ADAPTIVENESS OF SOFTWARE SYSTEMS USING REFLECTION

Ján KOLLÁR, Michal FORGÁČ, Jaroslav PORUBÄN
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice,
E-mail: Jan.Kollar@tuke.sk, Michal.Forgac@tuke.sk, Jaroslav.Poruban@tuke.sk

SUMMARY

This paper deals with basic principles of metaprogramming and reflection with connection to aspect-oriented programming (AOP). Metaprogramming is about writing programs that represents and manipulate other programs or themselves, i.e. metaprograms are programs about programs. The impact of metaprogramming is obvious in traditional development processes, by sorting existing programs as transformational processes with inputs and outputs. Open implementation and metalevel architectures are related to their reflective properties. Reflection is an entity's integral ability to represent, operate on and otherwise deal with itself in the same way that it represents, operates on, and deal with its primary subject matter. Structural reflection represents the ability of a program to access a representation of its structure, as it is defined in the programming language. Behavioural reflection represents the ability of a program to access a dynamic representation of itself, that is to say, of the operational execution of the program as it is defined by the programming language implementation (processor). AOP allows development of required application using principle of separation of concerns. Reflection and AOP share many similarities in concepts, possibilities and applied techniques. There are several solutions to provide a reflective system among which belong following approaches: MetaclassTalk, Geppetto Reflex and Iguana/J. The former two are systems based on Smalltalk - Squeak; the latter two are based on Java.

Keywords: *complex software systems, open implementations, metaprogramming, reflection, aspect-oriented programming*

1. INTRODUCTION

There are many ways how to create complex software systems. According to the area of software engineering there are five general steps: collecting requirements, assignment of specification, design, implementation and evolution. When software developers create an initial version of a software application, which includes all requirements in this time needed, they do not have knowledge about future requirements. The new addition, change or removal of functionality requires additional costs. In fact, changes needed to satisfy new requirements take several times longer than initial design and realization. In order to have evolution of created complex system affordable efficiency it is important to use appropriate methods for effective evolution which depend on previous steps and also adequate environment for running complex system.

Implementation of complex systems can be expressed in two ways: black-box abstraction or open implementation [18]. Black-box represents solution in which desired module of a system can expose its functionality but its implementation is hidden. Open implementation allows changing or altering parts of the underlying software to enable required needs.

Efficiency of creation and maintenance of complex systems is influenced also according to used implementation languages. Sometimes only one general-purpose programming language is sufficient.

But in any cases there is useful to exploit one or more domain-specific languages, because they are designated for specific kind of tasks.

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

Aspect-oriented programming [5, 7, 10, 17] supports means for separation of crosscutting concerns. Base functionality can be expressed by some base-level language and crosscutting concerns can be expressed by one or more domain-specific languages. Weaving [7, 10] is utilized after proposal of individual parts. Computer system has to be located in environment, which enable run-time changes. We are thinking that adaptiveness can be achieved also using adaptive aspect-oriented language [8]. Run-time adaptability of aspect-oriented language can be one of the solutions which can help in software evolution [4, 9, 11], but there are some obstacles to easy creation of this solution. Therefore, when somebody is creating complex system, it has to pass through many implementation points. Employed language or more languages should be minimal and strongly associated with the properties of the software system in any point of its implementation [8], thus there is the need to change language when it is useful.

This paper is structured as follows: basic information about metaprogramming and groups of programs which can be considered according to utilization of metaprogramming are presented in section 2. Section 3 deals with main principles of reflection and utilization of reflection in aspect-oriented programming. Section 4 presents some practical reflective solutions. Finally, section 5 presents conclusion of this paper.

2. METAPROGRAMMING

Metaprogramming is about writing programs that represents and manipulate other programs or themselves, i.e. metaprograms are programs about programs [3]. The impact of metaprogramming is obvious in traditional development processes, by sorting existing programs as transformational processes with inputs and outputs. According to [15] there are some groups of programs which can be considered according to utilization of metaprogramming. The notation n → p means, that n is the sort of programs of whose inputs are code, and p is the sort of programs of whose outputs are also code. These groups are as follows:

- 0 → 0: doesn't take or return code (non-meta program).
- 1 → 0: takes one piece of input code, doesn't output code (interpreter, code analyzer).
- 0 → 1: outputs code without inputting any (data precompiler, generator).
- 1 → 1: takes one input program, outputs one back (compiler)
- 2 → 0: takes two programs as input, doesn't output code (metainterpreter).
- 2 → 1: takes two input program, returns one program (metacompiler).
- 1 → 2: takes one program as input, returns two (phase splitter).

For adaptive systems, the most interesting groups are 2 → 0 and 2 → 1. But in general, inputs may represent more than two programs. These programs can be expressed not only in any general-purpose programming language but also in various domain specific languages. Thus these groups can be generalized and written as:

- n → 0: takes n programs as input (n>1), doesn't output code (metainterpreter).
- n → 1: takes n input programs (n>1), returns one program (metacompiler).

Another important issue is temporal facet, because in case of metainterpreters, individual input programs can be inserted and processed at arbitrary time. Typical examples are interpreters which support run-time weaving [9]. Thus input programs can alter behaviour of interpreted functionality.

3. REFLECTION

Open implementation and metalevel architectures are related to their reflective properties. Reflection is an entity's integral ability to represent, operate on and otherwise deal with itself in the same way that it represents, operates on, and deals with its primary subject matter [3]. A metalevel provides information about selected system and makes the software self-aware. A base level includes the application logic.

In general, in any reflective system, the meta-level control over the base level takes place in two steps [12]:

- The base object (in case of object-oriented programming) calls the metaobject requesting a change in terms of semantics. This is called the reification of one implementation or semantic aspect.

- After this is done, the flow of control returns from the metaobject back to the base object. Because the metaobject modified a part of the base object, its behaviour or representation is now changed. This process is also called reflecting the changes back into the base object.

The reification operation (Fig. 1) then consists in expliciting some base concepts or mechanisms that are usually transparent for the programmer. Those reified concepts or mechanisms are usually implemented in some metaobjects (at the metalevel). In a functional point of view, the reification process occurs when the base level gives hand to the metalevel.

The reflection operation (Fig. 1) consists in modifying the base level interpretation to change the base level objects semantics. In a functional point of view, the reflection process occurs when the metalevel processes some extra computing and gives back the hand to the base level.

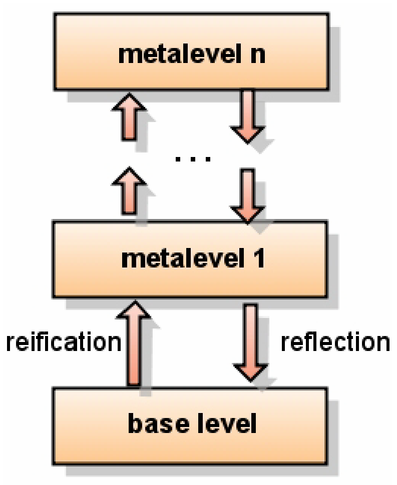

Fig. 1 The reification and reflection operation.

Thus reflection in wider meaning consists of reflection point of view and reification point of view. In the text below we will mean under reflection both views together.

Reflection can be divided into two groups: structural and behavioural reflection [18]:

Structural reflection represents the ability of a program to access a representation of its structure, as it is defined in the programming language. For instance, in an object-oriented language, structural reflection gives access to the classes in the program as well as their defined members.

Behavioural reflection represents the ability of a program to access a dynamic representation of itself, that is to say, of the operational execution of the program as it is defined by the programming language implementation (processor). In an object-oriented language, behavioural reflection could for instance give access to base-level operations such as method calls, field accesses, as well as the state of the execution stack of the various threads in the program.

3.1. Reflection for interpreted and compiled languages

A programming language is said to be reflective if it provides an explicit representation (i.e., reification) of entities that either represent program building blocks (e.g., classes, methods) or are involved in program execution (e.g., stack, garbage collector) [18]. Developers thus can define system (software) functionalities and also new program building blocks or execution mechanisms (how functionalities will be performed). Using reflection and metaobject protocol [1] ist the most appropriate for interpreted languages. An interpreter is the ideal place for metalevel information about running program. This support is not present in two situations:

- for compiled languages, where source code is turned into code directly executed by the machine, since there is no interpreter (such as C++),
- for interpreter languages whose standard interpreter is non-reflective and hardly extensible or modifiable (like Java virtual machines).

When there is the need to add support for reflection into non-reflective compiled language, it is necessary to keep metalevel information beyond the compilation process and perform transformation of source code with appropriate links to the metalevel information. This proposal can be supported through so-called hooks, which have to be introduced into transformed code. Hooks are pieces of code and allow the reification process, because they trigger shift to the metalevel when they are reached by the execution flow. Metalevel consist of behaviour, which can be accessed and changed dynamically. But significant disadvantage is significant execution overhead, if hooks are used at each and every place in the code, because program must evaluate both the hooks and the metalevel code. This problem solves partial reflection [18], which uses limited set of hooks.

In the case of languages, for which only interpreter without reflective support is available, there are two options. The first one is to add the interception and redirection mechanisms in the source or binary code, similarly as in the case of compiled languages. Thus interpreter will be without changes, but similar major disadvantage performance overhead remains. The second option represents advantage of direct access to internal structure of interpreter and thus it supports greater flexibility and expressiveness for supporting dynamic adaptation. The major disadvantages are the loss of compatibility with standard environments and the complexity of the implementations.

3.2. Aspect-oriented programming using reflection

Aspect-oriented programming [5, 7, 10, 17] allows development of required application using principle of separation of concerns. Reflection and AOP share many similarities in concepts, possibilities and applied techniques [1].

A concern is a particular goal, concept, or area of interest; it means that it is in substance semantical concern. From the structural point of view a concern may appear in source code as a component or as an aspect [7]. A component is cleanly encapsulated in building block of the programming language; it is structurally compact (core) concern. Aspect is a property which crosscuts components and tends to affect components performance and semantics. Using reflection, aspect code is separated from base code using the natural separation between the base level and the metalevel. Base code is defined at the base level, while aspects are defined at the metalevel. Base objects represent base code, and metaobjects represent aspects. Separating aspect definitions one from another is done by making use of a specific set of metaobjects for each aspect. [1].

Separated crosscutting concerns in traditional AOP are mutually woven with components through weaving process, which can be performed during compile-time, load-time or run-time. Our interest is mainly in run-time weaving. Using reflection, run-time weaving can be performed using the meta-link and meta-object cooperation. Aspects are thus woven with base code using the meta-link.

4. EXISTING REFLECTIVE SOLUTIONS

There are several different solutions to provide a reflective system. In the following list, the former two approaches are systems based on Smalltalk – Squeak; the latter two are based on Java. Squeak is open source full-featured implementation of Smalltalk programming language and environment [2, 16].

MetaclassTalk [2] represents a reflective extension of Smalltalk language for simplifying experiments with new programming paradigms (e.g. aspect-oriented programming). Smalltalk language [6] by itself supports reflective facilities, thus it can be understood as independent reflective solution (such as Smalltalk-80 in [16]). Although Smalltalk has many reflective facilities, they provide little help for changing the execution mechanisms. Last version of MetaclassTalk does not extend the virtual machine of Squeak in order to provide advanced reflective functionality, but it is established on extending of the compiler. MetaclassTalk thus requires the source code to introduce its reflective capabilities into the system. This solution is portable between different Squeak images.

Geppeto [16] is based on Squeak and works on principle of insertion hooks into bytecode of methods. Hooks are placed in every method, where we want to reify required operations. Hook insertion can occur at any time in any method of any class in the whole running system, even in system classes. This solution allows portability, because it doesn't modify virtual machine.

Reflex [18] is open reflective system for Java. It is based on insertion of hooks into bytecode thus it does not modify Java virtual machine, only transforms bytecode. Hooks reify information about a base level operation and pass this information to the metalevel. But hooks are inserted only at load-time, because Java platform has technical limitations and thus only anticipated reflection is possible. This solution is portable, because it does not modify Java virtual machine.

Iguana/J [14] is also reflective framework for Java. It is based on bytecode transformation and it requires adapted and extended Java virtual machine. This proposal modifies the interpreter by making use of JIT compiler interface. This solution is not portable, but it supports unanticipated reflection.

5. CONCLUSION

In this paper we have presented basic principles of metaprogramming and reflection with connection to aspect-oriented programming. We have mentioned about groups of programs, which can be considered according to utilization of metaprogramming and we have generalized this overview with respect to utilization of various inputs languages in metainterpreters and metacompilers. We have also summarized information about existing reflective solutions.

Our current research concentrates on how a language (not a program) can vary its semantics, reflecting not just compile time, as it is in [13] but also runtime events. This work have brought theoretical basis about concept of metaprogramming and reflection, which we have recognised as fundamental.

REFERENCES

[1] Bouraqadi, N., Ledoux T.: Aspect-oriented programming using reflection. Technical Report 2002-10-3, Ecole des Mines de Douai, October 2002.

[2] Bouraqadi, N.: Concern Oriented Programming using Reflection. In Workshop on Advanced Separation of Concerns – OOSPLA 2000, 2000.

[3] Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison Wesley (2005), 832 pp.

[4] Ebraert, P., Tourwe, T.: A Reflective Approach to Dynamic Software Evolution, Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution(RAM-SE'04), 15th of June 2004, Oslo Norway, 2004, pp. 37-43.

[5] Filman, R.: Friedman, D.: Aspect-oriented programming is quantification and obliviousness, Workshop on Advanced Separation of Concerns (OOPSLA 2000), October 2000.

[6] Goldberg, A., Robson, D.: Smalltalk-80: The Language. Addison Wesley, 1989.

[7] Kiczales, G., et al.: Aspect-Oriented Programming. 11th European Conf. on Object-Oriented Programming, volume 1241 of LNCS, Springer Verlag, 1997, pp. 220-242.

[8] Kollár Ján, Porubän Jaroslav, Václavík Peter, Bandáková Jana, Forgáč Michal: Adaptive Language Approach to Software Systems Evolution, International Multiconference on Computer Science and Information Technology: 1st Workshop on Advances in Programming Languages (WAPL'07), Wisla, Poland, October 15-17, Polish Information Processing Society, 2007, 2, pp. 1081-1091, ISSN 1896-7094 .

[9] Lehman, M., Ramil, J.: Towards a theory of software evolution - and its practical impact. Proceedings of the International Symposium on Principles of Software Evolution, Nov. 2000, Japan, pp. 2-11.

[10] Nicoara, A., Alonso, G.: Dynamic AOP with PROSE. In: Proceedings of International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005) in conjunction with the 17th Conference on Advanced Information Systems Engineering (CAISE 2005), Porto, Portugal, June 2005.

[11] Oriol, M.: An Approach to the Dynamic Evolution of Software Systems, Ph.D. Thesis, University of Geneva, Geneva, Switzerland, April 2004.

[12] Pawlak, R.: Metaobject Protocols For Distributed Programming. Technical report, Laboratoire CNAM-CEDRIC, Paris, 1998.

[13] Rebernak, D., Mernik, M. Rangel H. P., and Pereire M.J.V.. Aspectlisa: an aspect-oriented compiler construction system based on attribute grammars. In LDTA'06: 6th Workshop on Language Descriptions, Tools and Applications, Vienna, AT 2006.

[14] Redmond, B., Cahill, V.: Supporting Unanticipated Dynamic Adaptation of Application Behaviour. In Proceedings of European Conference on Object-Oriented Programming, volume 2374, Springer-Verlag, 2002, pp. 205–230.

[15] Rideau, F.: Metaprogramming and Free Availability of Sources, Two Challenges for Computing Today, CNET DTL/ASR, 1999.

[16] Röthlisberger, D.: Geppetto: Enhancing Smalltalk's Reflective Capabilities with Unanticipated Reflection, PhD thesis, University of Bern, December 2005.

[17] Steimann, F.: The paradoxical success of aspect-oriented programming, in: OOPSLA '06, Portland, Oregon, USA, 2006, pp. 481–497

[18] Tanter, E.: From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming PhD thesis, University of Nantes, France, and University of Chile, Chile. November 2004.

BIOGRAPHIES

Ján Kollár was born in 1954. He received his MSc. summa cum laude in 1978 and his PhD. in Computing Science in 1991. In 1978-1981, he was with the Institute of Electrical Machines in Košice. In 1982-1991, he was with the Institute of Computer Science at the University of P.J. Šafárik in Košice. Since 1992, he is with the Department of Computers and Informatics at the Technical University of Košice. In 1985, he spent 3 months in the Joint Institute of Nuclear Research in Dubna, Soviet Union. In 1990, he spent 2 month at the Department of Computer Science at Reading University, Great Britain. He was involved in the research projects dealing with the real-time systems, the design of (micro) programming languages, image processing and remote sensing, the dataflow systems, and the implementation of functional programming languages. Currently the subject of his research are adaptive languages and software systems.

Michal Forgáč was born in 1983. In 2006 he graduated at Technical university of Košice. He is working on his PhD. degree at the Department of Computers and Informatics FEEI, Technical university of Košice. His scientific research is focusing on the aspect oriented programming paradigm, software evolution and adaptiveness of complex software systems.

Jaroslav Porubän was born in 1977. He received his MSc. summa cum laude in 2000 and his PhD. in Computing Science in 2004. Since 2003 he is with the Department of Computers and Informatics at Technical University of Košice. He was involved in the research projects dealing with implementation of functional programming languages and parallel programming. Currently the subject of his research is the application of process functional paradigm in aspect oriented programming and program profiling systems.