On Dynamic Reconfiguration of Behavioural Adaptations

Pascal Poizat[1], Gwen Salaün[2], and Massimo Tivoli[3]

[1] IBISC FRE 2873 CNRS – University of Évry Val d'Essonne, Genopole
Tour Évry 2, 523 place des terrasses de l'Agora, 91000 Évry, France
Pascal.Poizat@ibisc.univ-evry.fr
[2] VASY project, INRIA Rhône-Alpes, France
655 avenue de l'Europe, 38330 Montbonnot Saint-Martin, France
Gwen.Salaun@inrialpes.fr
[3] POPART project, INRIA Rhône-Alpes, France
655 avenue de l'Europe, 38330 Montbonnot Saint-Martin, France
Massimo.Tivoli@inrialpes.fr

Abstract. Software components are now widely used in the development of systems. However, incompatibilities between their observable interfaces may happen and then make their composition impossible. Software adaptation aims at generating as automatically as possible new components called adaptors whose role is to compensate such incompatibilities. Since development of adaptors is costly, it is crucial to make their reconfiguration possible when one wants to modify or update some parts of a running system involving adaptors. In this first attempt, we present the problem of dynamically reconfiguring adaptors and we sketch some ideas of solution on an example. Finally, we end with a list of open issues to be worked out.

1 Introduction

Software components are now widely used in the development of systems, including embedded systems, web services or distributed applications. This area known as Component-Based Software Engineering has still many issues to be solved. Main challenges focus on composition, adaptation and verification of these applications. Software adaptation aims at generating as automatically as possible new adaptors whose role is to compensate incompatibilities appearing in a system constituted of communicating entities.

It is now becoming accepted that entities and in particular their *public interfaces*, most of the time the only observable part of a component due to its black-box feature, have to be represented using dynamic behaviours [22, 10, 3, 19, 7]. In this paper, we deal with adaptors fixing incompatibilities in their behavioural interfaces.

The construction of adaptors can be costly, in particular when built from scratch. Consequently, when one wants to update or modify some parts of a running system, add some new functionalities or needs, suppress out-of-date services, we should propose automated techniques to reconfigure the running adaptors without stopping the whole system.

Reconfiguration can be performed off-line or dynamically at run-time. Dynamic reconfiguration seems more realistic because it is applied while the system is running. On the other hand, it is more difficult in this case to practically take changes into account since modifications have to be made without interrupting parts of the system which are not affected by them. Several possible changes are upgrade, addition or removal of components, and reconfiguration of the architecture such as addition or suppression of connections.

Dynamic reconfiguration [18] is not a new topic and many solutions have already been proposed dealing with distributed systems and software architectures [15, 16], graph transformation [1, 21] or metamodelling [14, 17]. However, to the best of our knowledge, nobody has already worked on the reconfiguration of systems involving adaptors which raises specificities since any change induces modification of the adaptor.

In this work, we consider *open* systems, that are systems where the number of connectors and components is not fixed, and then can vary. Additionally, systems we handle can be made up of several components and several adaptors, even if we modify only one adaptor at a certain moment. Therefore, the other adaptors involved in the system to be reconfigured are viewed as any other component.

A related problem is incremental adaptation [5] which argues for the construction of adaptors step by step by successive refinements. Such successive steps can be viewed as several reconfigurations, then the reconfiguration issue is more general and subsumes incremental adaptation.

The rest of this paper is organized as follows. Section 2 presents our formal model to describe component interfaces and adaptors. In Section 3, we show possible changes that can be performed on a system with several components and an adaptor. This section also sketches some solutions to the reconfiguration issue through an example. Section 4 ends with concluding remarks and perspectives.

2 Component Interfaces and Adaptors

Component interfaces are given using a signature and a behavioural interface.

A *signature* Σ is a set of operation profiles. This set is a disjoint union of *provided* operations and *required* operations. An operation profile is simply the name of an operation, together with its argument types, its return type and the exceptions it raises.

We also take into account behavioural interfaces through the use of labelled transition systems (LTS). A *Labelled Transition System* (LTS) is a tuple (A, S, I, F, T) where: A is an alphabet (set of event labels), S is a set of states, $I \in S$ is the initial state, $F \subseteq S$ are final states, and $T \subseteq S \times A \times S$ is the transition function.

The alphabet of the LTS is built on the signature. This means that for each provided operation p in the signature, there is an element $p?$ in the alphabet, and for each required operation r, an element $r!$. Communication between two LTSs involves one event with complementary actions $p?/p!$. Higher-level behavioural languages such as process algebras can be used to define behavioural interfaces in a more concise way.

We point out that our communication model is *synchronous*: two components synchronize on one event (rendez-vous) and then continue their own evolution. Asynchronous communication can be modelled adding components representing the message queues and interacting with the other components in a synchronous way.

To check if a system made up of several components presents behavioural mismatch, its synchronous product is computed and then the absence of deadlocks is checked on it [8]. An abstract description of an adaptor is given by an LTS which, put into a non-deadlock-free system yields a deadlock-free one. For this to work, the adaptor has to preempt all the component communications. Therefore, prior to the adaptation process, component message names may have to be renamed prefixing them by the component name, *e.g.*, `c:message!`.

3 Adaptor Reconfiguration

3.1 Preliminaries

Changes. First of all, let us summarize the possible changes [18] that can be applied to a system made up of a set of incompatible components and an adaptor making all the entities work correctly together. We distinguish three main classes of changes: (i) upgrade of a component, (ii) addition of a new component, (iii) suppression of a component belonging to the system. Note that an upgrade is a specific case which could be computed as a suppression and an addition of a new component.

In the real world, such changes may appear in many cases. For example, let us imagine two components which can respectively receive orders of books and CDs; a third component could be added to handle DVDs. Another example could be an invoice component in charge of generating invoices for a french electricity company which would be updated to handle only prices in euros and abandon the double printing in euros and francs.

Substitution. As far as component upgrade is concerned, a first case is when the new component has exactly the same behaviour as the one before. Formally, it means that both behaviours are strongly equivalent and it can be checked automatically using `Bisimulator` [6], a tool of the CADP toolbox [11] which allows to verify the most common notions of behavioural equivalences (trace, tau*.a, safety, observational, branching, strong). Equivalences are relations which are preserved on the structure of two behaviours described as automaton. A strong equivalence can be preserved instead of a weak one because our model does not take into account τ actions that are internal actions unobservable from the environment.

If components are not equivalent, several changes can take place in the new component interface. Operations can be removed or added in the signature.

More important are possible modifications of the behaviour where it can concern minimal changes such as renaming of events, addition of an interaction (a new transition) in the automaton, removal of an interaction (suppression of a transition), or bigger changes such as addition or removal of several interactions that are modifications of pieces of behaviour.

Silent portion. We emphasize that if the architecture has already changed, we should have an update (abstract) description of the adaptor since a system can be targeted by several successive changes. As regards adaptor updates *wrt.* component changes, there are two ways to take them into consideration: either modifying the current adaptor, or adding a new adaptor in-between the new component and the previous adaptor [4]. Note that if the adaptor is dynamically updated, modifications have to apply on a *silent portion* of the behaviour, that is a portion not currently engaged in interactions with components to be updated.

Correctness guarantee. Another point concerns the reliability of the updated adaptor *wrt.* the former one. Indeed, checking the absence of deadlocks is required but is not enough to ensure that the system is left in a consistent state after modification. Therefore, the adaptor-to-be must be validated (invariant? checking properties?) off-line before really modifying its running version. Another approach is to build a correct-by-construction adaptor, but in this case our reconfiguration techniques have to be proven as respecting such a claim.

3.2 An example

In this section, we present an example with three components: `C1` communicates with `C2` to send it as arguments a set of documents to store; `C2` receives documents, stores them in a repository, and alerts another component `C3` in charge of counting the number of handled requests. These components cannot interact correctly together because their interfaces are incompatible. Indeed, a deadlock exists at the beginning because no matching of messages is possible. This can be worked out with a simple reordering of events in components `C1` or `C2`. The LTSs for these three components and an abstract description of the adaptor are given in Figure 1 with initial and final states respectively emphasized using an input arrow or a black circle. The adaptor is built following the method proposed in [8] with the three vectors $\langle comm!, comm?, \varepsilon\rangle$, $\langle args?, args!, \varepsilon\rangle$, $\langle \varepsilon, inc!, inc?\rangle$ as an abstract description of the mapping specifying how components `C1`, `C2`, `C3` have to interact.

In the following, we show issues and sketches of solutions on this example for several changes that might be applied to components involved in this system. In this example, we chose to modify the adaptor at hand instead of developing new adaptors in-between as in [4].

A first simple modification is the *renaming* of a message. That induces the renaming of all the instances of this message in the adaptor.

Suppression of a message implies its suppression in the adaptor. It can be automatically computed using CADP tools [11] hidding the concerned message and applying a tau*.a reduction. For instance, if the message `inc!` is removed in `C2`, messages `C2:inc?` and `C2:inc!` are removed in the adaptor. In this case,

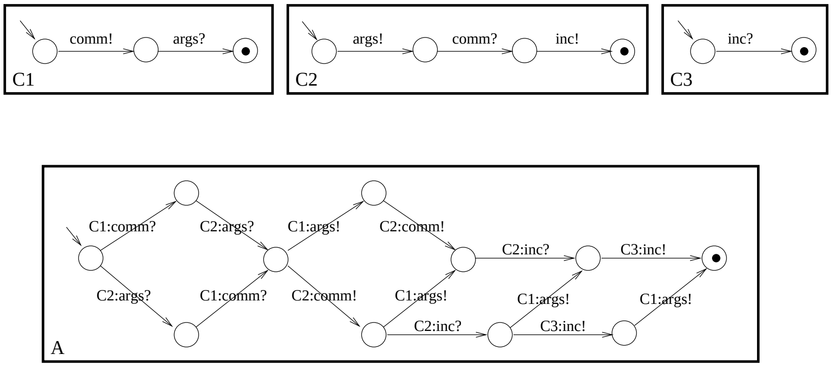

Fig. 1. Three components and an adaptor

the system is not deadlock-free anymore, since `C3` communicates on `inc` too. Let us solve this situation removing the component `C3`. In case of a component suppression, all the messages involved in this component have to be removed from the adaptor, that are `C3:inc?` and `C3:inc!`. We show in Figure 2 all the transitions concerned by these suppressions (red and bold font).

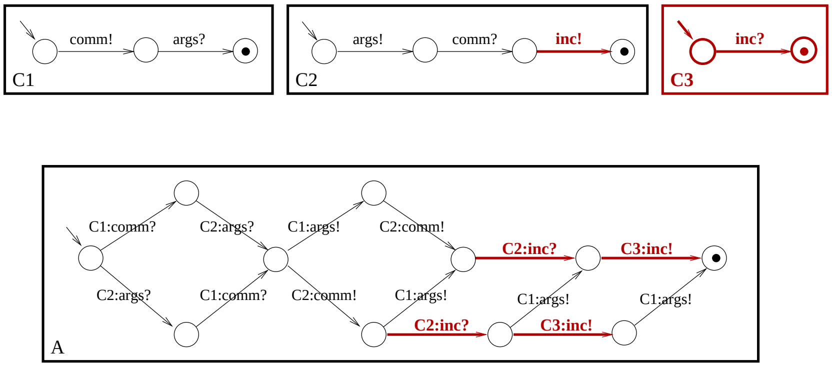

Fig. 2. Suppression of message and component

The adaptor obtained after suppression of these transitions (Fig. 3) is deadlock-free. We emphasize that when components or messages are removed, some services (the ones implemented in the suppressed parts) can be lost. Therefore, the designer has to be informed of that before changes to be effectively taken into account.

The last case focuses on *addition* of message and component. Now, let us add (again) the message `inc!` in `C2`. The resulting updated adaptor can be computed in two ways: (i) computing the new adaptor off-line, and then applying the adequate insertions into the running adaptor *wrt.* it, (ii) traversing the

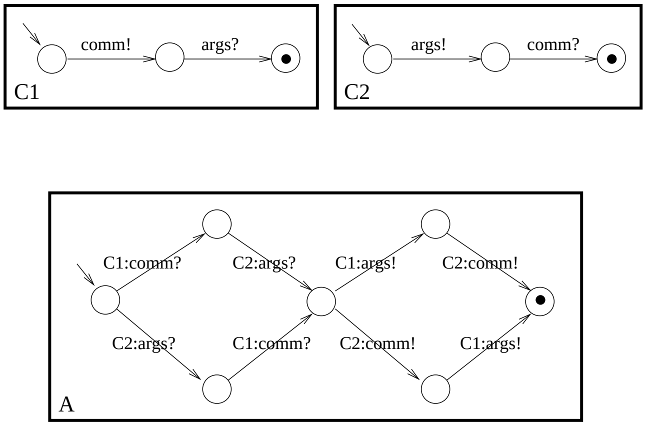

Fig. 3. Adaptor obtained after suppression

adaptor and adding directly into it the new message when it is possible *wrt.* updated component interfaces. Note that in case (i) updates are applied only if the adaptor is deadlock-free whereas in case (ii) the new adaptor can contain deadlocks. Both approaches are meaningful: (i) ensures that the modified adaptor will work, but (ii) can be a first modification followed by another one (the addition of former component `C3`). The latter case (ii) takes place when several modifications should be made successively. These changes have to be applied in sequence within a same silent portion to avoid the running adaptor to have an unexpected behaviour and possibly insert deadlocks within the system.

Figure 4 shows the addition of message `inc!` in `C2` following approach (ii). In a second step, component `C3`, and the original system of Figure 1 is obtained.

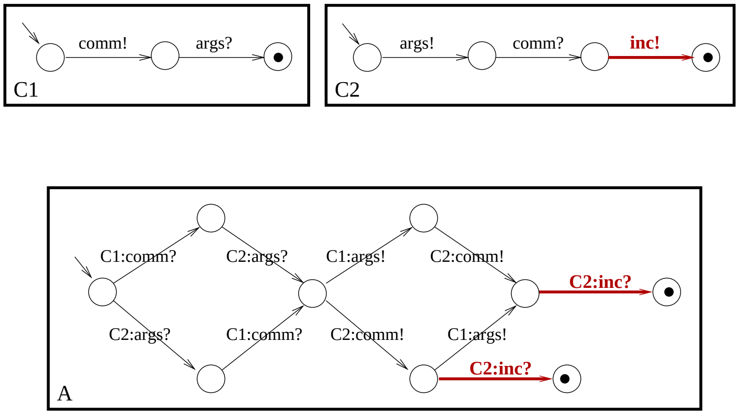

Fig. 4. Addition of message `inc!` in `C2` and `A`

3.3 Automatic handling of the reconfiguration process

An interesting aspect of our approach is concerned with its full automation in handling the reconfiguration process. In other words, to make the reconfiguration process as automatic as possible, we could develop techniques that allow the adaptor to automatically detect and react (by triggering the synthesis of the new adaptor) to changes on the components that it controls.

A possible idea is to enrich (during the automatic synthesis of the adaptor's actual code) the implementation of the adaptor with mechanisms that are suitable for that.

A solution could be the use of exception handling techniques and in particular *Architectural exceptions* [13, 20] that are exceptions that flow between two components. *Fault tolerance* is intended to preserve the delivery of correct services in the presence of active faults. It is generally implemented by error detection and subsequent system recovery. Error detection originates an error signal or message within the system.

Coming back to our context, supposing that a component need to be changed, this activity could be represented as an exception that triggers the component change. System recovery techniques can be used to bring the system in a consistent state before components replacement. For instance, if the component that must be replaced is in execution, system recovery techniques can help the system to reach the state before the component execution.

4 Conclusion

In this paper, we presented the problem of dynamic reconfiguration in the context of a system involving several incompatible components for which an adaptor was implemented and deployed. It was illustrated using a simple example based on a formal model of component interfaces describing signatures and dynamic behaviours (ordering of messages).

It remains several open issues to be worked out before having a satisfactory and completely automated solution to this problem:

- ensuring the correctness of a reconfiguration applied on the system (deadlock-freeness is not enough): correct-by-construction? properties to be checked?;
- applying automatic reconfiguration while the system is running needs to define a notion of consistent state or silent behaviour: how can it be computed? how can it be ensured that it can be obtained?;
- studying reconfiguration as a generation of new adaptor in-between the original one and the involved updated components;
- formalising a language which can be used by a developer to write out the changes he wants to make on the system;
- writing down the different algorithms automating the possible changes *wrt.* a given formal description of component interfaces and expected reconfigurations;

- experimenting our approach on existing implementation languages and frameworks such as COM/DCOM architectures [12], BPEL for web services [2], the Fractal model and its implementations, *e.g.*, ProActive [9];
- reconfiguration/evolution of the adaptor independently of any change in the system.

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-Verlag, 2003.
2. T. Andrews et al. *Business Process Execution Language for Web Services (WS-BPEL).* BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems, February 2005.
3. F. Arbab, F. S. de Boer, M. M. Bonsangue, and J. V. Guillen Scholten. A Channel-based Coordination Model for Components. In *Proc. of FOCLASA'02*, volume 68(3) of *ENTCS*, 2002.
4. M. Autili, P. Inverardi, M. Tivoli, and D. Garlan. Synthesis of "Correct" Adaptors for Protocol Enhancement in Component-based Systems. In *Proc. of Specification and Verification of Component-Based Systems (SAVCBS'04), Workshop at FSE'04*, 2004.
5. S. Becker, C. Canal, J.M. Murillo, P. Poizat, and M. Tivoli. Coordination and Adaptation Techniques for Software Entities. In *ECOOP 2005 Workshop Reader*, 2005. To appear.
6. D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In *Proc. of TACAS'05*, volume 3440 of *LNCS*, pages 581–585, Scotland, 2005. Springer-Verlag.
7. D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web Service Interfaces. In *Proc. of WWW'05*. ACM Press, 2005.
8. C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *Proc. of FMOODS'06*, Italy, 2006. Springer-Verlag.
9. D. Caromel, W. Klauser, and J. Vayssière. Towards Seamless Computing and Metacomputing in Java. *Concurrency - Practice and Experience*, 10(11-13):1043–1061, 1998.
10. L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proc. of ESEC/FSE'01*, pages 109–120. ACM Press, 2001.
11. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2002.
12. P. Inverardi and M. Tivoli. Deadlock Free Software Architectures for COM/DCOM Applications. *Journal of Systems and Software*, 65(3):173–183, 2003.
13. V. Issarny and J.-P. Banatre. Architecture-Based Exception Handling. In *Proc. of HICSS'01*. IEEE Computer Society Press, 2001.
14. 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-Verlag, 2004.
15. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

16. J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.
17. J. Matevska-Meyer, W. Hasselbring, and R. Reussner. Software Architecture Description Supporting Component Deployment and System Runtime Reconfiguration. In *Proc. of WCOP'04*, 2004.
18. N. Medvidovic. ADLs and Dynamic Architecture Changes. In *SIGSOFT 96 Workshop*, pages 24–27. ACM Press, 1996.
19. S. Moschoyiannis, M. W. Shields, and P. J. Krause. Modelling Component Behaviour with Concurrent Automata. In *Proc. of FESCA'05*, volume 141(3) of *Electronic Notes in Theoretical Computer Science*, pages 199–220, 2005.
20. C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, and F. Castor Filho. Exception Handling in the Development of Dependable Component-based Systems. *Softw. Pract. Exper.*, 35(3):195–236, 2005.
21. 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 Press, 2001.
22. D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.