Advanced Features of Hierarchical Component Models

Petr Hnětynka[1] and Tomáš Bureš[2]

[1] Performance Engineering Lab, School of Computer Science and Informatics, University College Dublin, Ireland
petr.hnetynka@ucd.ie
[2] Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Malostranské náměstí 25, Prague 1, 11800, Czech Republic
bures@dsrg.mff.cuni.cz

Abstract. Using software components has become a widely used development technique for building large enterprise systems. However in practice, component applications are still primarily built using simple component models and advanced component models offering valuable features like component nesting, multiple communication styles, behavior validation, etc. are omitted (by industry in particular). Based on our experience, such an omitting is mainly caused due to usually unbalanced semantics of these advanced features. In this paper, we present a "next-generation" component model SOFA 2.0, which in particular aims at a proper support of such advanced features.

Keywords: Component models, dynamic architectures, connectors, component runtime environment.

1 Introduction

Component-based development (CBD) [18] has become a commonly used technique for building not only large scale enterprise systems but also for virtually any type of sotware. Thanks to the explicit specification of not only components' provided services but also required ones, components have brought easier reuse, better integration, and rapider development compared to former development techniques.

There are many views and definitions of what a software component is but a general agreement is that a component is a black-box software entity with well defined interfaces and behavior. The set of all component features and rules for component lifecycle, composition etc. is usually called *component model.* From the composition point of view, component models can be divided into two categories — flat component models and hierarchical component models. In contrary to the flat ones, the hierarchical component models allow for composite components (in addition to primitive components), which means components hierarchically composed of other components. Thus, an application can be seen as a tree of nested components.

Currently, there are a number of component models each of them trying to provide an ideal solution for building applications. Component models developed and driven mainly by industry (e.g. EJB, CCM, and Koala) offer to developers a stable and mature environment but they use just a flat component model (Koala uses a hierarchical one) and they do not provide advanced features like multiple communication styles, composition verification, seamless distribution, dynamic architectures etc. On contrary, component models developed mainly in the academic environment use in almost all cases hierarchical component models and provide above mentioned advanced features. However academic component models focus very often just on component design and they provide no or very limited runtime environment, which is a major factor that hinders a common (and especially industrial) usage of hierarchical component models. The main reason for the limited runtime support in the area of component models with advanced features is in our view that it is quite hard to properly balance a semantics of all these advanced features.

The goal of the paper is to show strong and weak points of contemporary used component models and then to present the SOFA 2.0 [4] component model that supports hierarchical components and a number of advanced features, while it also provides a stable runtime environment, which serves for executing of component applications.

2 Current component models

In this section, we discuss and compare representatives of common contemporary component models with respect to their supported features and implementations. In particular, we focus on support for describing architectures, possibility of dynamic architectures, support of advanced features like behavior description, multiple communication styles via connectors, etc. (see [4] for detailed explanation of these particular features and the problems related to them), and an existence of runtime support for components. We discuss Darwin, Wright, and ACME component models, which although already rather old had a strong influence on component models developed later. Further, we present CCM, EJB, and Koala as the representatives of the industrial models, and we also devote attention to several representatives of academic component models, namely Fractal, ArchJava, and SOFA. At the end of the section, we briefly refer to several non-standard component models

As it has been mentioned above, Darwin [12] is a classical component model that influenced many later component models. It uses a hierarchical component models without connectors. Darwin allows expressing dynamic changes of architectures using *lazy* and *direct dynamic instantiation*. The lazy dynamic instantiation allows for deferred instantion of components described in the architecture, while the direct dynamic instantiation allows for arbitrary changes. The architecture changes introduced by the direct dynamic instantiation are not captured in the architecture description and thus they are completely uncontrollable. As

Darwin is just an architecture description language (ADL), it does not provide any runtime environment.

Another classical ADL without a runtime environment is Wright [2]. It also uses hierarchical component model but with connectors among components. Behavior of both components and connectors is described using CSP-like notation. Wright does not allow any dynamic changes of an architecture. As it supports connectors, it can use any communication style.

ACME [7] is an ADL intended to serve as a common representation for architecture descriptions. It uses a hierarchical component model with connectors. Both components and connectors can have associated properties (for describing auxiliary information) and design constraints, which mainly serve for describing dynamic changes of an architecture. The language for the design constraints is based on first order predicate logic.

Enterprise Java Beans (EJB) component model [6] developed by Sun Microsystems uses just a flat component model. Even more, it has quite a limited support for describing requirements of components and does not provide any additional features. On the other hand, it provides a stable and mature runtime environment and it is used in many enterprise applications. As it uses a flat component model, EJB does not have any problems with dynamic adding and removing components. From communication styles, it supports method invocation and sending messages.

OMG CORBA Component Model (CCM) is quite similar to EJB but it allows for explicit description of component requirements. From communication styles, it supports synchronous and asynchronous method invocation. Compared to EJB, it is multiplatform and independent of a particular programming language.

Koala [14] has been created by Philips as a component model for developing embedded software (for TVs, set-top-boxes, etc.). It uses a hierarchical component model heavily inspired by Darwin. Primitive components in Koala are implemented as a set of C functions. Koala strongly focuses on component design and optimizations; the Koala compiler (a tool which from ADL generates C header files for implementation) allows removing unused components based on components' configuration properties and further architecture optimizations. The runtime possibilities are however quite limited (as the model is targeted to an embedded environment).

Fractal [3] is a general purpose component model. It uses a hierarchical component model without connectors. Connectors can be simulated using "normal" components (the Fractal specification even instructs to do so), however this results into rather unclean and incomprehensible architectures mixing different levels of abstraction. Fractal separates components functional and non-functional (control) parts. The non-functional part is managed using controllers, which are from the architectonic view provided interfaces. Fractal also introduces the concept of shared components, i.e. a single subcomponent instance shared by several composite components. Such an approach easily allows for runtime changes of an architecture, but it breaks a component encapsulation hierarchy and can result in clumsy and uncontrollable architectures. By itself, Fractal is just a specifica-

tion defining a set of component features and standard interfaces, and it has a number of implementations.

Julia is a Java-based reference implementation of Fractal, which allows component programming in Java. Components can be created either directly via Julia API or using Fractal ADL. In addition to design time, components also exist and can be referenced at runtime. For implementation of control parts of components, Julia uses so called *mixins*, which are Java classes that are woven with the original components' classes using bytecode manipulation. The experience however shows that Julia's approach the component control parts is poorly manageable and hard to extend and debug.

AOkell [17] is another Fractal implementation. It is similar to Julia but it has an elaborate mechanism for building control parts of components based on aspect-oriented programming, thus it addresses several issues Julia has in this context.

The SOFA component model [15] is like Fractal a general purpose component model. It also uses a hierarchical component model but with connectors (and therefore with multiple communications styles). In addition, these connectors allow for transparently distributed applications. Components' behavior can be described using behavior protocol and these can be subsequently used to verify component composition and communication. For describing components and architectures, SOFA uses its own ADL. Similarly to Fractal, SOFA components also exist and may be instantiated at runtime. The weak points of SOFA comprise no support for dynamic changes of an architecture (it just supports a dynamic update of a single component), not clearly separated and non-extensible control part of components, and a limited set communication styles.

Both SOFA and all implementations of Fractal create a component platform over the Java platform; in fact they are Java libraries. At runtime, instances of components exist but they are mapped to a set of Java classes. The ArchJava [1] component system goes another way. It introduces components as a new construct directly into the Java language. Such an approach should prevent violations of the architecture at runtime. However, neither in SOFA nor in Fractal implementations, component developers can diverge from an architecture at runtime. Moreover, as components in SOFA and Fractal are implemented in pure Java, they can be much more easily integrated with other legacy systems.

In addition to the component models presented so far, there also exist component models, which try to capture and describe architectures in a rather formal way. For example CHAM [11] or system based on graph rewriting [19], both targeting description of dynamic architectures. But these systems are very complicated (even for simple architectures) and they do not provide any runtime environment.

3 SOFA 2.0

As described above, currently there are either component systems with strong support for describing architecture and no or almost no runtime support or

systems with a stable and functional runtime environment but rather poor possibilities for designing components. In the middle, there are system like SOFA and the Fractal implementations, which provide both but still they have obstacles that hinder from common usage. SOFA 2.0 which is a next generation of the original SOFA tries address these obstacles and issues.

SOFA 2.0 has inherited a hierarchical component model with connectors and most of other features from its ancestor. The main differences against the original SOFA version are meta-model-based design of components, support of dynamic architecture reconfigurations, support of any communication style, and clearly separated and extensible control parts of components.

SOFA 2.0 is defined using a meta-model. In contrast to ADL-based definition used in the original SOFA, such an approach has many advantages like automated generation of a repository with standardized interface, standardized XML-based interchange format, support for automated generation of models' editors, etc. (see [8] for details). The meta-model defines all features of the component model (in detail described in [4]) and is employed in generation of a repository, which stores component descriptions as well as component implementations. As a particular technology for defining meta-model and generating repository we use EMF [5].

Components are described using *frame* and *architecture* constructs. The frame defines a set of interfaces (services) provided and/or required by a component. The architecture then defines an implementation of the frame; for primitive components the architecture is empty and for composite components it defines subcomponents (again using frames or other architectures) and connections among them.

The support of dynamic architecture reconfigurations is realized via well-defined reconfiguration patterns [9]. Currently, three reconfiguration patterns are provided: factory pattern, removal pattern, and service access pattern. As their names suggest, in factory pattern a designated component serves as a component factory. The removal pattern serves for destroying of a component previously created, while the last pattern allows an access to external services through *utility interfaces.*

For specifying component behavior formally, SOFA 2.0 uses *behavior protocols* [16]. They can be used for verification of component composition during designing an application architecture and also there is a possibility to check a primitive component implementation classes versus protocols.

The SOFA 2.0 runtime environment is called *SOFAnode.* It consists of the repository and a set of deployment docks, each of them can be located on a different computer. A deployment dock is a container for launching components. A single application can spread over several docks; connectors among components assure transparent distribution.

As it has been said above, the repository is generated from the meta-model and stores both component descriptions and implementations. At development time, developers store into it the component information and also reuse the information already stored there. The repository and in fact whole SOFA 2.0 allow

versioning of components. The versioning model used in SOFA 2.0 is the same as in the original SOFA. To guarantee integrity of the development of different versions and the convenient development, the repository supports *cloning*. At the beginning of the development, developers create a new clone of the repository, which mirrors its content. Then they work with this clone and finally, they merge the clone with the original repository. In the cloned working copy, developers can create temporary inconsistencies but the cloning and merging permits to merge just fully consistent repository. The repository also allows export and import of already developed components (e.g. developed by third parties).

From the implementation point of view, a component is set of Java classes. For primitive components, developers have to provide implementation classes. There are not any requirements on these classes, i.e. they do not have to implement/extend any SOFA-specific interfaces/classes. Instead, we use an annotation-based approach where components' provisions, requirements, initializing methods, etc. are marked using annotations. Benefits of this approach are no dependencies on the underlying platform (a single implementation can be resused in different component platforms) and implementation classes can be easily tested by tools like JUnit without starting the whole SOFAnode. As composite components are composed of other components, they do not have any direct implementation and developers do not create any code for them.

Due to versioning, Java class name clashes can occur at runtime, i.e. a situation, when two different classes having the same name are to be loaded into the virtual machine. To address this, SOFA 2.0 uses bytecode manipulation; after compiling component classes, the bytecode of the classes is modified and the classes are renamed to have unique names. In detail, the approach is described in [10].

All bindings among components in SOFA 2.0 are realized via connectors, which are first-class entities in charge of addressing the communication logic. Their use brings the transparent distribution, different communication styles (e.g. synchronous method invocation, asynchronous message delivery, shared memory, streaming, etc.) and the possibility of non-functional aspects (e.g. logging, benchmarking, runtime behavior verification, security, etc.). A connector is specified at design time as a binding (i.e. an hyper-edge connecting several component interfaces) and as a communication style and a set of properties associated with each component interface participating in the binding. In our approach, we use connectors not only in design but also at runtime as well-defined code artifacts. The transition from the high-level design time specification to connector code is realized by a connector generator, which can automatically generate the connector implementation based on design-time and deployment requirements. We perform the generation of connectors only at deployment time, which is when we have the complete information of the application to be launched (including the distribution of particular components to deployment docks); thus, we can choose the most appropriate middleware for addressing the distribution and perform other potential optimizations.

In SOFA 2.0, we have striven to bring out the control part of components and make it easily extensible. The control part (or component controllers) deals with the management logic of a component, which includes management of bindings, lifecycle, some sort of introspection, controlling the update, etc. In the existing component systems (except Fractal), this logic is typically strictly defined by the respective system and cannot be easily changed or extended. Inspired by the Fractal component model, we have introduced to SOFA 2.0 the concept of explicit component controllers. There is a dedicated micro-component model, which allows defining an architecture of the control part, and the concept of a control aspect, which encapsulated a set of micro-components and defines a consistent extension of the control part. Thus, we are able to easily model, extend or even replace the control part of a component.

4 Conclusion and future work

In this paper, we have presented a short overview of contemporary component models and pointed out their main limitations with respect to advanced component features (such as component nesting and modeling, possibility of dynamic architectures, description of component behavior, connectors, multiple communication styles, extensible component control functionality) and the existence and maturity of a runtime environment. Then, we have described a new component model SOFA 2.0 which aims at addressing these limitations. It allows for all the mentioned advanced features and provides a balanced support for them at design-time as well as at development-, deployment- and run-time. At present, SOFA 2.0 has been formalized using meta-models and also an implementation of its runtime is available. The development tools are work in progress.

This work was partially supported by the Grant Agency of the Czech Republic project 201/06/0770. The support of the Informatics Commercialisation initiative of Enterprise Ireland is gratefully acknowledged.

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. J.: A Formal Approach to Software Architecture, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, May 1997
3. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: The Fractal Component Model and Its Support in Java, Software Practice and Experience, Special issue on Experiences with Auto-adaptive and Reconfigurable Systems, 36(11-12), 2006
4. Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, Proceedings of SERA 2006, Seattle, USA, IEEE CS, Aug 2006
5. Eclipse Modeling Framwork, http://www.eclipse.org/emf/
6. Enterprise Java Beans specification, version 2.1, Sun Microsystems, Nov 2003

7. Garlan, D.; Monroe, R. T.; Wile, D.: Acme: Architectural Description of Component-Based Systems, In Foundations of Component-Based Systems, Cambridge University Press, 2000
8. Hnetynka, P., Pise, M.: Hand-written vs. MOF-based Metadata Repositories: The SOFA Experience, Proceeding of ECBS 2004, Brno, Czech Republic, May 2004
9. Hnetynka, P., Plasil, F.: Dynamic Reconfiguration and Access to Services in Hierarchical Component Models, Proceedings of CBSE 2006, Vasteras, Sweden, LNCS 4063, Jun 2006
10. Hnetynka, P., Tuma, P.: Fighting Class Name Clashes in Java Component Systems, Proceedings of JMLC 2003, Klagenfurt, Austria, Aug 2003
11. Inverardi, P.; Wolf, A.L.: Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model, Transactions on Software Engineering, vol. 21, no. 4, Apr 1995
12. Magee, J.; Kramer, J.: Dynamic structure in software architectures, Proceedings of FSE'4, San Francisco, USA, Oct 1996
13. OMG: CORBA Components, v 3.0, OMG document formal/02-06-65, Jun 2002
14. van Ommering, R., van der Linden, F., Kramer, J., Magee, J., The Koala Component Model for Consumer Electronics Software, In IEEE Computer, Vol. 33, No. 3, pp. 78-85, Mar 2000
15. Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, Proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press, May 1998
16. Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002
17. Seinturier, L., Pessemier, N., Duchien, L., Coupaye, T.: A Component Model Engineered with Components and Aspects, CBSE'06, LNCS 4063, Jun 2006
18. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edition, Addison-Wesley, Jan 2002
19. Wermelinger, M.; Fiadeiro, J. L.: A graph transformation approach to software architecture reconfiguration, Science of Computer Programming, Volume 44, Issue 2, Aug 2002