Seamless UML Support for Service-based Software Architectures*

Matthias Tichy and Holger Giese

Software Engineering Group, Department of Computer Science
University of Paderborn, Germany
[mtt|hg]@uni-paderborn.de

Abstract. The UML has become the *de facto* standard for the analysis and design of complex software. Tool support today includes the generation of code realizing the structural model described by class diagrams as well as code realizing the reactive behavior of the dynamic model described by statecharts. However, the CASE tool support for service-based architectures and especially later process phases addressed with component and deployment diagrams is rather limited. In this paper a seamless support of the whole service life cycle of service-based software architectures by means of UML is presented. We present the employed concepts to support the design of services (including full code generation), to compose applications out of services at design time, and to deploy services at run-time. Additionally, we describe our realization of these concepts in form of a CASE tool extension and a run-time framework.

Keywords: development methodologies for UML, service-based architectures, design of distributed Java applications.

1 Introduction

Open service-oriented software architectures [1–3] have received considerable attention as approach to overcome the maintainability problems of large monolithic software. UML [4] CASE tool support for the mentioned approaches for service-based architectures is, however, usually rather restricted. As these service-based approaches traditionally have focused on language mappings to C++ or Java rather than design notations such as UML, currently support for them is most often found in programming environments. Therefore, *generic* UML CASE tools are used for the analysis and design and thus no support for service composition or deployment in dynamic service-based architectures is provided.

Those UML CASE tools can generate source code based on the design results for the structural model described by class diagrams to improve maintainability. Some more elaborated tools can also generate code for the reactive behavior of the dynamic model described by statecharts. Unfortunately, the generated

* This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

source code must be manually adapted by the developer to comply with the special requirements of service-based architectures.

Therefore, the manually realized implementation usually differs from the original design in many aspects. Since the design documents are not changed accordingly when implementing the system the maintainability of such systems becomes deteriorated. The current design is not correctly documented any more and the lost traceability information must be re-engineered by looking into the source code when a re-design is required. Build-in support for roundtrip engineering by some tools tries to overcome the traceability problem for direct changes of the implementation. However, this approach is restricted to high level model information which is still contained in the implementation.

The sketched maintainability problems of service-based software result from software adjustments usually done in the later phases (composition and deployment) to comply with the requirements of service-based architectures. Thus, only the *seamless* support of the whole software life cycle for service-based systems and especially the later phases can prevent the described deterioration of maintainability.

In this paper we present an approach for a seamless UML support for the complete life cycle of service-based systems. UML diagrams are used for all phases in the life cycle. The seamless support manifests itself in specific extensions of the used UML diagram types for service-based architectures, since all specific information due to the nature of service-based architectures must be included in the models. Additionally, the results of prior phases are used as inputs to later phases. Based on the respective UML diagrams, full source code for the service implementation and description files for their composition and deployment are generated by our realization of this approach in form of a CASE tool extension of the Fujaba Tool Suite[1]. Additionally a framework is provided which executes the specified services in a fault-tolerant manner. Jini [1] is used as a representative for a service-based architecture. Due to space restrictions we mainly focus in this paper on the later life cycle activities (Service Realization, Service Composition, and Service Deployment Planning). Nevertheless, we briefly address the System Design activity to highlight the seamless integration with the later activities.

In the following Section 2, we present our approach for service-based architectures. Afterwards, we review in Section 3 the existing body of work and how the presented approach differs. The paper is closed with some final conclusions and an outlook on future work.

2 Seamless Support

Figure 1 shows the activities of the life cycle for service-based systems, which diagram types support the different activities, and the artifacts an activity generates or uses. The different activities of Figure 1 correspond to the subsections of this section.

[1] www.fujaba.de

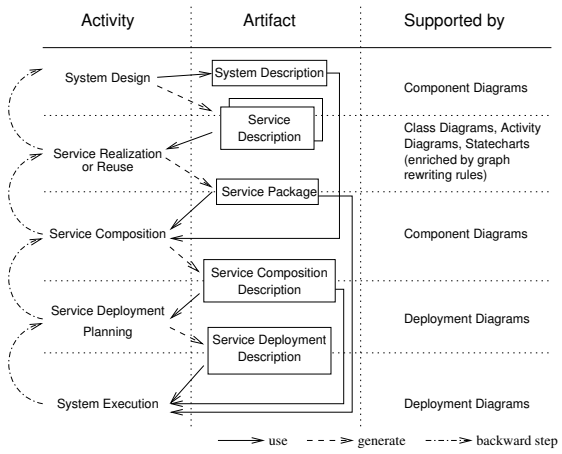

Fig. 1. Activities during the service life cycle

In Section 2.1 the first life cycle activity is described. The topic of this activity is the analysis and design of the system, which should be realized as a service-based architecture. Here, we show how a system (our running example) can be partitioned into several services. This decomposition is described by a UML component diagram. As a result the required services, their interfaces, and connections, which are specified within the component diagram, result in a System Description and multiple Service Descriptions.

We proceed with a description of how one service is realized in Section 2.2. In our approach we use UML class diagrams for the modeling of the structure. For the modeling of behavior we use activity diagrams and statecharts enriched by graph rewriting rules. Based on these design diagrams the implementation resp. the full source code for the service is automatically generated and packed into a Service Package, which is later executed by the run-time environment.

In Section 2.3, service composition is described. We use UML component diagrams to specify the required restrictions for the composition of services and store them in a specific Service Composition Description. The composition of services at run-time by the run-time environment respects the specified interface types and additional attribute restrictions.

The planning of service deployment using UML deployment diagrams is then presented in Section 2.4. For the usage in service-oriented architectures, we have added requirements of node characteristics to deployment diagrams. The planned configuration is then stored in a Service Deployment Description.

In Section 2.5 we present the execution of the specified services by a Jini-based run-time environment. The information provided by the earlier activities (Service Packages, Service Composition Description, and Service Deployment Description) is used for the correct deployment and online-binding. Using the planning deployment diagrams the current situation of a running system is visualized online to support administration.

2.1 Service Design

During the System Design activity the different services which form the service-based system must be determined. During the identification of services UML component diagrams are used. The approach proposed in [5] can be used for the identification of the different services. After the initial set of services are identified their connecting (provided and used) interfaces must be defined. The definition of interfaces includes the declaration of their operations. After the interfaces have been defined the different services are connected via their provided and used interfaces to complete the specification. Thus in the resulting initial component diagram the identified services, their interfaces, and their connections are shown.

Based on the resulting component diagram a System Description is generated which contains information about all services and their connections. For each service contained in the component diagram a Service Description is generated which contains the specification of the service and its interfaces.

Throughout this and all following sections we show the application of our approach using the service-based version and configuration system DSD (Distributed Software Development) [6] as running example. For the sake of a clearer presentation we show only a subset of DSD containing some basic services.

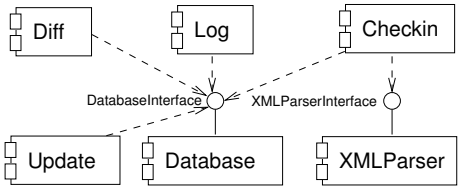

Fig. 2. DSD example

After the initial analysis step several services have been identified (see Figure 2). A Database service and an XML parser service are two of the main services in DSD. Both services are used by the services which provide the version and configuration functionality. In the considered subset of DSD all services use the Database service in order to gather information about the user and its roles in the different development projects. The Checkin service additionally uses the XML parser to read some intermediate file which is generated during the commit process. The information stored in this intermediate file is then written into the database via the database service for long-term storage.

After the identification of the different services and the definition of their interfaces, the realization resp. reuse of services follows.

2.2 Service Realization or Reuse

The above mentioned Service Descriptions contain the services and their interfaces. It is possible that there are already some standard services available which can be used in place for some specific services in the system. For example the

XML parser and the database service are likely already available and can be simply reused. If no standard service is available, the service must be realized. For this service the name and the defined interfaces are extracted from its Service Description. Based on this information an initial class diagram for that service is generated.

This initial class diagram contains a main class and the different interfaces the service provides or uses. The different interfaces are marked by stereotypes ≪ProvidedInterface≫ resp. ≪UsedInterface≫. These stereotypes are later used for the generation of the Service Description contained in the Service Package to differentiate between these two kinds of interfaces. Since the service will be executed by a run-time environment, an initial draft of a helper class is generated, too. This helper class provides some service specific support to the run-time environment.

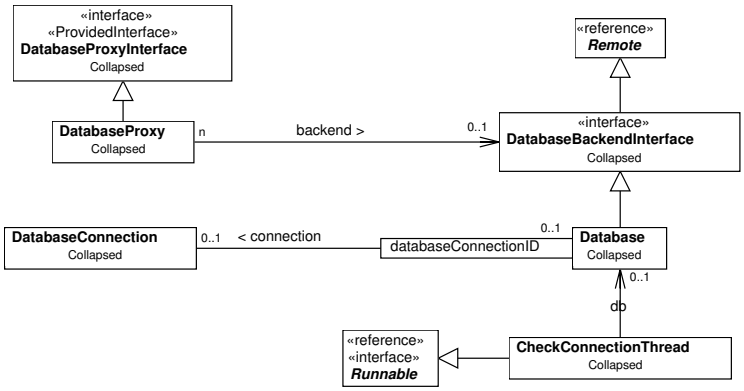

Fig. 3. Simplified class diagram of the database service

The initial class diagram provides a starting point for the developer. For the structural part the initial class diagram must be extended by the developer. Figure 3 shows a simplified class diagram of the database service which has been extended by the developer. Note the ≪ProvidedInterface≫ stereotype attached to the DatabaseProxyInterface interface.

For the behavioral part of the implementation the Fujaba Tool Suite provides additions to basic activity diagrams and statecharts in form of graph rewriting rules [7, 8]. Graph rewriting rules are a powerful design notion for the specification of changes on the object structure. The Fujaba Tool Suite provides code generation for class diagrams, activity diagrams, statecharts, and the graph rewriting rules additions [7, 8]. By the use of these diagrams, their well-defined semantics [9] and the provided code generation the separation between design and implementation is lifted. Thus, the complete behavioral specification of a service is designed using UML and the full source code implementing the design is generated. Therefore, the maintainability of the resulting services is greatly im-

proved compared to manually written or after code generation manually adapted services.

After the realization of the service is finished, the source code is generated and compiled. The resulting service class files and its generated Service Description, which includes information about the service, its interfaces and the helper class, are packaged into a Service Package. When all needed services have been developed and Service Packages have been created, the composition to a service-based system using component diagrams follows.

2.3 Service Composition

To provide seamless support the initial component diagram developed at the beginning (see Section 2.1) is the starting point for the service composition. Now this initial component diagram is refined to reflect the logical structure of the service-based design of the system and to include the implemented services. Our approach targets service-based architectures in which services are not assembled in monolithic applications but form a loosely coupled system of services. These services have no hardwired connections but connect themselves dynamically by the use of online binding. The connection is based on their interfaces. Due to the dynamic nature of service-based architecture we explicitly support *dangling* used interfaces during the service composition activity. Those dangling used interfaces are connected to provided interfaces of other independently deployed services during run-time. In dynamic service-based architectures richer semantic information about the provided and used interfaces is required. Thus, we have added support for the setting of attributes to provided interfaces and adding of attribute restrictions to used interfaces. Our approach is extensible w.r.t. concepts for specifying and matching the behavioral meaning of interfaces (e.g. [10])

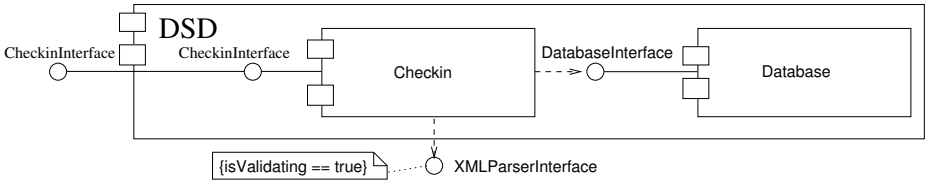

Fig. 4. Service composition

Figure 4 shows a part of the DSD system which describes the composition of the Checkin, Database, and XML parser service. Since the Checkin service should connect to an already available XML parser service during run-time, the XML parser service itself is not part of the component diagram.

In our example the checkin service must be connected only to XML parsers which have the ability to validate the XML file's conformance to an XML schema. Therefore we add a corresponding restriction `isValidating==true` to the used interface. The run-time environment respects these additions to the interfaces

and only connects services where the following conditions are met: 1) the provided interface is the same or derived from the used interface, 2) the attribute restriction expression of the used interface evaluates to true based on the attributes of the provided interface. In our example the database service offers its service via its interface only inside of the compound DSD service. For this encapsulation we use service groups provided by Jini. The specification defined in the component diagram is written to the Service Composition Description which will be used in the Service Deployment Planning activity and for the execution by the run-time environment.

2.4 Service Deployment Planning

After specifying the composition of the services in the next step the deployment (physical mapping) of the services contained in the Service Composition Description has to be planned. In this planning stage the required properties of computation nodes to execute the services are specified using UML deployment diagrams.

Deployment diagrams show the relation between services and nodes. According to the current UML specification [4] this relation means that the service will be executed on that node. Especially for service-based architectures it is rather useful to describe the characteristics a node must have to be able to execute a service. Thus, the deployment plan includes more degrees of freedom which can be utilized by the run-time environment. In our realization the set of nodes, which have the needed characteristics, is defined by boolean expressions on node attributes. The list of node's attributes includes but is not limited to: hostname, jdk version, operating system, memory, ip address.

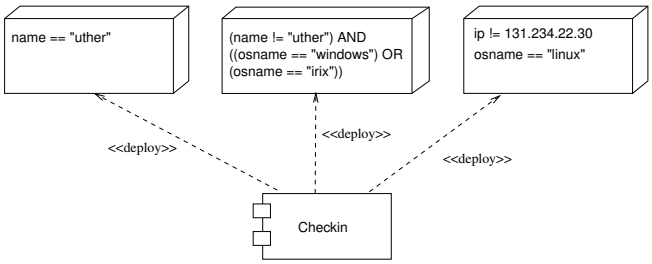

Fig. 5. Deployment planning diagram

For the sake of a clearer presentation of the set of nodes, on which a service can be executed, the service is allowed to be connected with more than one node via a deploy-edge in the deployment diagram. If a service has multiple deploy-edges, a disjunction is used to build the final expression out of the boolean expressions defined for each node. As you can see in Figure 5, two types of specifying a set of nodes are possible. The first type uses a conjunction of a set of attribute restrictions. In the diagram they are displayed vertically as in the right node. Since in some situations this is not convenient a second type is possible,

where an arbitrary boolean expression based on the attribute restrictions can be specified as in the middle node. A Service Deployment Description results from the Service Deployment Planning activity. The actual deployment of the services by the run-time environment is based on this description.

2.5 System Execution

After the planning has been finished, the administrator uses the run-time environment to execute the system of services, modeled by UML diagrams in the previous steps (cf. Figure 1), in a fault-tolerant manner. For each service contained in the component diagram specified during the service composition step (stored in the Service Composition Description), the run-time environment looks for a computational node which satisfies the constraints specified in the deployment diagram (stored in the Service Deployment Description). Then, the run-time environment loads the compiled source code of the service, which has been generated from the structural and behavioral design diagrams, from the Service Package and executes the service on that node. After that, the run-time environment connects the interfaces of the different services according to the constraints specified in the component diagram in the service composition step. Finally, it supervises the execution of all started services and ensures the availability of the services in case of failures. A more detailed description of this run-time environment can be found in [11].

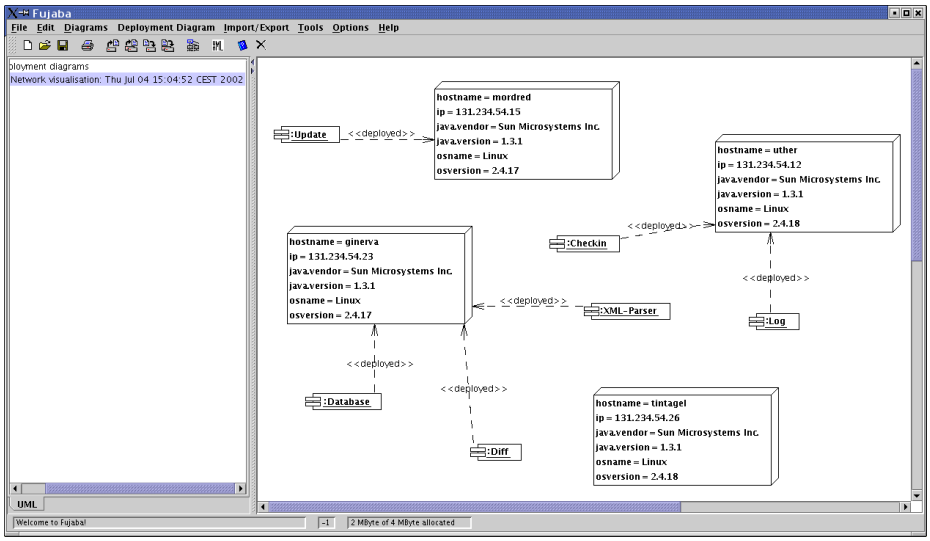

Fig. 6. Deployment visualization diagram

The executed system of services is visualized using the deployment planning diagrams. In this context all nodes which are in the system are displayed including their actual characteristics as well as all services which are executed in the

system. Here the services have connections only to the nodes they are currently executed on. See Figure 6 for a screenshot of the Fujaba Tool Suite displaying the current deployment situation (Note, that the deployment edges now have ≪deployed≫ stereotypes).

3 Related Work

The presented goal of full life cycle support with the UML is very much in line with the model driven architecture (MDA) initiative [12] of the OMG. The MDA claims that full platform-independence is possible by model compilers for the design models. In contrast to this ambitious concept, the presented work focuses on the question how complete UML based life cycle tool support for the specific case of service-based architectures can be achieved.

Cheesman and Daniels in [5] only cover the component specification process, whereas we also cover the later phases (implementation support and especially composition and deployment issues for service-based architectures) and provide tool support. Additionally, they do not take the inherent dynamics (spontaneous networking and online binding) of service-based architectures into account.

Baresi et al. in [13] also use graph rewriting rules in their process for service-based architectures. They focus mainly on the collaboration part of the system and verify that the different collaborations at run-time can indeed be reached by the application of the graph rewriting rules. Our application of graph rewriting rules is targeted at the service implementation not their interconnection. We restrict ourselves to basic composition constraints (type and attribute-conditions) in order to connect the service instances during run-time automatically.

For Jini [1] a reasonable approach for life cycle support of services has been developed in the RIO project [14]. The specification of the service composition via interfaces is done only by an XML file. No UML support and no code generation are provided. The RIO framework itself provides a tool for the visualization of the deployment situation which uses a proprietary graphical representation of service instances. Therefore, support for the life cycle is restricted to composition and deployment, only.

4 Conclusions and Future Work

In this paper we proposed an approach which supports the complete life cycle of service-based architectures by the use of UML. It offers a higher level of maintainability of the resulting service-based system due to the seamless use of UML, the direct generation of full source code, and the direct usage of the UML component and deployment diagrams by the run-time environment to execute the service-based system. Our approach takes the special characteristics of service-based architectures into account. Especially in the later activities our approach offers added value to the developer compared to other approaches.

We are currently further developing our approach to support the UML 2.0 superstructure final adopted specification [15]. In this specification the addition

of ROOM [16] elements like capsules, protocol state machines etc. is proposed. We plan to use protocol state machines, describing the dynamic characteristics of interfaces, in addition to the proposed attribute restrictions to check at run-time, whether the connection of two services via their interfaces is correct.

Acknowledgments
The authors wish to thank Sven Burmester, Matthias Meyer, and Daniela Schilling for comments on earlier versions of the paper.

References

1. Arnold, K., Osullivan, B., Scheifler, R.W., Waldo, J., Wollrath, A., O'Sullivan, B.: The Jini(TM) Specification. Addison-Wesley (1999)
2. Microsoft: Microsoft .NET: Realizing the Next Generation Internet. Technical report, Microsoft (2000) White Paper.
3. Sun Microsystems: Sun[tm] Open Net Environment (Sun ONE) Software Architecture. (2001)
4. OMG: Unified Modeling Language Specification Version 1.5. Object Management Group, 250 First Avenue, Needham, MA 02494, USA. (2002)
5. Cheesman, J., Daniels, J.: UML Components, A simple process for specifying component-based software. Addison-Wesley (2000)
6. Gehrke, M., Giese, H., Tichy, M.: A Jini-supported Distributed Version and Configuration Management System. In: Proc. of the International Symposium on Convergence of IT and communications (ITCom2001), Denver, USA. (2001)
7. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In Engels, G., Rozenberg, G., eds.: Proc. of the 6^{th} International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany. LNCS 1764, Springer Verlag (1998)
8. Köhler, H., Nickel, U., Niere, J., Zündorf, A.: Integrating UML Diagrams for Production Control Systems. In: Proc. of the 22^{nd} International Conference on Software Engineering (ICSE), Limerick, Irland, ACM Press (2000) 241–251
9. Zündorf, A.: Rigorous Object Oriented Software Development. Habilitation thesis, University of Paderborn (2001)
10. Giese, H., Wirtz, G.: The OCoN Approach for Object-Oriented Distributed Software Systems Modeling. Computer Systems Science & Engineering **16** (2001) 157–172
11. Tichy, M., Giese, H.: An Architecture for Configurable Dependability of Application Services. In: Proc. of the Workshop on Software Architectures for Dependable Systems (WADS), Portland, USA (ICSE 2003 Workshop 7). (2003)
12. Gokhale, A., Schmidt, D.C., Natarajan, B., Wang, N.: Applying model-integrated computing to component middleware and enterprise applications. Communications of the ACM **45** (2002) 65–70
13. Baresi, L., Heckel, R., Thöne, S., Varro, D.: Modeling and Validation of Service-Oriented Architectures: Application vs. Style. In: Proceedings of the ESEC/FSE 03, September 1 5, 2003, Helsinki, Finland, ACM Press (2003)
14. Sun Microsystems: RIO - Architecture Overview. (2001) 2001/03/15.
15. OMG: UML 2.0 Superstructure final adopted specification. Technical Report ptc/03-08-02 (2003)
16. Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc. (1994)