A meta-model driven methodology for state transfer in component-oriented systems

Yves Vandewoude
Yolande Berbers

Report CW 349, October 2002

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A meta-model driven methodology for state transfer in component-oriented systems

Yves Vandewoude
Yolande Berbers

Report CW 349, October 2002

Department of Computer Science, K.U.Leuven

Abstract

State transfer between versions is one of the most difficult challenges related to online upgrading. This position paper describes a methodology that attempts to solve this problem using meta-models constructed from the source of a component. This meta-model is subsequently modified to prepare the component for run-time replacement. The methodology uses heuristics and algorithms to anticipate which member variables in different component versions are meant to correspond. Additional semantic information is provided by the user. The proposed technique offers full separation of concerns, and has prospects to include intelligent state transfer support for live update systems.

Keywords : Dynamic Updating, Component Systems, State Transfer.

1 Introduction

Recent research shows that on average, maintenance accounts for more than 80% of the IT budget of Europe's companies. Updating software to a new version is a tedious and costly process due to the impact this has on day-to-day operations. Halting large software systems is expensive and inability to continue operation often leads to a loss in customer goodwill. This problem can be avoided using the technology of *dynamic updating*[1]. Although a common terminology hasn't been established so far in this relatively new domain, the term usually stands for all modifications that are applied to a program at run-time (i.e. without stopping it).

Examined more closely, the issue of dynamic updating consists of two sub-problems:

Addition of code: When an application is modified at run-time, the new version of changed code must be loaded by the application.

State transfer: The state that is contained in the part of the application to be replaced, must be transferred to the new version.

Most work in the field of live updates concentrates on the first sub-problem. In the past, many solutions were presented to replace portions of an application at run-time. For procedural systems, Hicks ([7, 8]) uses dynamic patches that consist of verifiable native code. With CONUS, the programmer can replace a component using a declarative specification of the desired changes ([10]). For object-oriented applications written in Java, systems were proposed that use a variety of methods in order to modify a running application. Technologies used include modifications to the Java Virtual Machine ([1, 11]), language extensions ([3, 5]) and meta-architectures ([12]). To give an exhaustive overview of existing systems is beyond the scope of this paper, and we refer to [6, 7, 13, 14] for more complete surveys.

The second sub-problem, transferring state, is usually left to the programmer, or ignored completely. Although certain systems include some tool support ([1, 7]), this is often limited to the generation of a framework in which the user can implement the transition. Little intelligent support is offered to transfer the state itself.

The remainder of this paper will propose a method to build a more intelligent tool that can support the programmer during the proces of a state transfer for component-based systems. We have begun implementation of this tool specifically for a component system that is developed by Flemish universities in the context of the SEESCOA project funded by the Belgian IWT ([2]). Although the methodology is designed with the SEESCOA component system in mind, we are confident that meta-model driven state transfer is applicable to other component systems or object-oriented systems.

In section 2 different representations for the state of a component are discussed. We present in detail the different steps of our approach in section 3.1. After

[1] In this paper we will use the terms *dynamic updating*, *live updating*, *dynamic adaptability* and *run-time reconfiguration* interchangeably.

a motivation for the use of meta-models (section 3.2), future work is given in section 4. We conclude in section 5.

2 State-representation

State transfer implies the need for a suitable representation of internal state of a component. This subject justifies some more thought, since the choice of representation has its implications when a mapping must be found between different states. Two possible representations will be discussed: *abstract state* and *implementation based state*.

Abstract State: in this approach, an abstract representation of the state is created and all implementation specific program states are mapped to this abstract representation. This approach has the advantage that when different old versions must be upgraded to the latest version, only one mapping must be defined (i.e. between the abstract state and the new implementation). Furthermore, knowledge of the implementation of the old component version is not required, since all relevant information is contained in the abstract state. Some systems already apply this technique, albeit in a restricted problem domain (e.g. in the DiPS/CuPS framework ([9]) an abstract representation is used in the context of protocol stacks).

Implementation Based State: in this approach, the state that is extracted from a specific implementation version must be interpreted by the transition function. Since every component has its own state representation, the state transition function depends on both the old and the new version of the component. An advantage of this ah hoc method of representing state is that it does not require the developer of the component to provide a mapping of the component state to the abstract state.

Abstract states seem to be the most elegant solution, but it is not yet clear whether this approach is practical or even possible in a broad application domain. We are currently investigating a hybrid approach, which uses implementation based state representation for the component as a whole, but does use abstract state for specific structures (e.g. an abstract *collection* that represents a `List`, a `Hashtable` or a `Vector`).

3 Meta-model driven state-transfer

3.1 Transferring state in five steps

In a series of subsequent steps, the source-code of a component will be instrumented with the necessary functionality to allow it to be replaced at run-time. This will include methods to export its state and to import the state of a previous component version. Where necessary, extra code will be added to deal with additional administrative tasks as moving ports from the old the new component

version[2]. The entire process passes through following five steps:

1. In a first step, the source code of the newest version of the component is parsed and an Abstract Syntax Tree is constructed. Starting from this AST, a meta-model is built. This meta-model is a logic representation of the internal structures contained in the component sources, and as such a convenient method to access the semantic information present in the component sources. Analogously, a second meta-model is constructed from the sources of the previous (active) component version.

2. Using these two meta-models, the state of both the active and the new component version can be determined. The SEESCOA component system requires that a component is inactive before it can be replaced[3]. Therefore, the state of the components is entirely located in the instance variables of the classes that make up the component. In some cases, references to open files or external libraries whose source code is not available may be present in a component. These references are currently treated as any other member variable. Possible complications related to this matter are being examined, but are not treated in this paper.

3. The third and most difficult step is identifying which variables in the two versions correspond. This process can never be automated completely, since not all semantic information can be derived from the sources of a component. Consider the case where a triangle is stored in the component. While the first version of the component can represent the triangle as a sequence of three points, the new version may very well use three angles and a side. The semantic equivalence between those two representations can not be derived from the sources, hence human interaction will always be required. However, many cases exist in which algorithms or heuristics can derive a correct correspondence between member variables. Next to relative simple techniques (mapping the contents of variables that carry the same name), the meta-model representation of the state allows for much more intelligent heuristics and algorithms. For example, many algorithms that were developed in the field of database scheme evolution are excellent candidates to be integrated in our tool. The programmer still guides the proces and has the option to override the proposals made by the tool.

4. Combining the results of the available algorithms with the information provided by the user, the meta-model of the new version of the component is modified. Methods are generated that allow the component to export or import its state. Additional modifications that are required to ensure a smooth replacement are also carried out. Examples are renaming classes in a Java based component system (Java does not allow to load a class that is already present) and the initialization of ports.

5. From the modified meta-model, the source code for the updateable component is generated. This component contains all necessary information

[2]This is the case for the SEESCOA component system. It is to be expected that other component systems require similar tasks to be performed before replacing a component.

[3]This inactivity (or *quiescence*) of a component that will be replaced is required by most systems for live updating. ([1, 8, 10])

to accomplish its replacement at run-time.

An illustration of the entire process is given in figure 1.

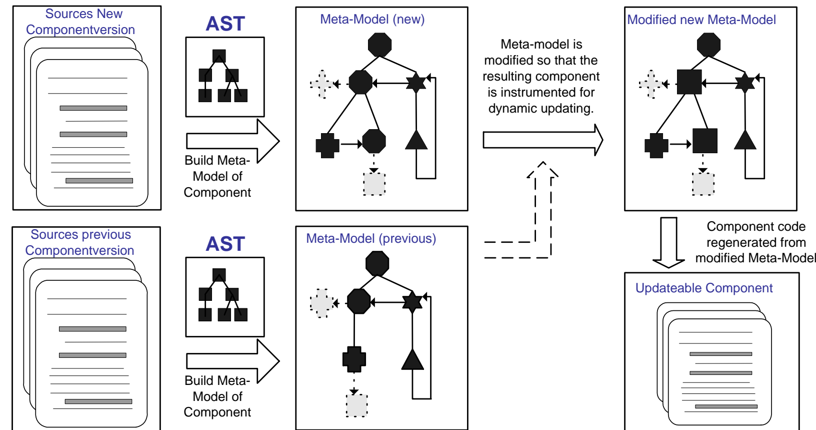

Figure 1: Schematic overview of the methodology

The five steps in which a component is prepared for dynamic updating occur in a separate phase of development, after the functional behaviour of the component is implemented. This *separation of concerns* avoids adding replacement functionality to a component unless it is required, and keeps the original design of the component from being cluttered with non functional code.

3.2 Using meta-models for state transfer

The usage of meta-models to perform the transformation to the component is an important aspect of the methodology. The global overview offered by the meta-model results in two major advantages:

Information-extraction: Retrieving information from a collection of source files is hard. Since the meta-model is a global representation of a component, it allows for easy extraction of information that is contained in in many different source files. A meta-model can also contain additional information that is not present in the source files (e.g. it can contain information about JDK base classes).

Component Instrumentation: Due to the global range of the model, system wide changes can be achieved much more easily. Changing a name of a method or class can be an extremely complex operation on the source-code level. The global view also allows for the implementation of more complicated heuristics or algorithms.

Although the usage of a meta-model facilitates transformations of a component, construction of a good meta-model from scratch is far from trivial. Therefore, we will use an existing meta-model for Java as a starting point. The JNome project

([4]) constructs a meta-model from the sources of a Java application in order to generate meaningful and detailed documentation. The JNome framework will be adjusted and extended with the concepts required to build a meta-model of a component, and will then be integrated in the tool.

Another important advantage of the proposed methodology is the separation of concerns that was discussed in the previous section.

4 Future Work

The concepts presented in this paper are being implemented in a tool that will assist the programmer to transfer state between different versions of a component. In order to be useful, the tool must be able to make intelligent suggestions to the programmer and detect corresponding structures in both versions. Therefore, algorithms from different research domains will be applied to state transfer and their results will be analyzed. We will also investigate the applicability of our methodology to other component-based and object-oriented systems. Finally, we wish to examine to what extend the validity of proposed transformations can be verified by the tool.

5 Conclusion

We have proposed a methodology that allows for state transfer between components using meta-models. Due to the large amount of semantic information that is available in the meta-models, the proposed method has prospects of allowing for complex algorithms to assist the user in transferring state between component versions. Since the component is modified in a separate step during development, full separation of concerns is achieved.

References

[1] J. Andersson and T. Ritzau. Dynamic code update in JDRUMS. In *Workshop on Software Engineering for Wearable and Pervasive Computing*, Limerick, Ireland, June 2000.

[2] Y. Berbers. A component-oriented approach for building complex embedded systems. In I. Mang, editor, *Proceedings of the 7th International conference on Engineering of Modern Electric Systems*, pages 205–210, 1999.

[3] P. Costanza. Dynamic object replacement and implementation-only classes. In *Proceedings of the 6th International Workshop on Component-Oriented Programming at ECOOP*, Budapest, Hungary, June 2001.

[4] J. Dockx, N. Smeets, K. Mertens, and E. Steegmans. JNome: A Java Metamodel in detail. Technical Report CW323, KULeuven Department of Computer Science, December 2001.

[5] Günter Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of ECOOP 99*, Lisbon, Portugal, June 1999.

[6] D. Gupta. *On-line Software Version Change.* PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, November 1994.

[7] M. Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, June 2001.

[8] M. Hicks, J. T. Moore, and S. Neccles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.

[9] N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten. Towards Hot-Swappable System Software: The DIPS/CuPS Component Framework. In *Proceedings of the Seventh International Workshop on Component-Oriented Programming*, Malaga, Spain, April 2002.

[10] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[11] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, June 2000.

[12] B. Redmond and V. Cahill. Iguana/J: Towards a dynamic and efficient reflective architecture for Java. In *Workshop on Reflection and Meta-Level Architectures at 14th European Conference on Object-Oriented Programming*, Cannes, France, June 2000.

[13] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for a dynamic updating. *IEEE Software*, 10(2):53–65, 1993.

[14] Y. Vandewoude and Y. Berbers. An overview and assessment of dynamic update methods for component-oriented embedded systems. In *Proceedings of The International Conference on Software Engineering Research and Practice*, Las Vegas, USA, June 2002.