System Evolution by Metalevel Modification

Article · May 2009

Source: DOAJ

CITATIONS
0

READS
29

2 authors, including:

Michal Vagac
Univerzita Mateja Bela v Banskej Bystrici
21 PUBLICATIONS **11** CITATIONS

SEE PROFILE

System Evolution by Metalevel Modification

Michal Vagač and Ján Kollár

Matej Bel University, Faculty of Sciences
Tajovského 40 Banská Bystrica, Slovakia
Email: vagac@fpv.umb.sk

Department of Computers and Informatics
Technical University of Košice, Faculty of Electrical Engineering and Informatics
Letná 9, 042 00 Košice, Slovakia
Email: Jan.Kollar@tuke.sk

***Abstract* – This paper describes system evolution managed by corresponding metasystem. The metasystem builds a metamodel of base system and allows its modification. The modification is propagated back to the base system. The application model presents the example of standard graphics user interface developed with Java Abstract Windowing Toolkit (AWT), which is a part of the Java Foundation Classes (JFC). The main aim is to confirm the possibility of application properties monitoring using aspect-oriented programming, their abstraction in a metamodel, possibility of their alternations by metamodel modifications and consequent change in the original application model.**

***Keywords:* System evolution, AOP, metalevel, metasystem, dynamic adaptation.**

I. INTRODUCTION

Software maintenance is often more expensive than the development itself and adaptation is undervalued in the traditional development process [1].

Most software systems model real processes, which evolve and change in time. Hence parallel evolution of software system and process is needed. The most significant change requirements include adding new features and removing bugs. The most common solution of this problem is to modify system's source files. The implementation of any change is tightly coupled with the system.

A new approach is to have a presence of a general technique which monitors existing system and allows its consequent modification. The result will be a new version of the original system, which can by monitored (and later again modified) using the same general technique.

This work was supported by Project VEGA No. 1/4073/07 Aspect-oriented Evolution of Complex Software Systems

The paper follows describes an example of extending application model with a monitoring code, which tracks specific properties of an application and constructs corresponding metamodel.

After metamodel alternations, the changes are reflected back to the base system. As an example of application model a standard graphics user interface developed with Java Abstract Windowing Toolkit (AWT) is provided. With utilization of AOP, the application model is extended with a code which is monitoring transitions between dialogs of the application and records these transitions into state diagram. After modifying the state diagram, changes are reflected in the original application.

II. ASPECT-ORIENTED PROGRAMMING, METAPROGRAMMING AND SOFTWARE SYSTEMS EVOLUTION

Aspect-oriented programming (AOP) is a programming technique for modularizing concerns that crosscut the basic functionality of programs. It allows a programmer to identify and treat separately concerns that, subsequently, can be woven to different target applications.

Metaprograms are programs that represent and manipulate other programs or themselves as their data (metaprograms are programs about programs).

Metaobjects describe, control, and implement or modify objects of an application domain. In the case of multi-layered architecture, a metaobject can control other metaobjects.

A *computational system* is a system written in a particular programming language whose purpose is to answer questions about and/or support actions in some domain. A *reflective system* is a computational system whose domain is the system itself [3]. A *metasystem* is a computational system whose domain is another computational system. The domain of a metasystem is called its *base system* [4]. The most common metasystems processing tool is a compiler.

Introspection is the ability of a program to simply reason about reifications of otherwise implicit aspects of itself or of the programming language implementation

[5]. Introspection play significant role in program visualization [6] and in reflective environments for real-time systems [7], [8]. Without ability for introspection, aspect oriented approach to software development would fail [9].

The *software evolution* is the process to achieve satisfactory, controlled and disciplined software change [10]. A reason for software change can be for example a change request or a bugfix.

Computer systems that support dynamic software evolution have the ability to change their implementation at runtime allowing them to extend, customize or upgrade the services that they provide without the need for system recompilation or reboot [11].

Computer system change can be accomplished by system modification or adaptive language implementation (change of computer system behavior depends on adaptive language change) [2]. Language modification means difficult process. To simplify this process, it is possible to specify language by a set of annotated domain classes [13].

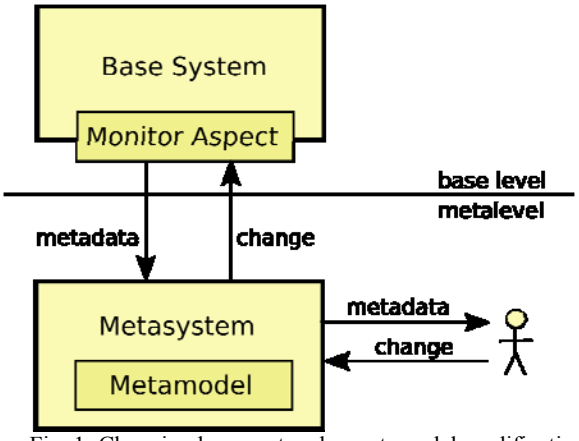

Fig. 1: Changing base system by metamodel modification

III. METALEVEL CONSTRUCTION AND MODIFICATION USING AOP

Besides clearer modularization possibility, aspect-oriented programming allows extensions of existing code with a new functionality. It is even possible to add a new functionality without access to source codes. In our approach, this AOP property is used to extend base system with necessary functions used to gather information about a base system and to access base system's internal structures.

The gathered information is used to build a metamodel. It represents known state of a base system's selected property. The metamodel is provided to a user who can use it to explore the state of the base system or apply desired changes. After applying a change, metasystem uses saved metadata to propagate change back to the base system. Fig. 1 describes basic structure and behavior of the approach.

After applying a change, the metasystem (with use of monitoring aspect) is used to reflect changes in the metamodel. Updated metamodel can be again changed using the same described mechanism (Fig. 2).

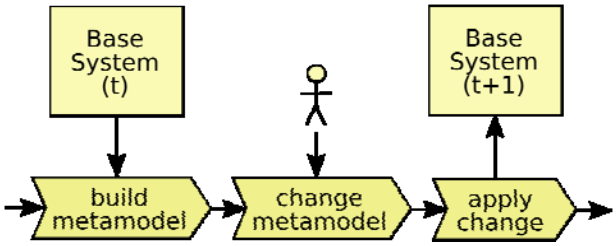

Fig. 2: Base system and metamodel change in time

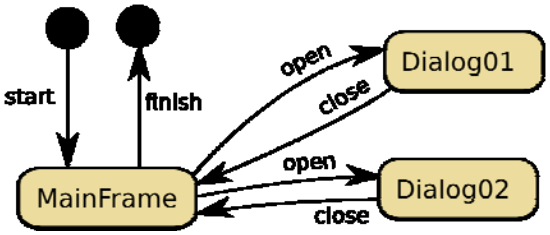

Fig. 3: State diagram example, representing application model

IV. STATE DIAGRAM – AN ABSTRACT MODEL OF THE APPLICATION FLOW

Certain applications with graphics user interface can be represented as a state diagram. Each node of the diagram represents dialog or window of user interface, transitions represent possible user actions (e.g. click on a button event, dialog close event). Fig.3 shows an example of the state diagram of such application.

A. Architecture

A base system is extended with monitoring code by AspectJ weaving. AOP technique allows using this approach without the need of base system modifications. Weaved aspect collects desired data and process it in the metasystem (Fig.4).

The metasystem functionality can be accessed remotely via RMI (Remote Method Invocation) calls. RMI is a Java API for performing the object equivalent of remote procedure calls. On the remote JVM (Java Virtual Machine) a RMI server must be created.

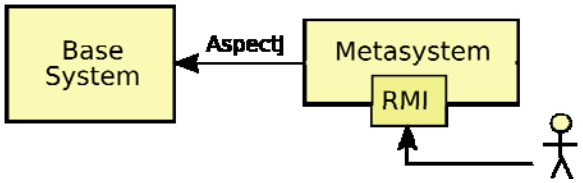

Fig. 4: Architecture

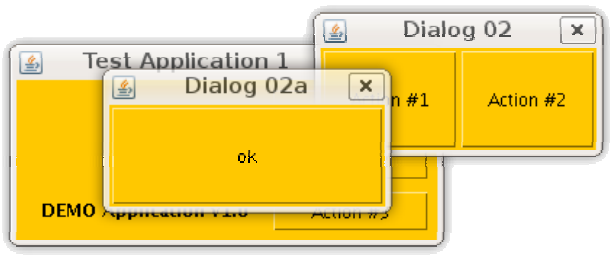

Fig. 5: Dialog examples of the application model

In our approach, a RMI server is created in the metasystem. With help of control client it is possible to communicate with metasystem and to get information about metamodel or to give orders to modify it.

B. *Application model*

The application model was developed using the AWT (Abstract Windowing Toolkit). The AWT is platform-independent windowing, graphics, and user-interface widget toolkit, which is part of the Java Foundation Classes (JFC). Fig.5 shows an example of the application model (main application frame and two opened dialogs).

Observed objects were possible transitions between dialogs of the application. The assumption was made that transitions can be invoked by pressing a button or by closing a dialog.

Pressing on AWT button invokes the method *actionPerformed* in the class implementing *ActionListener*. According to the pressed button a new dialog is opened. Closing a dialog is implemented by invoking method *setVisible(false)*.

C. *Metamodel creation*

The metamodel is created by keeping track of user's actions in application model. The monitoring code (watching from/to state transitions and building state diagram) is weaved using standard AspectJ implementation.

Tracking transitions between states (dialogs) in application model is implemented with aspects crosscutting the base application in the following pointcuts:

•Invocation of a method *actionPerformed (ActionEvent event)*. This event is recorded as an invocation of an action (start of a transition).

•Invocation of a method *setVisible(boolean visible)*. If set to true, new dialog (state) is opened. This state is coupled with the latest event (action) invoked. If set to false, flow is moved to the previous state.

After user clicks on a dialog button, a method *actionPerformed(ActionEvent event)* is invoked. Weaved aspect code records the component which invoked the event (in this case *java.awt.Button*). Following method *setVisible(boolean visible)* with a parameter set to *true* is used to show a dialog. Within this method, weaved aspect code records a transition to a new or an existing state. This state is coupled with the component which invoked the event (*java.awt.Button*). If this combination already exists, the flow simply moves to an already existing state. If it doesn't exist, a new state is created and the flow is moved to this state. Important part of a state data is a reference to a graphical component representing it (*java.awt.Frame* or *java.awt.Dialog* object). It is used later to allow application model modifications.

When user closes a dialog, method *setVisible(Boolean visible)* is used with a parameter of value *false*. According to this parameter, weaved aspect code recognizes action of closing a dialog and moves the flow in state diagram to previous state.

All aspect activity is delegated to *MetadataManager* (Fig.6). *MetadataManager* is creating a state diagram which is represented as a graph of custom *State* classes. Each *State* class contains a list of states it is connected to and a reference to a graphical component representing it. Each connection (transition) is coupled with invoking component. Each *State* class also records its previous state - to have the possibility of return (when closing a dialog).

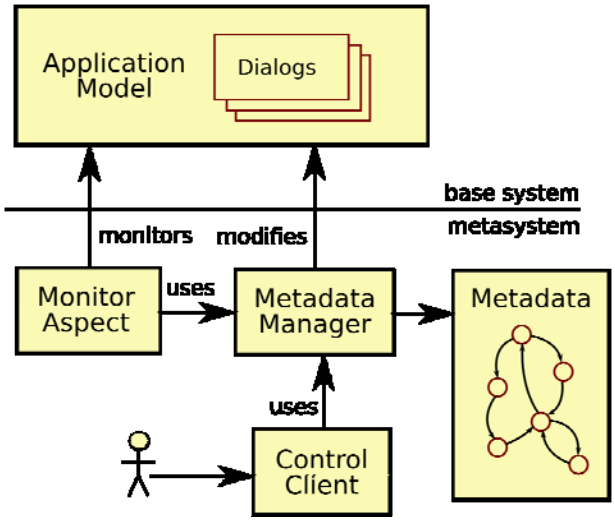

Fig. 6: Main components

D. *Metamodel modification*

At the moment there are three basic possibilities to work with metamodel:

- Print whole state diagram.
- Remove state.
- Create new state.

While creating a state diagram, each state gets unique number identificator. The action of printing state diagram displays all transitions for each state (states are represented by their number identificators).

Removing state removes both - object *State* representation from metamodel (state diagram) and also associated graphical component from application model. Other possibility would be to clean the whole state diagram and let the metasystem to build a new one (already without removed state). The current implementation is more effective.

Creating a new state requires following additional information:

- Id of the source state. The new state will be accessible from this state.
- Position of button opening a new dialog.
- Label of button opening a new dialog.
- Implementation class of a new dialog.

When creating a new state, a graphical component representation of source state is obtained (it is stored in source state metadata). It is used as a container for a new *java.awt.Button* component. The *java.awt.Button*

component is created at the specified position and with the specified label. The component action is set to create a new instance of specified implementation class of a new dialog. After creating the instance, it is displayed (by invoking a method *setVisible(true)*). Since the button action corresponds to the *actionPerformed* method, after it will be used it will be automatically tracked by the monitoring aspect. The new state metadata will be created by the same mechanism as described above.

Since adding new states uses Java Reflection, it is possible to extend application with any (proper) classes from the classpath.

V. CONCLUSION

In this paper, we confirmed the possibility of application properties monitoring, metamodel creation, metamodel modification and the possibility of propagation the change back to the application model.

The woven aspect tracks specific application properties and instructs metadata manager how to construct a metamodel represented as a state diagram. The state diagram can be modified - specified state can be removed or a new state can be created. Metamodel modifications affect application model and change its behavior. Changed application model remains monitored and the corresponding metamodel (state diagram) is updated. There is no difference between original code and the new code added during modifications. The same monitor/modify technique is used for both - original code and added code.

Described example uses aspects only to monitor target application model. All modifications are applied to original application model structures. This can lead to many unexpected situations (e.g. *ConcurrentModificationException* while modifying a collection). Another topic for research is the possibility of making modifications by using dynamic AOP.

Some modifications are done only in the application model (and then automatically tracked by monitor aspect and transferred to metamodel), while some are more effective when done in both - metamodel and application model.

Since the used weaving method is not capable to weave in system classes, it was impossible to hook the method *setVisible* directly in classes *java.awt.Frame* and *java.awt.Dialog*. Instead two helper classes *MyFrame* and *MyDialog* were created. These two classes are inheriting mentioned system classes and overriding just *setVisible* method. The overriden method just calls identical method in the parent.

With use of more advanced weaving method it would be possible to apply this technique also to applications with no source code available.

Another weakness is impossibility of using this technique in already running application. To weave aspect code it is needed to prepare aspects already before loading affected classes.

This very simple example works only with a specific kind of application. To have working solution for more generic application model, more complex assumptions are required.

REFERENCES

[1] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in Software Evolution", In Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE'05), pp. 13-22, 2005.

[2] J. Kollár, J. Porubän, P. Václavík, J. Bandáková, and M. Forgáč: "Software Evolution From A Meta-Level Compiler Perspective", SCIENCE & MILITARY, 2, 2, 2007, pp. 29–32.

[3] E. Althammer, "Reflection patterns in the context of object and component technology", University of Konstanz, [PhD thesis], 2002.

[4] E. Tanter, "From Metaobject Protocols to Versatile Kernels for Aspect Oriented Programming", University of Nantes, France, and University of Chile, Chile, [PhD thesis], 2002.

[5] D. Rothlisberger, "Geppetto: Enhancing Smalltalk's Reflective Capabilities with Unanticipated Reflection", Masterarbeit der Philosophisch-naturwissenschaftlichen Fakultat der Universitat Bern, 2005.

[6] D. da Cruz, M. Bern, P.R. Henriques, M.J.V. Pereira, "Strategies for Program Inspection an Vizualization", In: Proceedings of CSE 2008 International Scientific Conference on Computer Science and Engineering, Sep.24-26, 2008, High-Tatras, Slovakia, pp. 107–117.

[7] D. Zmaranda, G. Gabor, "Software Environment for Task oriented Design of Real Time Systems", In: Proceedings of CSE 2008, International Scientific Conference on Computer Science and Engineering, Sep. 24-26, 2008, High-Tatras, Slovakia, pp. 359–366.

[8] D. Zmaranda, G. Gabor, M. Gligor, "A Framework for Modeling and Evaluating Timing Behaviour for Real-Time Systems", In: Proc SINTES 12 Int. Symposium on Systems Theory, Oct. 20-22, University of Craiova, Romania, 2005, pp. 514–520.

[9] M. Bebjak, V. Vranic, P. Dolog, "Evolution of Web Applications with Aspect-Oriented Design Patterns", In: Marco Brambilla and Emilia Mendes, editors, Proc. of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with 7th International Conference on Web Engineering, ICWE 2007, July 19, 2007, Como, Italy, pp. 80–86.

[10] M. Lehman, F. Ramil, "Towards a theory of software evolution and its practical impact", Invited Lecture. In: Proc. Intl. Symp. on Principles of Software Evolution, 2000, 2–11.

[11] J. Dowling, V. Cahill, "Dynamic Software Evolution and The KComponent Model",Workshop on Software Evolution, OOPSLA, Tampa, Florida, USA, Oct. 2001.

[12] J. Porubän, P. Václavík, "Generating Software Language Parser from Domain Classes", In: Proceedings of CSE 2008 International Scientific Conference on Computer Science and Engineering, The High Tatras - Stará Lesná, Slovakia, Sep. 24-26, 2008, pp. 133-140.

[13] P. Václavík, and J. Porubän, "Template-based content management system, AEI 2008 International Conference on Applied Electrical Engineering and Informatics", Athens, Greece, September 8-11, 2008, pp. 153-157.