Dynamic Reconfiguration and Access to Services in Hierarchical Component Models

Petr Hnětynka[1] and František Plášil[1,2]

[1] Department of Software Engineering
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, Prague 1, 11800, Czech Republic
{hnetynka, plasil}@nenya.ms.mff.cuni.cz
[2] Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod Vodárenskou věží, Prague 8, 18000, CzechRepublic
plasil@cs.cas.cz

Abstract. This paper addresses the unavoidable problem of dynamic reconfiguration in component-based system with a hierarchical component model. The presented solution is based on (1) allowing several well defined patterns of dynamic reconfiguration and on (2) introducing a *utility interface* concept, which allows using a service provided under the SOA paradigm from a component-based system. The paper is based on our experience with non-trivial case studies written for component-based systems SOFA and Fractal.

1 Introduction

Component-based development (CBD) [19] has become a commonly used technique for building software systems. There are many opinions as to what a component is. One typically agrees that it is a black-box entity with well defined interfaces and behavior, which can be reused in different contexts and without knowledge of its internal structure (i.e., without modifying its internals). However, from a design view, components – especially hierarchical ones – can be viewed as glass-box entities with the internal structure visible as a set of communicating subcomponents. Typically, the collection of the related abstractions, their semantics and the rules for component composition (creation of component architecture) are referred to as a *component model* and an implementation of it as a *component system/platform*. In our view, the concept of "component" has always to be interpreted in the semantics of a particular component model.

Many component systems currently exist and are used both in industry and academia. Typically, the industrial component systems, such as EJB [6] and CCM [15], are based on a flat component model. On the contrary, the academic component systems and models usually provide advanced features like hierarchical architectures, behavior description, coexistence of components from different platforms, dynamically updatable components, support for complex communication styles, etc.

However, it is hard to properly balance the semantics of advanced features – in our view, this fact hinders a widespread, industrial usage of hierarchical component models. Based on our experience with the SOFA [17] and Fractal [4] component models, we claim that this issue is primarily related to dynamic reconfiguration of an architecture, i.e., adding and removing components at runtime, passing references to components, etc. A simple prohibition of dynamic reconfiguration (even though adopted by some systems [2]) would be very limiting, since dynamic changes of architecture are inherent to many component-based applications [14]. On the other hand, particularly in hierarchical component models, an arbitrary sequence of dynamic reconfiguration can lead to "uncontrolled" architectural modification, which is inherently error-prone (we call this *evolution gap problem*, also *architecture erosion* [3]). Moreover, for description of component architectures, most of the component models provide an architecture description language (ADL) [2,4,13,14], which typically captures just the initial components' configuration. (The idea of software architectures and ADL specification came from hardware design, which is static by nature). Thus a challenge is to somehow capture reconfiguration in an ADL.

Another currently emerging paradigm is the service-oriented architecture (SOA) [21]. SOA-based systems (WebServices, etc.) are commonly used in industry. In a high-level view, there is no difference between the SOA and CBD paradigms [10] – both a service and component have a well defined interface, their internal structure is not visible to their environment, and they can be reused in different contexts without modification. However, in SOA, services are not nested and their composition is typically done with the granularity of each request call, frequently being data driven. Thus, because of lack of any continuity in the architecture, there is no problem with dynamic reconfiguration similar to component models.

In this paper, we employ experience with our hierarchical component model SOFA [17] which supports many advanced features like dynamic update, behavior description via behavior protocols, software connectors, and an open-source prototype of which is available [18]. However, based on case studies, we identified deep-going SOFA limits, including dynamic reconfiguration restricted to a dynamic update of a component and the lack of any cooperation with external services, which lead us to the design of the SOFA 2.0.

The goal of the paper is to show how we propose to address the dynamic reconfiguration in SOFA 2.0 with the aim to avoid the evolution gap problem and allow for accessing external services provided through the SOA paradigm. To address the goal, the paper is structured as follows. Section 2 introduces the key contribution – the nested factory pattern and utility interface pattern. Section 3 contains evaluation and related work, while the concluding Section 4 summarizes the presented ideas.

2 Dynamic Reconfiguration and Its Patterns

By *dynamic reconfiguration* we mean a run time modification of an application's architecture. As a special case this includes dynamic update of a component supported by the original SOFA (and also in SOFA 2.0); here the principle is that a particular component is dynamically replaced with another one having compatible interfaces. This kind of dynamic reconfiguration is easy to handle, because all the changes are located in the updated component and are transparent to the rest of the application. Since the new component can have a completely different internal structure, such a component update in principle means replacing a whole subtree in the component hierarchy, being thus a “real” architecture reconfiguration. Also, as an aside, dynamic update is not usually initiated by the application itself but by an external entity (the user, provider, etc.); on the contrary though, a general dynamic reconfiguration is an arbitrary modification of an application architecture typically initiated by the application itself. We have identified the following five elementary operations such a dynamic reconfiguration is based upon: (1) removing a component, (2) adding a component, (3) removing a connection, (4) adding a connection, (5) adding/removing a component’s interface.

As mentioned in Sect.1, in hierarchical component models an arbitrary sequence of these operations can lead to “uncontrolled” architectural modification (the evolution gap problem). To avoid it in SOFA 2.0, we limit dynamic reconfigurations to those compliant with specific *reconfiguration patterns*. At present, we allow the following three reconfiguration patterns: (i) nested factory, (ii) component removal, and (iii) utility interface. In principle the operations (1) – (4) are to be employed in these patterns only, and the operation (5) is limited to the use of collection interfaces (an unlimited array of interfaces of a specific type in principle [8]). The choice of these patterns is based on our experience gained out of non-trivial case studies. Due to space constrains, we below discuss and analyze only (i) and (iii) which we consider the key ones.

2.1 Nested Factory Pattern

The nested factory pattern covers *adding a new component* and *a new connection* to an architecture. The new component is created by a *factory* component as result of a method invocation on this factory. The key related issues are (i) where in the hierarchy the new component should be placed, and (ii) how the connections of/to the new component should be lead.

Consider the situation on Fig. 1a) capturing a fragment of an application featuring the DAccess component, which logs all method calls to a set of loggers connected via a required collection interface. The DAccess is a data access component, which is bound to LFactory (the logger factory) featuring a collection required interface for accessing the loggers. As a result of a call to its provided interface, the logger factory creates a new logger component and returns a reference pointing to it. Such a call is issued by the DAccess component, which in response receives a reference to a new logger and binds to it via the collection interface (dashed line on Fig. 1a).

Provided the DAccess and LFactory components are siblings in the flat architecture, such a dynamic reconfiguration is easy. However, a problem arises when this assumption does not hold as on Fig. 1b). The issue is, where the newly created component (Logger) should be placed in the architecture and how the connection to it should be established.

A straightforward answer to the question where to put the dynamically created Logger components might be into the FactoryManager. However a decision how to manage their connections to DAccess is not that intuitively obvious. If we allow a direct connection between the DAccess and Logger, then the connection will go through the FactoryManager component boundaries and violate the requirement of encapsulation. The second option, to add a copy of the Logger provided interface to the FactoryManager component and lead the connection through it is also not ideal, because it would mean that FactoryManager had to mediate traffic of all connections. In general, if a component A asking creation of another component B (and also assuming A is to be connected to B) is located in a different part of the hierarchical architecture than B is, the problem of mediating connections becomes pressing.

In SOFA 2.0, we have adopted the following rule: The newly created component B becomes a sibling of the component A that initiated the creation (and A's call to the factory also determines the A's collection interface the connection is to be established to). In the example above, the Logger component becomes a sibling of the DAccess component – see Fig. 1c).

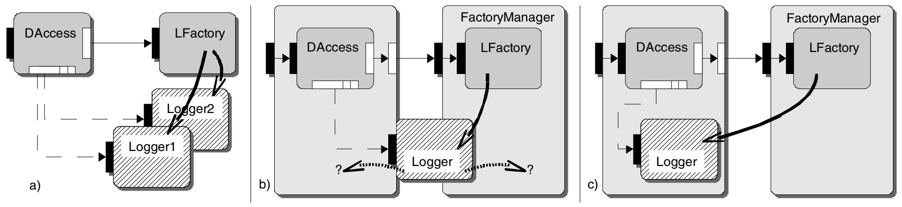

Fig. 1. Dynamic application example

The main reason, why the newly created component B does not become a sibling of the factory component (as this can seem to be also an obvious simple solution) is that the component A which initiated the creation typically needs to intensively collaborate with B which is obviously easier to manage when B is a sibling of A. The next positive outcome of the rule is better performance, because it is not necessary to create complicated connections going up and again down through the hierarchy.

Technically, to identify a factory component, *factory* annotation can be syntactically attached to the factory methods of an interface.

The newly created component B is not limited to having just a provided interface (as it is shown in Fig.1) but it can have also required interfaces. However, these are restricted just to the types featured by the component A initiating the creation. At the moment the provided interface of B is bound, the required interfaces are also bound to the same provisions as the required interfaces of A are. As an aside, this pattern works also in the case when B is a composite component.

2.2 Utility Interface Pattern

While working on case studies, we frequently faced the situation when a component provides a functionality, which is to be used by multiple components in the application at different levels of nesting (i.e. the need of use is orthogonal to the components' hierarchy). The functionality is typically some kind of a broadly-needed service such as printing. A solution can be to place such a component on the top level of the architecture hierarchy and arrange "tunnel" for connections through all the higher-level composite components to those nested ones where the functionality is actually needed. But this solution leads to an escalation of connections and makes the whole component architecture blurred (by making the utility features visible to the components where they are not actually needed) and consequently error-prone. Another typical situation we faced is that a reference to such a service is to be passed among components (e.g., returning reference to a service from a call of a registry/ naming/trading component).

For these reasons, we have introduced *utility* interfaces (the complete meta-model is in [8]). The reference to a *utility* interface can be freely passed among components and the connection made using this reference is established orthogonally to the architecture hierarchy (Fig. 2).

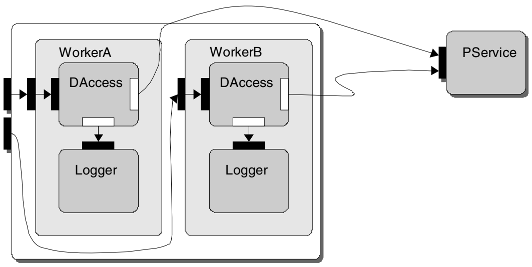

Fig. 2. Utility interface example

From a high-level view, the introduction of utility interfaces brings into component-based models a feature of service-oriented architectures (since Pservice can be seen as an external service). Such feature fusing allows to take advantages of both these paradigms (e.g., encapsulation and hierarchical components of component models and simple dynamic reconfiguration of SOA).

As a side effect, the introduction of utility interfaces this way consequently means that – in a limiting case – the whole application can be built only of components with utility interfaces and therefore the component-based application becomes an ordinary service-oriented application (inherently dynamically reconfigurable). Thus, service oriented architecture becomes a specific case of a component model.

3 Evaluation and Related Work

Evaluation: The approach to dynamic reconfiguration in a hierarchical component model presented in this paper is based on our experience with not-trivial case studies crafted for the SOFA and Fractal component models.

In principle, our approach to handling dynamic reconfiguration is based on combining the features of hierarchical component models and service-oriented architecture. From the component models point of the view, we allow just several types of dynamic reconfiguration compliant with well-defined patterns. Such a prohibition of an arbitrary reconfiguration and allowance of several well-defined modifications only is used in the most of component models (as discussed below), however none of them tackles the issue of how the component factory concept should be integrated into a hierarchical component model. Nevertheless, in addition to addressing this factory issue, the novel contribution of this paper is the introduction of utility interfaces which brings into a component-based system a feature of SOA and allows simplified dynamic reconfiguration without losing some advantages of component models such as focus on reusability and support for integration. Overall, in our view, the utility interface concept sophisticatedly integrates paradigms of the hierarchical component model and service-oriented architecture.

The authors of [12] define a taxonomy of component-based models using the criterion of component composition at different stages of component lifecycle (design and deployment). Using this taxonomy, they classify the existing component systems, including SOFA (the original version), which with Koala and KobrA fits into the most advanced category characterized by (i) composing components at design time, (ii) storing composed components in a repository and (iii) reusing already stored components (including composite ones) in further composition. The only missing feature of these three systems is no composition at deployment time and runtime. With incorporating the proposed dynamic reconfiguration patterns, SOFA 2.0 meets all the criteria imposed in [12] (assuming the authors under "deployment" understand also runtime).

As mentioned in Sect. 2, our choice of reconfiguration patterns is based on our experience with non-trivial case studies of component-based applications. In most of them, we faced a situation where dynamic reconfiguration was necessary. Since the original SOFA has dynamic reconfiguration limited to updates only, we usually had to overcome this lack by restricting the desired dynamic architecture modification via employing "dynamic parts" of a predefined static architecture (e.g., in the example application from Sect. 2.1, a maximum number of concurrent loggers was predefined and the corresponding number of the Logger components was instantiated at launch time). But this approach led to non-generic applications with rather big performance penalties (creating all necessary instances during launching). Also, several of our case studies have been based on the Fractal component model. Fractal provides support for dynamic reconfiguration but as we discuss below it suffers the evolution gap problem.

Related work: Component systems with a flat component model (CCM [15], C2 [20]) do not consider dynamic reconfiguration as an issue, since there is no problem where to place a newly created component and a service can be seen as another component in the flat component space. However, the evolution gap problem is inherently present.

In the area of hierarchical component models, there are several approaches as to how to deal with dynamic reconfiguration.

(1) *Forbidding*. A very simple and straightforward approach used in several component systems (e.g., [2]) is not to allow dynamic reconfiguration at all. But this is very limiting, revealing in essence all the flaws of the static nature of an ADL.
(2) *Flattening*. Another solution is to use hierarchical architecture and composite components only at the design time and/or deployment time. However, at run time the application architecture is flattened and the composite components disappear – this way the evolution gap problem becomes even more pressing, since the missing composite components make it very hard to trace the dynamic changes with respect to the initial configuration. This approach is used, e.g., in the OMG Deployment & Configuration specification [16], which defines deployment models and processes for component-based systems (including CCM). The component model introduced in this

OMG specification is hierarchical, but finally, in the deployment plan, the application structure is flattened and the composite components are removed.

(3) *Restricted reconfiguration*. Several systems forbid an arbitrary reconfiguration but allow special and well-defined types of dynamic reconfiguration:

(a) *Patterns*. Being an extension of Java, ArchJava [1] is a component system employing a hierarchical component model. Components in ArchJava can be dynamically added (using the *new* operator), but an addition of new connections is restricted by *connection patterns*. These patterns define through which interfaces and to which types of components the new component can be connected. Moreover, only the direct parent component can establish these connections (among direct subcomponents).

(b) *Shared components*. Fractal introduces shared components (at the ADL level); a shared component is a subcomponent of more than one other components. This way, component hierarchy becomes a DAG in general (not a tree). Appling this idea to the Fig. 1 would mean that the Logger component would be used by LFactory and DAccess. This solution works nicely, however, an architecture with shared components can be confusing, since it is not easy to determine who is responsible for lifecycle of a shared component, reasoning about architecture (e.g., checking behavior compliance) is very complicated, and several advanced features of component models (e.g., dynamic update of a component subtree) cannot be applied.

(c) *Formal rules*. Several systems (e.g., CHAM [9], "graph rewriting" [23]) define a formal system for describing the permitted dynamic reconfigurations. These systems allow complex definition of all architecture states during an application's lifecycle. But they are very complicated, even for simple architectures.

(4) *Unlimited.* Darwin [13] uses direct dynamic instantiation, which allows defining architecture configurations that can dynamically evolve in an arbitrary way (but the new connections among components are not captured). Julia [11], an implementation to Fractal, allows a general component reference passing (so that any time a reference is passed, it mimics establishing a new connection – this works orthogonally to specifying a shared component in ADL). Obviously, the evolution gap problem is ubiquitous in these cases.

However, let's emphasize that SOA is typically based on dynamic reconfiguration, since the composition of services is done with the granularity of individual calls captured in coordination languages like Linda [22] or by routing of messages [5].

4 Conclusion

We have shown a way of addressing dynamic reconfiguration in a hierarchical component model. With the aim to avoid uncontrolled architecture modification, the presented solution is based on the proposition of three reconfiguration patterns, which include the introduction of the utility interface concept that allows to use a service provided under the SOA paradigm from a component-based system. The paper is based on our experience with non-trivial case studies written for component-based systems SOFA and Fractal. Currently, we have specified the whole meta-model of SOFA 2.0, all necessary interfaces for the development time, deployment and runtime. A working prototype is expected within several months.

Acknowledgements

The authors would like to thank Tomáš Bureš, Vladimír Mencl and Lucia Kapová for valuable comments, Jan Klesnil, Ondřej Kmoch, Tomáš Kohan and Pavel Kotrč for contributing to meta-model design, and Pavel Ježek and Jan Kofroň for sharing experience with a Fractal case study. This work was partially supported by the Grant Agency of the Czech Republic project 201/06/0770.

References

1. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting Software Architecture to Implementation, Proceedings of ICSE 2002, Orlando, USA, May 2002
2. Allen, R.: A Formal Approach to Software Architecture, PhD thesis, CMU, 1997
3. Baumeister, H., Hacklinger, F., Hennicker, R., Knapp, A., Wirsing, M.: A Component Model for Architectural Programming, Proceedings of FACS'05, Macao, Oct 2005
4. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J. B.: An Open Component Model and Its Support in Java, Proceedings of CBSE 2004, Edinburgh, UK, May 2004
5. Chappell, D. A., Enterprise Service Bus, O'Reilly Media, Jun 2004
6. Enterprise Java Beans specification, version 2.1, Sun Microsystems, Nov 2003
7. Hnětynka, P., Píše, M.: Hand-written vs. MOF-based Metadata Repositories: The SOFA Experience, Proceedings of ECBS 2004, Brno, Czech Republic, IEEE CS, May 2004
8. Hnětynka, P., Plášil, F., Bureš, T., Mencl, V., Kapová, L.: SOFA 2.0 metamodel, Tech. Rep. 11/2005, Dept. of SW Engineering, Charles University, Prague, Dec 2005
9. Inverardi, P., Wolf, A. L.: Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model, IEEE Trans. on Soft. Eng., v. 21, n. 4, 1995
10. Iribarne, L.: Web Components: A Comparison between Web Services and Software Components, Colombian Journal of Computation, Vol. 5, No. 1, Jun 2004
11. Julia, http://forge.objectweb.org/projects/fractal/
12. Lau, K.-K., Wang, Z.: A Taxonomy of Software Component Models, Proceedings of EUROMICRO-SEAA'05, Porto, Portugal, Sep 2005
13. Magee, J., Kramer, J.: Dynamic Structure in Software Architectures, Proceedings of FSE'4, San Francisco, USA, Oct 1996
14. Medvidovic, N.: ADLs and dynamic architecture changes, Joint Proceedings SIGSOFT'1996 Workshops, ACM Press, New York, USA, Oct 1996
15. OMG: CORBA Components, v 3.0, OMG document formal/02-06-65, Jun 2002
16. OMG: Deployment and Configuration of Component-based Distributed Applications Specification, OMG document ptc/05-01-07, Jan 2005
17. Plášil, F., Bálek, D., Janeček, R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, Proceedings of ICCDS'98, Annapolis, USA, IEEE CS, May 1998
18. SOFA prototype, http://sofa.objectweb.org/
19. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edition, Addison-Wesley, Jan 2002
20. Taylor, R. N., et al: A Component- and Message-Based Architectural Style for GUI Software, IEEE Transactions on Software Engineering, Vol. 22, No. 6, Jun 1996
21. WebServices, http://www.w3.org/2002/ws/
22. Wells, G.: Coordination Languages: Back to the Future with Linda, Proceedings of WCAT'05, Glasgow, UK, Jul 2005
23. Wermelingera, M., Fiadeiro, J. L.: A graph transformation approach to software architecture reconfiguration, Science of Computer Programming, Vol. 44, Iss. 2, Aug 2002