Metaobject Protocol as a Tool for Language Evolution

Ľubomír Wassermann and Ján Kollár
Department of Computers and Informatics
Technical University of Košice, Faculty of Electrical Engineering and Informatics
Letná 9, 042 00 Košice, Slovakia
Email: {Lubomir.Wassermann, Jan.Kollar}@tuke.sk

***Abstract* – This paper deals with evolution of software languages. Metaobject protocol (MOP) technology is suitable for support of evolution of language. The main goal of designing and developing MOP for software language is to achieve its extensibility. Evolution can be realized by the call of appropriate MOP operation. Very important aspect of language evolution is to preserve compatibility and consistency. To achieve this goal, each evolution step needs to be properly documented to keep track of all the changes made to the language. Transformation of language will be based on its description generated and realized by MOP and its operations. MOP operations alter language processor to conform to changes made to the language. Alteration of language processor itself can be done with help of reflection mechanisms.**

***Keywords:* language evolution, metaobject protocols, metaprogramming, program transformation, reflection.**

I. INTRODUCTION

Nowadays, software development is challenging due to two main factors - complexity and change [7]. Software products during their lifecycle are coping with change. Change causes software product to evolve. But to maintain quality of software product, evolution should be controlled in some way with help of language mechanisms and constructs. Traditionally, languages have been designed to be viewed as set of black box abstractions [9]. End programmers have only part or no control at all over the semantics or implementation of these abstractions. This point of view is related to misconception that software languages are immutable [5]. Programs are dependent on language, in which they are written, and tools which this language offers (e.g. interpreters, compilers, etc.). When we admit that program is subject to evolution and is tightly coupled with language in which is written we can assume that language can be subject to evolution, too.

Important aspect of language evolution is to preserve compatibility of programs written in different versions of language. To achieve this, each evolution step need to be properly documented to keep track of all the changes made to the language. "Opening up" language abstractions and implementation could be one way how to achieve language evolution.

But as stated in [9], this opening should not expose unnecessary details and thus overwhelm programmer. Only the essential structure of the implementation should be exposed. Providing an open implementation can be advantageous in a wide range of high-level languages. Metaobject protocol (MOP) technology is a powerful tool for providing this [9]. The main goal of designing and developing MOP for language is to achieve its extensibility. Thus well-designed MOP can serve as a tool for language evolution.

II. METAPROGRAMMING AND METASYSTEMS

As the prefix "meta" is suggesting (meta = being about), metaprogramming is writing programs that represent and manipulate other programs (or programs that write programs). The most common metasystems processing tool is a compiler [13]. To define metasystem we can come out from definition of computational system. According to [17] computational system is a system that acts and reasons about a domain. With this definition we can define metasystem as a system whose domain is another computational system. The computational system which acts as domain of a metasystem is called its base system [24]. Due to fact that every change in domain is reflected in its computational system (causal connection) we can see that every change in computational system is reflected in metasystem reasoning about that computational system, and vice versa. The computational system operates at domain level. It contains application logic represented by domain objects. Metasystem operates at meta-level. It contains system metaobjects, which describe, control or modify domain objects. Metaobject is an object which reflects the structural, and possibly also the behavioral aspect of a single object. Meta-level provides information about selected system and makes the software self-aware.

We can easily introduce reflection to all these definitions when we regard the fact that metasystem is computional system, too. Then computational system which reasons and acts about itself is called reflective system. Reflective program is a program describing a system that accesses its own metasystem [24].

A. *Reflection*

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. Reflection is used as one of the tools of metaprogramming. It is important aspect of relationship between domain object and its metaobject. This relationship allows domain object to ask for the services of metaobject and metaobject to change domain object implementation. Meta-level control over domain level takes part in two steps (Fig. 1) [19], [6]:

- Domain object calls metaobject requesting change in term of semantics. This is called reification. Reification is process of making concrete an implicit aspect of an object that can be changed by the metaobject.
- Flow of control is returned back to domain object. Because the metaobject modified a part of domain object, its behavior and/or structure is now changed. This process is called reflection - reflecting the changes back to domain object.

As defined in [18], a reflective mechanism is any tool made available to a program *P* written in a language *L* that either reifies the code of *P* or some aspect of *L*, or allows *P* to perform some reflective computation. When system reifies some parts of program, there is variety of possible actions over these reifications. Here comes to difference between reflective mechanisms - introspection and intercession.

Introspection is the ability of a program to simply reason about reifications of otherwise implicit aspects of itself or of the programming language implementation (processor). In analogy with file systems, introspection can be seen as a read access to reifications. Introspection play significant role in program visualization [4] and in reflective environments for real-time systems [27], [28]. Without ability for introspection, aspect oriented approach to softvare development would fail [2].

Intercession is the ability of a program to actually act upon reifications of otherwise implicit aspects of itself or of the programming language implementation (processor). Following the same analogy, intercession corresponds to a write access to reifications [24].

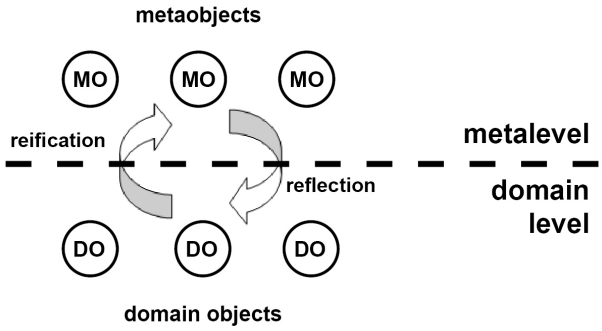

Fig. 1. Processes of reflection

B. *Metaobject Protocols*

Metaobject protocol (MOP) can be described as an interface through which domain objects and metaobjects are communicating. This interface can be seen as standard interface between objects but transposed in the area of reflection and metaprogramming. This interface allows for independent development of domain system and metasystem.

According to definition stated in [10], Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behavior and mplementation, as well as the ability to write programs within the language. This definition was later refined and describes three principles of how MOPs work:

1) The basic elements of the programming language (classes, methods and generic functions) are made accessible as objects. They are given the special name of metaobjects because these objects represent fragments of a program.
2) Individual decisions about the behavior of the language are encoded in a protocol operating on these metaobjects - MOP.
3) For each kind of metaobjects, a default class is created, which lays down the behavior of the default language in the form of methods in the protocol.

C. *Methodology of MOP design*

When we are designing MOP, we can look at it as we were designing a programming language. (comparison is depicted in Fig. 2). Language designers are working with two different levels of design process at the same time - the level of designing particular programs in terms of a given language, and the level of designing the language to support the lower-level design processes.

MOP design is similar, with the addition of one more level in design process. Again, designer is considering programs that could be written in language but is not thinking about only one language but a whole range of languages that can be expressed with designed MOP.

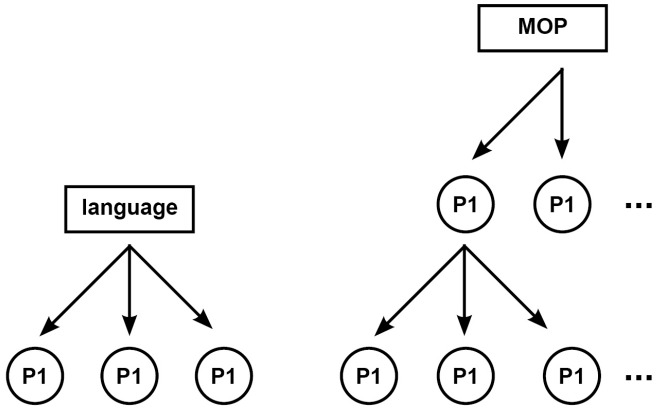

Fig. 2. Language design vs. MOP design

III. EVOLVING LANGUAGE

Well designed language MOP can be used to modify the language and its behavior. We assume that MOP is suitable tool for support of the evolution of language. Particular MOP operations can change different aspects of language (Fig. 3). But as mentioned before, to avoid inconsistency, evolution of the language should be controlled. Imagine that company has developed systems that use domain specific language (DSL) designed by them. New requirements on system can cause the need to cover more concepts from given domain. This necessary leads to extension of system to cover new area of given domain.

To be able to express new concepts in DSL language, it is possible that language would have to be extended by new constructs and/or concepts, or altering existing ones. But due to fact that DSL language is used not only by one system, change in the language can affect functionality of other systems. To keep the consistency, each evolution step should be documented and transformation of language described to be able to map previous version of language to new version of language (e.g. mapping constructs, elements, keywords, etc.).

If there is a need for change of the language, appropriate MOP operation can be called. The result will be language transformation and its description. This description can be used as documentation of versioning of system or language itself. Whole transformation can be performed at metalevel. Change of language can be achieved by reflective mechanisms of introspection and intercession [26].

Imagine we have our own DSL language used to write programs which serve as input for our system from medical domain. Programs written in DSL language are input for language processor (compiler or interpreter) which processes them. At metalevel is metasystem with our designed MOP protocol with set of operations that can be used to transform our DSL language. Domain system and metasystem are written in Java.

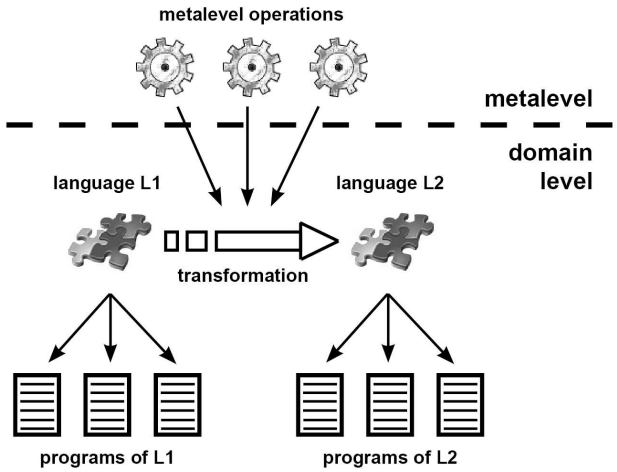

Fig. 3. Language Evolution Scheme

If we want to transform our language to achieve that existing concept Pill will be renamed to Medicine and will contain new property Type, appropriate MOP operations are called and description of the transformation is generated (Fig. 4). Transformation of language will be based on its description in XML file [25]. Metasystem then alters language processor to accept new or modified concepts.

Alteration of language processor itself can be done with help of some Java reflection extension or framework. For example, reflective extension JavAssist enables reification and alteration of existing classes and creation of new classes. JavAssist and its functionality is based on bytecode transformation [1].

Language processor is transformed in two ways. First, as mentioned before, language processor have to conform to new language syntax, to be able to process programs in new version of language, processing new concepts and constructs or altered concepts [21].

Second, language processor should be able to process programs written in the previous version of language. This program is transformed and processed according to description of the transformation of language. The reason why we stressed the necessity of description of transformations and documenting them as part of system (or language) versioning is the situation when we need to know the history of language evolution steps and their transformations. Program is then processed and transformed according to history of evolution steps.

This way the language processor would be able to process not only programs of recent version of language. Integrity and funcionality of systems that share same DSL language which is evolving through the time will not be compromised. Controlled evolution of language will ensure us better coping with change of software systems.

IV. RELATED WORKS

Automatic software construction by programs without excessive manual work of programmers is one of software engineering visions [7], [8]. Although modeling, prototyping, exploiting design and architectonic patterns are currently used in practice, no general theory, methodology and corresponding technology supporting automatic software production exists. Despite the fact that UML action semantics was published in 2001 [23], in contrast to well known methodology of the translation for programming languages, no common methodology for model transformation is available [3].

Advances in domain specific languages [15] as well as in software language engineering methods [11], [12], [14], [16], are inspirational to generalize intentional programming methodology [22] as well approaches in semantic web [20]. The aim for a new metaprogramming paradigm is an extremely actual theme of current research. This paper is just a small step in the research focusing to the analysis of available mechanisms for language evolution in object oriented paradigm.

V. CONCLUSION

In this paper we have presented our approach to a language evolution with the use of a metaobject protocol. We were concentrating on the preservation of consistency and compatibility between programs written in different "versions" of evolving language. Each evolution step and its language transformation is properly described. This description of transformation is an input for the transformation of language processor to accept new or modified concepts. Description of the transformation is also used for mapping between concepts of previous version of language with concepts of new version.

ACKNOWLEDGMENTS

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

REFERENCES

[1] E. Althammer: Reflection patterns in the context of object and component technology, University of Konstanz, PhD thesis, 2002, 120 pp.

[2] M. Bebjak, V. Vranić, and 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.

[3] K. Czarnecki, and S. Helsen: Feature-based survey of model transformation approaches, IBM SYSTEMS JOURNAL, VOL 45, NO 3, 2006, pp. 621–645.

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

[5] J.M. Favre: Languages evolve too! Changing the Software Time Scale, Proceedings of the Eighth International Workshop on Principles of Software Evolution, 2005, pp.33–44.

[6] D. Friedman and M. Wand: Reification: Refection without metaphysics, In Proceedings of the Annual ACM Symposium on Lisp and Functional Programming, August 1984, pp. 348–355.

[7] J. Greenfield, K. Short, S. Cook, and S. Kent: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley, 2004, 500 pp.

[8] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In Proceedings 2nd Working IEEE / IFIP Conference on Software Architecture (WICSA), 2001, pp. 45–54.

[9] G. Kiczales, J. Rivieres, and D. Bobrow: The Art of the Metaobject Protocol, MIT Press, 1991, 236 pp.

[10] G. Kiczales et al.: Metaobject Protocols: Why We Want Them, and What Else They Can Do In Object-Oriented Programming The CLOS Prospective, Andreas Paepcke, Ed., MIT Press, Cambridge, MA, 1993, pp. 101–118.

[11] P. Klint, R. Lämmel, C. Verhoef: Toward an Engineering Discipline for Grammarware, ACM Transactions on Software Engineering and Methodology, Vol.14, No. 3, July 2005, pp. 331–380.

[12] Kleppe, Anneke: A Language Description is More than a Metamodel. In the 4th International Workshop on (Software) Language Engineering. 2007.

[13] 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.

[14] R. Lämmel: Grammar Adaptation, In Proc. Formal Methods Europe (FME'01), volume 2021 of LNCS, Springer-Verlag, 2001, pp. 550–570.

[15] Lochmann, Henrik, Braeuer, Matthias: Towards Semantic Integration of Multiple Domain-Specific Languages Using Ontological Foundations. In the 4th International Workshop on (Software) Language Engineering. 2007, pp. 34–46.

[16] Martin von Löwis, Marcus Denker, Oscar Nierstrasz: Context-oriented programming: beyond layers, Proceedings of the 2007 international conference on Dynamic languages, ACM International Conference Proceeding Series; Vol. 286, Lugano, Switzerland, pp. 143–156

[17] P. Maes: Computional reflection, Ph.D. thesis, Artificial intelligence laboratory, Vrije Universiteit, Brussels, Belgium, 1987, 112 pp.

[18] J. Malenfant, M. Jacques, and F. Demers: A tutorial on behavioral reflection and its implementation, In Metaobject Protocols, Kiczales, Ed., MIT Press, Cambridge, MA, 1996, pp. 1–20.

[19] R. Pawlak: Metaobject Protocols For Distributed Programming, Technical report, Laboratoire CNAM-CEDRIC, Paris, 1998, 15 pp.

[20] Pareiras, Fernando Silva, Staab, Steffen, Winter, Andreas: On Marrying Ontological and Metamodeling Technical, ACM, 2007, p. 439–448

[21] J. Porubän and P. Václavík: Generating Software Language Parser from Domain Classes, 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.

[22] Simonyi, C.: The death of computer languages, the birth of intentional programming, 1995.

[23] Gerson Sunyé, Wai-ming Ho, Alain Le Guennec and Jean-marc Jézéquel: Using UML Action Semantics for executable modeling and beyond, Proceedings of the 13th Conference on Advanced Information Systems Engineering, 2001, Springer, pp. 433–447.

[24] E. Tanter: From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming, PhD thesis, University of Nantes, France, and University of Chile, Chile. November 2004, 104 pp.

[25] 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.

[26] C. Zimmermann: Advances in Object-Oriented Metalevel Architectures and Reflection, CRC Press, 1996, 336 pp.

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

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