Orthographic Modeling Environment

Colin Atkinson and Dietmar Stoll

Lehrstuhl für Softwaretechnik, University of Mannheim, Germany
{atkinson, stoll}@informatik.uni-mannheim.de

Abstract. The rise in importance of component-based and service-oriented software engineering approaches over the last few years, and the general uptake in model-driven development practices, has created a natural interest in using languages such as the UML to describe component-based systems. However, there is no standard way (de jure or de facto) of using the various viewpoints and diagram types identified in general model-driven development approaches to describe components or assemblies of components. To address this problem, we have developed a prototype IDE which provides a systematic and user-friendly conceptual model for defining and navigating around different views of components and/or component-based systems. This is supported by an infrastructure that allows the IDE to be extended with tools that create views and check consistency in an easy and systematic way, and a unifying metamodel which allows all views to be generated automatically from a single underlying representation of a component or component-based system.

1 Introduction

Over the last few years, two of the most significant trends in software engineering have been the move towards component and service-based ways of constructing software systems [1] and the use of model-based approaches to describe their properties and architecture [2]. This has naturally generated an interest in using the two approaches together. However, there is no standard (de jure or de facto) approach for using the various viewpoints and diagram types identified in general model-driven development approaches (e.g. UML class diagrams, UML state diagrams, UML activity diagrams, constraints etc.) to describe components and services or systems assembled from them.

In practice, therefore, developers today use an ad-hoc collection of views which are only loosely related to one another. This not only creates significant consistency problems, because there are no built-in mechanisms for making sure that the different views are consistent, it also creates significant management and navigation problems, because the mapping between logical components and concrete views has to be remembered by developers in their heads. Moreover, each different view generator or editor that is integrated into a development environment such as Eclipse has its own storage artifacts, data persistence formats and navigation trees which usually have to be managed and navigated separately.

There are two main challenges that need to be overcome to provide a clean and logically coherent environment for the model-based development of component and

service-based systems. The first is to devise a flexible and efficient mechanism by which individual views can be generated on the fly from a single, unified model of the component or system. The second is to provide a single coherent navigation mechanism which allows the appropriate view to be selected and managed as needed. In this paper we provide a brief introduction to the tool that we have been developing to meet these challenges. The underlying conceptual foundation for the approach is provided by the Kobra method for component modeling [3], although the ideas developed are more generally applicable. Moreover, the tool was originally customized to support the generation of workflows using pre-developed components [4], but the approach can be used in any domain.

2 “On the Fly” View Generation

During the development of a component, all kinds of artifacts like UML class diagrams, UML state charts and source code are produced. All represent different views of a component and are usually related to each other. As the number of views increases, navigation becomes more tedious and maintaining consistency between related views becomes increasingly difficult. An ideal solution to this problem would be for every tool of the IDE to work on a single underlying model (SUM) and for changes made to individual views to be synchronized directly with this model. In this way, the consistency between the editors and the model is automatically ensured, as long as each individual change to a view is checked for validity against this model. This “on the fly” generation of views is schematically depicted in Fig 1.

Fig. 1. On the fly generation of views

Our long term goal is to create an implementation of the IDE in which all views are generated on the fly in this way, regardless of their form (graphical or textual) or their level of abstraction (executable code or platform independent model). However, it is obviously impossible to populate the IDE overnight with all the editors and view generators needed to usefully model components. Therefore, as a pragmatic intermediate solution the current version supports a mixture of different “tool-native” formats as well as the single, underlying model.

3 Dimension-Based Navigation

Having the ability to generate views on the fly when needed, and to add new kinds of views by simply adding a new view metamodel and associated generation transformations, provides a powerful foundation for the view-based modeling of components, but it doesn't help define what views are most useful and how they should be managed and navigated around. We address this problem by applying KobrA's notion of organizing views around different, fundamental dimensions of concern. Various dimensions can be considered, but in the current version of the IDE the following dimensions are supported –

Composition: This dimension covers the (de)composition of components into sub-components. Selecting a point along this dimension corresponds to the specification of the component or subcomponent which is currently being worked on, e.g. a Bank component or an AccountManager as a subcomponent.

Abstraction: This dimension addresses the platform independence of a view. In other words, selecting a point on the abstraction dimension identifies the level of detail at which the component is being viewed. There are currently three levels of abstraction: specification, realization and implementation. The most abstract level is the specification which provides a black box view of the component. It describes all externally visible properties of a component and thus serves as its requirements specification. The realization of a component describes the design of the component and provides a white box view of its internal algorithms and subcomponents. The implementation level describes the source code and tests.

Projection: This dimension deals with the types of information contained in a view. The three projections currently available are the structural, functional and behavioral projections. The structural projection includes classes and associations involving the component as well as taxonomical information. Operations of a component and their interaction with other artifacts are modeled in the functional projection by means of operation specifications and interaction diagrams. Finally, the behavioral projection focuses on the sequencing and algorithmic properties of the component as manifest by state charts and activity diagrams.

Other dimensions are possible and can easily be added. For example, to model a family of components according to a product line approach, a variant dimension could be added in which each point defines a particular variant. Alternatively, to support views of realizations of different cross-cutting concerns along the lines of aspect-oriented programming, an aspect dimension could be added in which each point defines a different aspect (and one of them defines the core functionality). Since this approach of defining views according to particular combinations of orthogonal perspectives resembles the way in which orthographic projections of physical objects such as houses or cars are defined and organized, we refer to the approach as orthographic software modeling.

Fig. 2 shows a screenshot of the IDE. This particular picture shows a UML diagram representing the structural projection of the specification of a Bank component. The particular points along each dimension occupied by this view can be seen on the left hand side. These can be selected independently.

Fig. 2. IDE with the UML class diagram of a bank

4 Conclusion

In this paper we have outlined the key features of our tool for the orthographic modeling of component and service-based systems – an approach based on the metaphor of multiple orthogonal views. The current prototype version is a heterogeneous mix of native tool views and pure views dynamically generated from the single underlying model. We are currently implementing a view generation engine using a transformation language and aim to reduce the number of native tools to zero in the near future.

Acknowledgements. This work was largely performed as part of the AristaFlow project under the support of the State of Baden-Württemberg.

References

- [1] Szyperski, C., Gruntz, D., Murer, S.: Component Software - Beyond Object-Oriented Programming? 2nd edn., 608 pages. Addison-Wesley / ACM Press (2002), ISBN 0-201-74572-0
- [2] Mukerji, J., Miller, J.: Overview and guide to OMG's architecture (June 2003), <http://www.omg.org/cgi-bin/doc?omg/03-06-01>
- [3] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J.: Component-Based Product Line Engineering with UML. Addison-Wesley Publishing Company, Reading (2002)
- [4] Dadam, P., Reichert, M., Rinderle, S., Atkinson, C.: Auf dem Weg zu prozessorientierten Informationssystemen der nächsten Generation - Herausforderungen und Lösungskonzepte. In: Spath, D., Haasis, K., Klumpp, D. (eds): Aktuelle Trends in der Softwareforschung - Tagungsband zum doIT Software-Forschungstag 2005, June 2005, Schriftenreihe zum doIT Software-Forschungstag, Band 3, MFG Stiftung, pp. 47–67 (2005)