$. If at that point we have to move to configuration c_B, replacing the server A by the server B (for instance because the latter just became available and the client prefers it since it offers a wider functionality), we can check that both servers are history-aware interchangeable in the context of C and for the trace given. Thus, for performing the reconfiguration, we will have to initialise the new server B (adapted through $B_{C,B}$), with the process $connect!\ pwd!\ buybook!\ 0$ and then the reconfigured system would proceed without problems (possibly with the client taking advantage of the extended functionality provided by the new server).

Example. Consider now that we are in configuration c_B, and the trace already executed is $\{\langle C.login!, B.connect?\rangle$, $\langle C.passwd!, B.pwd?\rangle$, $\langle C.buymagazine!, B.buymagazine?\rangle\}$. If at that point the server B became unavailable, we could not move to configuration c_A since for the trace already executed both configurations do not fulfil the conditions of history awareness substitutability (in fact, they would not fulfil any of the notions of substitutability we have defined so far).

5.4 Advanced notions of substitutability

Apart from those presented in previous sections, more advanced notions of substitutability could be envisioned. For instance, we have identified that it would be useful to endow components with (possibly nested) transactions. Once a transaction is finished, there is no need to reproduce it if the component is substituted. This would lead to a notion of *transaction-aware substitutability*, whose utility is shown with the following example:

Example. In our client/server example, the servers would specify two nested transactions: one covers the full servers' protocol, from the login (either with $A.user$ or $B.connect$) to the logout phase (in both cases with $disconnect?$). The other one, would be a sub-transaction, that starts when receiving a buy order, and ends when the acknowledgement has been sent to the client (e.g. from $B.buybook$ to $B.bookok$ for the server B). Then, it would be possible to start the system in configuration c_B, buy some magazines from the server B (which is not supported by the server A) executing the trace $\{\langle C.login!, B.connect?\rangle, \langle C.passwd!, B.pwd?\rangle, \langle C.buymagazine!, B.buymagazine?\rangle, \langle C.ack?, B.bookok!\rangle\}$ and then move to configuration c_A, substituting B by A. As the sale sub-transaction has finished, it can now be safely ignored when substituting the server A. Hence, the trace we would have to consider is just $\{\langle C.login!, B.connect?\rangle, \langle C.passwd!, B.pwd?\rangle\}$, corresponding to the unfinished full session transaction. Now we can find that the move from configuration c_B to c_A fulfils the conditions of history-awareness. Moreover, this would also prevent the client from buying again an already bought product.

6 Component model support

We plan to validate the ideas presented above through real-world applications on implementations using the Fractal component model [5].

Fractal is a modular, extensible, and programming language independent component model for designing, implementing, deploying, and reconfiguring systems and applications. We consider that it is a suitable setting for showing the benefits of our proposals because it deals explicitly with system reconfiguration, and has been the origin of many interesting formal underpinnings that can be applied to analysis of interface compatibility and verification of system properties [6, 3].

The Fractal model is an open component model, and in that sense it allows for arbitrary classes of controllers and interceptor objects, including user-defined ones. This allows us to define our own reconfiguration controllers that will take care of component discovery, adaptation, initialisation, and system reconfiguration. Moreover, in Fractal all remote invocations go through a membrane that controls the component. This makes the membrane an ideal container for a behavioural adaptor: the membrane will intercept all incoming and outgoing messages and pass them to the behavioural adaptor; the latter will compensate mismatch accordingly to the adaptation rules and orchestrate safe executions.

A good starting point for experimenting with our results is to use the framework developed in [4]. The framework is based on a Fractal-compliant component model and uses custom reconfiguration controllers in order to allow the system to self-adapt to changes in the environment. Their model supports dynamic reconfiguration, dynamic component instantiation, and support for interception of functional requests. Moreover, controllers are implemented in the form of a component-based system, which means that each of our controllers would be seen as a component plugged in the component's membrane.

7 Related work

Dynamic reconfiguration [14] is not a new topic and many solutions have already been proposed in the context of distributed systems and software architectures [9, 10], graph transformation [1, 21], software adaptation [17, 16], metamodelling [8, 14], or reconfiguration patterns [7]. 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 [17], the authors present some issues raised while dynamically reconfiguring behavioural adaptors. In particular, they present an example in which a couple 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, notably an incremental approach at the behavioural level is presented in [18, 16]. 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 (i) reconfigure the system (components and adaptors) when a component is removed.

8 Conclusions

We have presented a new research track where components must be adapted to allow the system to be dynamically reconfigured. We have discussed some basic requirements for a runtime component substitution, and we have defined new interchangeability notions that allow to accommodate mismatch in behavioural interfaces. These notions are shown adequate for verifying compatibility of such components. For the same reason, we believe component discovery algorithms should also take into account components that have some degree of mismatch, as far as there is a specification of how mismatch can be worked out.

Finally, before reconfiguring the system, we have shown that the new component must be adapted and initialised accordingly to the current system state. These constitute new requirements for the runtime platform that we plan to address in the short-term.

The work presented here should be taken as an initial step towards dynamic reconfiguration where component candidates present both signature and behavioural mismatch. For the sake of simplicity, we have constrained ourselves to describe component protocols using LTS and CCS. However, one major drawback comes from this decision: data values present in message parameters are omitted. Since the protocols between components are often dependent on the data values carried in message parameters this limits the practical use of the proposal. An obvious extension of this work is to consider more expressive notations for describing behavioural interfaces, for instance Symbolic Transitions Systems (STS), or a value-passing process algebra. This will be part of our future work.

Acknowledgements

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] 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.

[3] 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.

[4] F. Baude, D. Caromel, L. Henrio, and P. Naoumenko. *A Flexible Model And Implementation Of Component Controllers*. Coregrid. Springer, 2008.

[5] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard 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.

[6] L. Bulej, T. Bures, T. Coupaye, M. Decky, P. Jezek, Pavel Parizek, F. Plasil, T. Poch, N. Rivierre, O. Sery, and P. Tuma. CoCoME in Fractal. In Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors, *CoCoME*, volume 5153 of *Lecture Notes in Computer Science*, pages 357–387. Springer, 2008.

[7] TomÃąš Bureš, Petr Hnetynka, and František PlÃąšil. 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.

[8] Javier Cámara, Jose Antonio Martin, Gwen Salaün, Javier Cubo, Meriem Ouederni, Carlos Canal, and Ernesto 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.

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

[10] A. Cansado and C. Canal. On the reconfiguration of components in presence of mismatch. In *Proc. of the 2nd Workshop on Autonomic and SELF-adaptive Systems (WASELF09)*, pages 11–20. SISTEDES, 2009.

[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] Jasminka Matevska-meyer, Wilhelm Hasselbring, and Ralf H. Reussner. Software architecture description supporting component deployment and system runtime reconfiguration. In *In Proc. 9th Int. Workshop on Component-oriented Programming*, 2004.

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

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

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

[18] 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.

[19] 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.