

On the Reconfiguration of Components in Presence of Mismatches

Antonio Cansado and Carlos Canal

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

Abstract. This paper discusses how to reconfigure systems in which components have mismatches in their behavioural specifications. We are interesting in performing component substitution without stopping the system, though we assume components are not designed with reconfiguration capabilities in mind. We also assume that components may need to be adapted before interacting with the system.

The contributions are the enumeration of requirements to allow runtime component substitution and different compatibility notions that are adequate to component substitution under behavioural adaptation.

Finally, we illustrate these requirements and compatibility notions in a case-study of a client/server system where the server needs to be substituted by a new one. Classic compatibility notions fail to find a new server because the only available one implements a different behavioural interface. We show how our compatibility notions could be used in order to let the system keep on working.

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 some mismatches in component's 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 connected or the execution could lead to deadlocking scenarios [2,9].

Still, one of the most challenging issues 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. In dynamic reconfiguration [14], reconfiguration must be applied without stopping the complete system, which 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 before and after reconfiguration. Unfortunately, dynamic reconfiguration is limited when dealing

with off-the-shelf components: not only behavioural mismatches must be compensated, but also the source code is not accessible in case some components do not support reconfiguration.

Few works have studied the interplay of behavioural adaptation and reconfiguration so far. For example, substituting a component by another one usually requires the new component to implement the same functionality as the former. This means that substitution is usually limited to instances (or subtypes) of the same component. On the other hand, component substitution should also consider that components may need to be adapted before interacting with the system. That is, it is possible that a component cannot substitute another one but an adapted version can.

This paper presents requirements for runtime component substitution and different compatibility notions that are more fitted to component substitution under behavioural adaptation. The paper is structured as follows: Firstly, Section 2 provides some background on behavioural specification and adaptation. Then Section 3 introduces a client/server system that is used as running example. Section 4 enumerates some requirements that we have identified for allowing runtime component substitution. Section 5 defines different notions of compatibility that we believe are adequate for component substitution 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 this paper.

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.

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, \rightarrow is the set of transitions: $\rightarrow \subseteq S \times L \times S$.

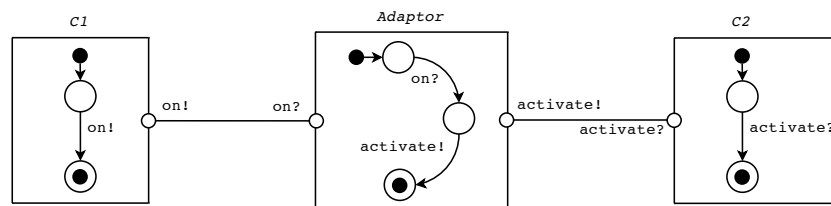
Communication between components are represented using *actions* relative to the emission and reception of messages corresponding to operation calls, or internal actions performed by a component. Therefore, in our model, a *label* is either the internal action τ or a tuple (M, D) where M is the message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

2.1 Specification of Adaptation Contracts

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} is specified using *vectors* [8]. 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.

For example, a vector $v = \langle C1 : on!, C2 : activate? \rangle$ denotes that the action $on!$ performed by component $C1$ must be synchronised with 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 messages between components, the adaptor will take also 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.



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 being 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. 1(a). Server A can sell only one book per transaction (see Fig. 1(c)); on the other hand, server B can sell a bounded number of books and magazines (see Fig. 2(b)).

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. 1(b)). Naturally, 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 server A . The latter is implicitly defined in the adaptation contract (Fig. 1(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. 1(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 is depicted in Fig. 2(b). Similarly, the client and the server agree on an adaptation contract $\mathcal{AC}_{C,B}$ (see Fig. 2(a)). Under configuration c_B , the client can buy a bounded number of books and magazines. In Fig. 2(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. 2(a)).

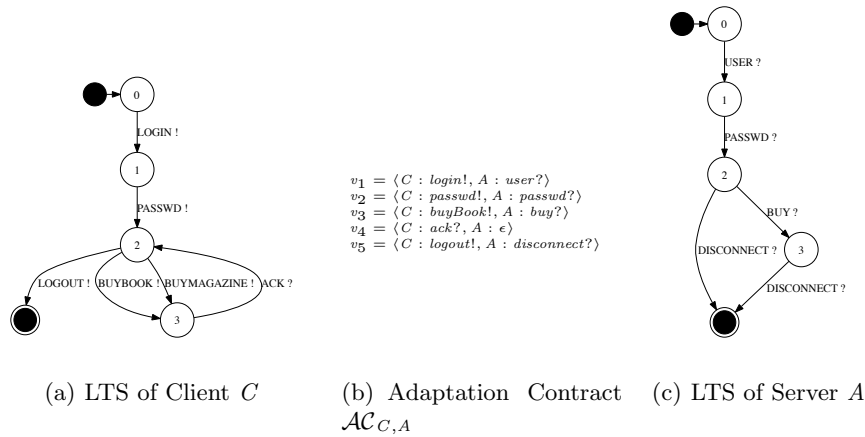


Fig. 1. Configuration c_A .

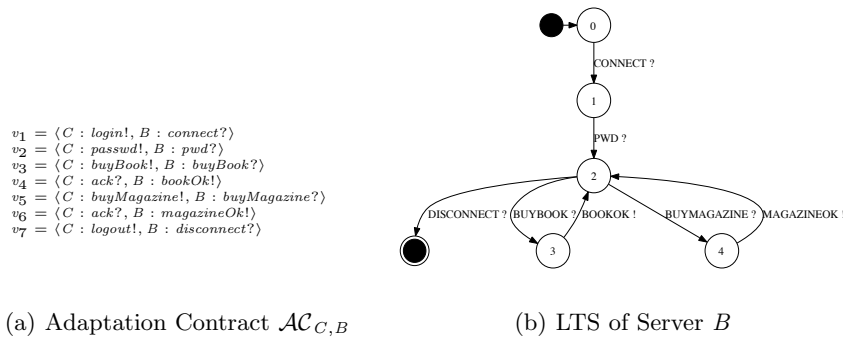


Fig. 2. Configuration c_B .

4 Requirements for Runtime Component Substitution

Runtime reconfiguration in highly dynamic scenarios – as those we are dealing with – impose additional requirements for component replacement. First, reconfiguration must be done at runtime, without stopping the system and trying to affect minimally its performance, in particular the functioning of those parts of it not directly involved in the reconfiguration. Second, new components may need some adaptation in order to fit in: behavioural analysis must provide the sources of incompatibilities, and provide solutions at runtime.

We analyse in this section requirements for performing this kind of dynamic reconfiguration.

4.1 Component Discovery

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 current 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 component. In both cases, we have first to detect the reconfiguration need 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 and parameters), but also to its behavioural interface and the QoS features provided/expected by the component and the system.

Example. If a reconfiguration is triggered to substitute server A , the system searches for candidates and finds server B . Both signature and behavioural analysis on server B point out mismatches, though the system recognises that these mismatches can be worked out and server B is finally accepted as a candidate.

4.2 Component Adaptation

Previously to the system 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).

Example. Before connecting the client to server B , an adaptor is generated based on the adaptation rules established in the adaptation contract $\mathcal{AC}_{C,B}$ in Fig. 2(a).

4.3 Component Initialisation

Interactions already initiated 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 the running example, if the login phase has already been realised, server B should be initialised such that the client does not need to re-login. Supposing the client has performed a trace $\sigma_C = \langle \text{login!}; \text{passwd!} \rangle$, an initialisation trace for server B could be $\sigma_B = \langle \text{connect?}; \text{pwd?} \rangle$.

4.4 System Reconfiguration

Once the new component is initialised, the system is reconfigured in order to use the new component. This must be as transparent as possible to the rest of the system, meaning that the other components should continue working on without being affected.

Example. The system is reconfigured from the configuration c_A to the configuration c_B . The substitution of server A by server B does not affect the client in the sense it does not need to re-login the system. In fact, the client continues working on transparently, though it is warned that the adaptation contract has changed.

5 Levels of Substitution

One of the key elements in allowing safe reconfiguration is to determine when two components are compatible. This has a strong influence in which servers will be accepted as candidates in the discovery phase of Section 4.1.

A bisimulation equivalence can be used to check whether two components can be substituted [18]. Nevertheless, in our case components have mismatches in their behavioural interfaces; thus a test based on bisimulation would immediately reject these components. In this section we discuss some substitutability notions that are more adequate to component substitution involving adaptation.

5.1 Partial Bisimulation

A bisimulation criteria on the behaviours of the former and the novel components is too restrictive because it takes into account all visible actions performed by the components and ignores the behaviour of the environment. In our case, we would like to consider the influence of the environment 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 provide different behavioural interfaces as far as their adapted versions provide the environment with the same functionalities.

Example. In the running example, a client that buys at most one book in each transaction can interact with either server A or B . Therefore, the client (in this case it is the environment) enforces some behaviour that makes servers A and B equivalent in the sense above. Therefore, we should be able to build a system that is able to reconfigure at any point from server A to server B (or from server B to server A).

5.2 Minimal Disruption

A more relaxed substitutability notion is what we call *minimal disruption*. Here, only the future actions performed by the environment are taken into account as far as the current execution can be simulated in the new configuration. This is useful when the new configuration has an incompatible behaviour up to a point and a compatible one afterwards, but for some specific trace – the current execution – the incompatible part of the behaviour works fine.

Example. If the execution of the a configuration can be simulated in another one, then the environment could have been working flawlessly together with the new component from the very beginning. After reconfiguration, the environment keeps on working exactly as it was supposed to because it is within the compatible part of the latter configuration.

5.3 History-aware Substitutability

When dealing with component upgrade it is more useful to demand a *History-aware Substitutability*. Here, 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.

Example. In the running example, a client logs in server *A*. After that, server *A* is substituted by server *B*. Now, new functionality is available to the client because it can now buy several books and magazines.

5.4 Transaction-aware Substitutability

Finally, we have also identified that it is useful to endow components with sub-transactions. Once a sub-transaction is finished, there is no need to reproduce the sub-transaction if the component is substituted.

Example. 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*. As the sub-transaction has finished, it can now be safely ignored when substituting server *A*; this also avoids the client from buying 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 (see Section 4). 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 mismatches 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 [11,12], graph transformation [1,19], software adaptation [16,15], meta-modelling [10,13], 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 [16], 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 pre-defined and fixed set of components. However, notably an incremental approach at the behavioural level is presented in [17,15]. 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) re-configure 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 identified new compatibility notions that are more adequate for verifying compatibility of such components and enumerated requirements for a runtime component substitution.

New compatibility notions are needed because they must allow some mismatches in the behavioural interfaces. 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 mismatches 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.

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. 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.
6. L. Bulej, T. Bureš, T. Coupaye, M. Děcky, P. Ježek, P. Parížek, F. Plášil, T. Poch, N. Rivierre, O. Šerý, and P. Tůma. *CoCoME in Fractal*, volume 5153 of *LNCS*. Springer, April 2008.

7. 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.
8. 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 *ICSE*, pages 627–630. IEEE, 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. 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.
11. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.
12. J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.
13. J. Matevska-Meyer, W. Hasselbring, and R. Reussner. Software Architecture Description Supporting Component Deployment and System Runtime Reconfiguration. In *Proc. of WCOP'04*, 2004.
14. N. Medvidovic. ADLs and Dynamic Architecture Changes. In *SIGSOFT 96 Workshop*, pages 24–27. ACM, 1996.
15. P. Poizat and G. Salaün. Adaptation of Open Component-Based Systems. In *Proc. of FMOODS'07*, volume 4468, pages 141–156. Springer, 2007.
16. P. Poizat, G. Salaün, and M. Tivoli. On Dynamic Reconfiguration of Behavioural Adaptation. In *Proc. of WCAT'06*, pages 61–69, 2006.
17. 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.
18. 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.
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.