A NOTE ON ASPECT, ASPECT-ORIENTED AND DOMAIN-SPECIFIC LANGUAGES

Marjan MERNIK, Tomaž KOSAR, Viljem ŽUMER
Institute of Computer Science, Faculty of Electrical Engineering and Computer Science, University of Maribor,
Smetanova 17, 2000 Maribor, Slovenia, tel. (+386 2) 220 7455
E-mail: marjan.mernik@uni-mb.si, tomaz.kosar@uni-mb.si, zumer@uni-mb.si

SUMMARY

Aspect-oriented programming is a programming technique for modularizing concerns that crosscut the basic functionality of programs. In aspect-oriented programming, aspect languages are used to describe properties which crosscut basic functionality in a clean and a modular way. In the paper aspect languages are discussed and compared with domain-specific languages, mainly from the implementation point of view. The difference between aspect languages and aspect-oriented languages is also pointed out. To show that existing domain-specific language implementation approaches can be used also for aspect(-oriented) language implementation the aspect-oriented language AspectCOOL has been designed and implemented with the compiler/interpreter generator tool LISA.

Keywords: *aspect language, domain-specific aspect language, general-purpose aspect language, aspect-oriented language, domain-specific language, general-purpose programming language*

1. INTRODUCTION

A programming language is the basic software engineering tool used to build a software system. It can greatly increase the programmer's productivity by allowing him to write a high-scalable, generic, readable and maintainable code. In this regard, a domain-specific language (DSL), which is a programming language for solving problems in a particular domain and provides built-in abstractions and notations for that domain, is by no means an exception. Domain-specific languages [13, 40, 45] are usually small, more declarative than imperative, and more attractive than general-purpose languages for a variety of applications because of easier programming, systematic reuse, easier verification. Domain-specific languages have been used in various domains such as graphics, financial products, and 3D animation. These applications have clearly illustrated the advantages of domain-specific languages over general-purpose languages in areas such as productivity, reliability, maintainability and flexibility.

Recently, aspect-oriented programming (AOP) [23] aiming to support a separation of concerns attracts many researchers [17, 18, 21, 29, 31, 33, 37]. In AOP, aspect languages are used to describe properties that crosscut basic functionality in a clean and a modular way. So far, many aspect languages have been proposed, designed and implemented. Here, many interesting questions arise:

- Is an aspect language a general-purpose or a domain-specific language?
- Can aspect language implementation benefit from implementation approaches known for domain-specific languages?
- Where is the difference between aspect languages and other domain-specific languages?
- Are compiler-generator tools dedicated to the development of domain-specific languages suitable also for development of aspect languages?

In this paper our point of view and answers to these questions are given. The organization of the paper is as follows. In section 2, aspect-oriented programming is briefly described. The comparison of the implementation of domain-specific and aspect languages is given in section 3, followed by design and implementation of our aspect-oriented language AspectCOOL in section 4. Related work is described in section 5. Finally, the conclusion is given in section 6.

2. ASPECT-ORIENTED PROGRAMMING

The major abstraction technique in software engineering is to divide the system into functional components in a such manner that changes to a particular component do not propagate through the entire system. However some issues, called aspects, are system wide and cannot be put into a single functional component. Failure handling, persistence, communication, coordination, memory management, and many others, are aspects of a system behavior that tend to cut-across groups of functional components. As a consequence, a code of functional components is tangled with aspect code. This tangling problem makes functional components less reusable, difficult to develop, understand and evolve. The problem could be avoided if we could code these aspects in separate modules and afterwards weave them into functional components by an aspect weaver. One of the first questions to which AOP had to answer was: ”Are current generalized procedure languages[1] [23] suitable for the description of aspects?” It was argued [37] that generalized

[1] Abstraction mechanisms in generalized procedure languages are subroutines, procedures, functions, objects, classes, and modules.

procedure languages do not provide the right abstraction for the description of aspects and that aspect languages are needed for the expression of aspects. An aspect language is a language whose abstractions can directly represent one or more aspects. Due to the well known classification of programming languages to general-purpose and domain-specific languages, the next question is one of the first which appear in our minds: ”Are aspect languages general-purpose or domain-specific programming languages?”. With general-purpose programming languages we can solve problems in many application domains such as: numeric computations, business processing, simbolic processing, while a domain-specific language is a programming language for solving problems in a particular domain and provides built-in abstractions and notations for that domain. Aspect languages provide language mechanisms that explicitly capture the crosscutting structure. With aspect languages only aspects can be expressed, and for this reason aspect languages are surely domain-specific languages. This is in line with the clear separation of component (base) languages and aspect languages in AOP [23]. Component languages are used to capture the basic functionality of the system, which is a result of the system's functional decomposition. These components have clearly defined the structure and responsibilities. Since AOP extends the already existing paradigms, component language can be any of the general-purpose programming languages belonging to different programming paradigms [47], such as procedural, functional, object-oriented, logic, process functional [25, 26]. Indeed, currently in AOP, component languages are procedural [21], functional [12, 30], process functional [28], logic [14], and object-oriented languages [5, 8, 6, 16, 44]. Moreover, there is no reason for a component language not to be also a domain-specific language (e.g., spreadsheet language). However, the clear separation of component languages and aspect languages has shortcomings. A desire is to integrate an aspect language into a component language in a seamless way using new constructs and new semantics. An aspect-oriented language is a programming language where component language is extended with an aspect parts in a seamless way. In such language, aspect constructs are nicely integrated into component language and these aspect parts are not declared as a separate aspect language with special name (e.g., AspectJ is an aspect-oriented language where object-oriented language Java has been extended with aspect constructs; AspectJ is a proper extension of Java, any valid Java program is also valid AspectJ program). Therefore, it is important to distinguish between aspect-oriented languages and aspect languages.

Aspects implement additional properties of the system which crosscut the basic functionality of the system. In order to achieve the desired properties of the system, we need an aspect weaver that combines the component and the aspect language by weaving advices at appropriate join points and may involve merging components, modifying them, optimizing, and so on. In [32] three ways of combining aspects and components were distinguished: juxtapose, merge, and fuse. There are several possible approaches to weaving which can be done by a pre-processor, during compilation, by a post-compile processor, at load or run-time, or using some combination of these approaches. Since there are many types of aspects, such as synchronization, distribution, persistence, debugging, another question is interesting: ”Do we need an aspect language for every type of aspect?”. Both answers are possible. Therefore, aspect languages are further divided into general-purpose aspect languages where different aspects are described in the same language, and domain-specific aspect languages where we need an aspect language for every type of aspect. With a domain-specific aspect language we can not support aspects other than they were designed for. The COOL and RIDL languages [33] are one of the first domain-specific aspect languages. General-purpose aspect languages are usually integrated in a seamless way into general-purpose programming languages forming aspect-oriented languages (e.g., AspectJ [24]). Note that we classified all aspect languages (but not aspect-oriented languages) as domain-specific languages! Here, the terminology is misguiding. When we talk about aspect languages terms "general-purpose" and "domain-specific" refer to the ability of a language to express certain kinds of aspects (general aspects vs. specific aspects), while these terms in a broader sense of programming languages refer to the ability of a language to express different kinds of problems (general problem vs. specific problem). General-purpose and domain-specific aspect languages have their own advantages and disadvantages (Table 1). An advantage of a general-purpose aspect language is at the same time a disadvantage of a domain-specific aspect language and vice versa.

An important feature of an aspect language is that it has to work together with the component language. When an aspect language is developed a component language is usually already known, designed and implemented. In that case the benefits are:

- we do not have to implement a component language,
- the component language is already known to programmers, and
- programmers just learn a new aspect language.

On the other hand, there are also some disadvantages:

- extensions and restrictions of the component language are difficult,
- there is no access to the component compiler, and
- design of join points is not completely free.

One of the main disadvantages of domain-specific aspect languages is that we need a new aspect language for each new aspect.

	general-purpose aspect languages	domain-specific aspect languages
advantages	• it is easier to learn one general-purpose aspect language than many domain-specific aspect languages; • it is more likely to be accepted by programmers.	• the aspect code can be more concise and easier to understand; • its syntax can be tailored to the specific aspect.
dis-advanatges	• low level of abstraction for aspect description; • can not guarantee separation of concerns [36].	• the introduction of new aspects increase the number of aspect languages; • hard to interact with other tools (debugger, editor, ...).

Table 1 Advantages and disadvantages of aspect languages

3. IMPLEMENTATION OF DSLS AND ASPECT LANGUAGES

Design and implementation of domain-specific languages, which aspect languages are, is itself a significant software engineering task, requiring a considerable investment of time and resources. Therefore, it is an obvious question if current implementation approaches of domain-specific languages are also suitable for aspect languages and where is the main difference between them.

Domain-specific languages are programming languages for solving problems in a particular domain and provide built-in abstractions and notations for that domain. Despite that conventional libraries of procedures/classes can introduce new abstractions for describing specific problems, although they are not suitable when the domain requires:

- syntax or/and semantic modifications,
- domain-specific optimizations, and
- domain-specific error checking and reporting.

Domain-specific languages are for many applications more attractive than general-purpose languages because of easier programming, systematic reuse, better productivity, reliability, maintainability, and flexibility. However, the benefits of domain-specific languages are not for free. Since the cost of domain-specific language design, development and maintenance has to be taken into account, one of the main questions is "When and how to develop a domain-specific language?" [40]. If we want to improve the productivity, reliability, reusability or enable end user programming in some narrow, but well defined domain, then the domain-specific language might be a solution and an answer to the first part of this question. The development of a domain-specific language usually includes the following phases: analysis, design, implementation and finally their use. In the analysis phase, the problem domain is identified and the domain knowledge has to be gathered; the domain-specific language is designed so as to concisely describe applications in the domain. The implementation phase can be mainly done using one of the following approaches [40]:

- Compiler/interpreter: Developing a compiler or an interpeter for a whole language from scratch and maintaining it is a difficult task. Standard compiler tools (e.g., lex, yacc [1]), or tools dedicated to the implementation of domain-specific languages (e.g., ASF+SDF [9], LISA [39], SmartTools [2]) can be of great help. The weakness of this approach is non-interoperability with other language processing tools (editors, debuggers, etc);
- Embedded language: An existing language is extended by user-defined operators which build a library of domain-specific operations. The advantages of this approach are easy implementation and programming features are inherited from the host language, while the weaknesses are the inability to express domain-specific optimizations, domain-specific error reporting, and domain-specific syntax.
- Preprocessor: With this approach new constructs are translated to statements in the base language by a preprocessor. The advantage of preprocessors is that the development cost of new constructs is much smaller than the cost of developing a whole compiler for the base language. Unfortunately, this advantage is also a disadvantage, since domain-specific error reporting and domain-specific optimizations cannot be performed.
- Extensible compiler/interpreter: A compiler or an interpreter is extended with new constructs; this is usually done by a reflection mechanism. Reflective languages have built-in language extension capabilities which allow us to modify the default semantics of the language [22]. However, extensibility of syntax is slightly restricted in these systems.

These general approaches for implementation of domain-specific languages are also appropriate for the implementation of aspect languages. It is interesting to see which approaches were used for

the implementation of some aspect(-oriented) languages (Table 2). We can notice that many aspect(-oriented) languages have been implemented by preprocessors. The reason is that static weaving is easily performed by preprocessors. On the other hand, aspect(-oriented) languages with dynamic weaving are usually implemented by a reflection mechanism.

compiler/interpreter	AspectJ ver 0.8 [24] AspectCOOL [3]
embedded language	TyRuBa [14]
preprocessor	D [33] AspectJ ver 0.6 [34] D^2AL [5] C++ Framework [15] TEA [6] Malaj [11] $TinyC^2$ [49]
extensible compiler/interpreter (reflection)	AOP/ST [8] Luthier MOP [42] JST [44]

Table 2 Implementation aproaches for aspect(-oriented) languages

We should also point to an important difference between aspect languages and DSLs. Namely, aspect languages influence the semantics of component languages, and their implementation has to interact with the implementation of component languages. This feature makes aspect language implementation harder. Every aspect language defines certain places, called joint points, where it is possible to associate aspect behavior. An important design decision for aspect languages is how to quantify places, pointcut designators, that need modifications. Pointcut sublanguage should be expressive enough to able to identify such points easily on type and instance level [43]. Even more important is how to define generic, reusable and comprehensible pointcuts that are not tightly coupled to an application's structure [27, 46]. In [46] a notation of inductively generated pointcuts was proposed as a solution to this problem.

4. AspectCOOL

To answer the last question stated in the introduction, namely "Are compiler-generator tools dedicated to the development of domain-specific languages suitable also for development of aspect languages?" an aspect-oriented language AspectCOOL has been implemented [3] with the compiler/interpreter tool LISA [39].

We have a lot of experience with the compiler/interpreter approach of domain-specific language implementation. In our opinion, the advantages of the formal definitions of general-purpose languages should be exploited, taking into consideration the special nature of domain-specific languages. An appropriate methodology that considers frequent changes of domain-specific languages is needed since the language development process should be supported by modularity and abstraction in a manner that allows incremental changes as easily as possible [41]. If incremental language development is not supported, then the language designer has to design languages from scratch or by scavenging old specifications. To be productive, the development of these languages has to be based on high-level automated tools [20]. Our tool LISA supports this methodology. It is a compiler/interpreter generator based on a special kind of attribute grammars [38] that supports incremental development with multiple attribute grammar inheritance and templates. Multiple attribute grammar inheritance is a structural organization of attribute grammars where the attribute grammar inherits the specifications from ancestor attribute grammars, may add new specifications, and may override some specifications from ancestor specifications. With inheritance the lexical, syntax and semantic parts of programming language specification can be extended. These features make the tool LISA very appropriate for the development of domain-specific languages. In this section the design of aspect-oriented language AspectCOOL is breifly described. More details on implementation can be found in [3]. AspectCOOL is an extension of the class-based object-oriented language COOL (Classroom Object-Oriented Language)[2], which have been designed and implemented simultaneously with AspectCOOL. Both languages were formally specified with multiple attribute grammar inheritance, which enables us to gradually extend the languages with new features and to reuse the previously defined specifications.

The COOL language is an object-oriented language used for studying object-oriented language design and implementation. In addition to features usually found in class based object-oriented languages it has the following features:

- Dynamic loading: Modules must be compiled separately. Each component is compiled in a separate object file. Loading of modules is done on demand.
- Method call interception: To support AOP we must be able to intercept method calls with all parameters from the caller and the callee.
- Easy environment transportation: Local environments inside methods should be easily transported to the aspects and accessible outside the local method environment.
- Reflection: Using the class Class we can gather information about join points. They contain information about classes, instance variables, methods and slots. Using reflection at method call interception we can also gather information about the caller and the callee.

[2] Do not confuse this general-purpose programming language with the specific aspect language COOL developed by C. Lopes as a part of her Ph.D. thesis [33].

- Dynamic referencing of the super class: Almost all object-oriented languages compile super as a static reference, which is calculated at compilation. The static approach can be problematic when implementing the calls to super classes inside the aspect.
- Slots: Methods may have explicit join points also called slots. Slots must also be accessible through reflection.

In the design of aspect-oriented language AspectCOOL we want to explore different AOP concepts and ideas such as:

- separation of advices and join points,
- explicit join points named slots,
- caller to callee method call transitions as join points,
- separate compilation of aspect and component code, and
- applying aspects to components where the source code is not available.

To achieve reusability of aspects [7] the aspect program is divided into two parts: aspect classes and the aspect application part. Each aspect class may consist of advices and slot advices. The former are applied as actions on join points, while the latter are applied on slots in the component. Considering the time of application, we have four types of advices similar to [31]: before, enter, exit and after. Before and enter advices are actions executed before the execution of the actual method. Exit and after advices are executed after the execution of the method. On the other hand, before and after advices are executed in the callers' environment while enter and exit advices are executed in the environment of the executed method.

When an advice is declared, the class on which the advice can be applied has to be provided. This is very important in the context of separate compilation because we have to perform a static analysis of the code in order to compile an advice. Each advice has a list of formal arguments, which are used for communication between an advice and the join points on which it is applied. In the context of separate compilation, method calls in components are already compiled in such way that the caller expects the return value of some type. In case that the type of return value is modified by an advice, it can only be modified to one of the subclasses of the original return type. Without this limitation we might come to the situation where the method on which an advice is applied, returns a value incompatible with the type expected by the method caller, resulting in runtime error.

The aspect application part of the program in AspectCOOL is the part where advices and slot advices are connected to concrete join points. In AspectCOOL we decided to describe the join point as a method call from one method to another. This method call with corresponding methods is also called transition [31] and it supports the jumping aspect code [10]. It is obvious that in this case the join point is specified with a pair of caller and callee methods. Since advices are usually applied on a set of join points, a mechanism for the description of such a *transition set* is also needed. Wildcards can also be used for the description of a transition set. Besides the transition set, the aspect, which is applied, and its application time must be specified.

The basic approach used for aspect weaving in the AspectCOOL language is a method call interception. At the moment when the method call is intercepted we have the possibility to perform additional operations before and after the call of the original method. In the process of aspect weaving the main role is played by the Aspect Manager. The Aspect Manager is a component responsible for the registration of advices on join points. When the method call takes place it is intercepted. The control flow is transferred from the caller to the method call interception. First, "before" advices are executed. The control flow is passed to the Aspect Manager, which then executes advices registered for that join point. After that, the control is passed back to the method call interception and the process is repeated for "enter" advices. Next, the method is executed followed by "exit" and "after" advices. After the execution of each type of advice, the control flow is transferred from the aspect manager back to the method call interception because the appropriate environment for each type of advice has to be assembled. For "enter" and "exit" advices the environment is bound to the callee object and for "before" and "after" advices to the caller object. For "exit" and "after" advices the return value also has to be added to the environment. The important issue here is also the optimization of the method call interception and advice execution. Since this approach of method interception for each method call introduces additional operations, this overhead has to be minimized as much as possible [4].

5. RELATED WORK

We strongly believe that the development of domain-specific languages should not differ much from the design and implementation of general-purpose languages, as suggested in [45]. Therefore, any tool that generates a compiler or an interpreter from formal language specifications can be used for efficient and rapid development of domain-specific languages. Until now, various tools based on different formal methods have been used for domain-specific language development.

To explore different aspect-oriented programming concepts and ideas, AspectCOOL has been designed and implemented. Some of these ideas have already been proposed in the literature. In [7] the author suggests that aspect description (aspect class) should be divided into two parts, where advices and join points are separately defined. Advice is a method-like construct where the semantics of the aspect is described. Join points are

well-defined points in the execution of the program where advices are executed. With this approach the advice part is potentially much more reusable. Unfortunatelly, this was just a proposition and the implementation has not yet begun.

The benefits of applying aspects to Java Commercial Off-the-Shelf (COTS) software is described in [48]. Since the source code of COTS software is not available or compiled classes may not be available before they are loaded into the Java Virtual Machine, the aspects have to be applied to already compiled classes. In [48] class loading is intercepted and the class bytecode is rewritten before the class is instantiated by the Java Virtual Machine. In our approach bytecode rewriting is not used since we have full access to the component compiler and instead of class loading interception the method calls are intercepted.

In the paper [19] the distinction of aspect-oriented languages and other programming languages is discussed through the studying of interpreters. Key elements to achieve aspect behaviors are redefinition of the set and lookup operators of the conventional interpreters.

Implementation of aspect languages through code instrumentation techniques [49] and partial evaluation as weaving [35] can be also seen as particular pre-processing approaches.

6. CONCLUSION

In the paper aspect languages are considered as domain-specific languages. The main feature of aspect languages, which differentiates them from other domain-specific languages, is their influence on the semantics of the component language. They have to work together with a component language using join points and weaving. We have also shown that compiler-generator tools dedicated to the development of domain-specific languages are suitable for development of aspect(-oriented) languages.

REFERENCES

[1] A.V. Aho, R. Sethi, J. Ullman. Compilers, Principles, Techniques, and Tools. Reading, MA, Addison-Wesley, 1986.

[2] I. Attalli, C. Courbis, P. Degenne, A. Fau, D. Parigot, C. Pasquier. SmartTools: A Generator of Interactive Environment Tools. 10th International Conference on Compiler Construction, CC'2001, LNCS, vol. 2027, pp. 355 - 360, 2001.

[3] E. Avdičaušević, M. Lenič, M. Mernik, V. Žumer. An experiment in design and implementation of aspect-oriented language. ACM Sigplan Notices, Vol. 36, No. 12, pp. 84 - 94, 2001.

[4] E. Avdičaušević, M. Lenič, M. Mernik, V. Žumer. Experimental aspect-oriented language AspectCOOL. Proceedings of 17th ACM symposium on applied computing, SAC 2002, pp. 943-947, 2002.

[5] U. Becker. D^2AL: A design-based aspect language for distribution control. Position paper. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98, 1998.

[6] F. Bergenti, A. Poggi. Aspect Views as a Means to Promote Reuse in Aspect-Oriented Languages. Position paper. Proceedings of the ECOOP'2000 Workshop on Aspects and Dimensions of Concerns, 2000.

[7] A. Beugnard. How to make aspects re-usable, a proposition. Position paper. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, 1999.

[8] K. Bollert. On Weaving Aspects. Position paper. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, 1999.

[9] M. van den Brand, A. Van Deursen, J. Heering, H. A. De Jong, M. de Jong, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, J. Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. 10th International Conference on Compiler Construction, CC'01, LNCS, vol. 2027, pp. 365 - 370, 2001.

[10] J. Brichau, W. de Meuter, K. de Volder. Jumping Aspects. Position paper. Proceedings of the ECOOP'2000 Workshop on Aspects and Dimensions of Concerns, 2000.

[11] G. Cugola, C. Ghezzi, and M. Monga. Malaj: A Proposal to Eliminate Clashes Between Aspect-Oriented and Object-Oriented Programming. In Proceeding of 16th IFIP World Computer Congress, International Conference on Software: Theory and Practice, WCC2000, 2000.

[12] W. De Meuter. Monads as a theoretical foundation for AOP. Position paper. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97, 1997.

[13] A. van Deursen, P. Klint, J. Visser. Domain-Specific Languages: An Annotated Bibliography. ACM Sigplan Notices, Vol. 35, No. 6, pp. 26 - 36, 2000.

[14] K. De Volder. Aspect-Oriented Logic Meta Programming. Position paper. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98, 1998.

[15] L. Dominick. Aspect of Life-Cycle Control in a C++ Framework. Position paper. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, 1999.

[16] E. Ernst. Separation of Concerns and Then What?. Position paper. Proceedings of the ECOOP'2000 Workshop on Aspects and Dimensions of Concerns, 2000.

[17] R. E. Filman, D. P. Friedman. Aspect-oriented programming is quantification and

obliviousness. In workshop on Advanced Separation of Concerns, OOPSLA, 2000.

[18] R. E. Filman. What is aspect-oriented programming, revisited. In Workshop on Advanced Separation of Concerns, ECOOP, 2001.

[19] R. E. Filman. Understanding AOP through the Study of Interpreters. In AOSD Workshop on Foundations of Aspect-Oriented Languages, FOAL‘03, 2003.

[20] J. Heering, P. Klint. Semantics of Programming Languages: A Tool-Oriented Approach. ACM Sigplan Notices, Vol. 35, No. 3, pp. 39 - 48, 2000.

[21] J. Irwin, J. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, T. Shpeisman. Aspect-Oriented Programming of Sparse Matrix Code. Proceedings of International Scientific Computing in Object-Oriented Parallel Environments, ISCOPE'97, LNCS, vol. 1343, 1997.

[22] G. Kiczales, J. Des Riveres, D. G. Bobrow. The Art of the Metaobject Protocol. MIT Press, 1991.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. Aspect-Oriented Programming. 11th European Conference on Object-Oriented Programming, ECOOP'97, LNCS, vol. 1241, pp. 220-242, 1997.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. An Overview of AspectJ. ECOOP'01, LNCS, vol. 2072, pp. 327-355, 2001.

[25] J. Kollar. Process Functional Programming. Proc. ISM’99, Rožnov pod Radhoštem, Czech Republic, April 27-29, pp. 41 - 48, 1999.

[26] J. Kollar. PFL expressions for Imperative Control Structures. Proc. Scient. Conf. CEI’99, Herl’any, Slovakia, October 14-15, pp. 23 - 28, 1999.

[27] J. Kollar, V. Novitzka. Static Weaving at Dynamic Joint Points. Acta Electrotechnica et Informatica, No. 1, Vol. 4, pp. 16 - 23, 2004.

[28] J. Kollar. Process Functional Properties and Aspect Language. Acta Electrotechnica et Informatica, No. 2, Vol. 4, pp. 25 - 32, 2004.

[29] J. Kollar. Poincut designators in an aspect oriented language. Acta Electrotechnica et Informatica, No. 3, Vol. 4, pp. 13 - 20, 2004.

[30] R. Laemmel, G. Riedewald, W. Lohmann. Adaptation of functional object programs. Position paper. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP’99, 1999.

[31] R. Laemmel. A Semantical Approach to Method-Call Interception. Proceedings of the 1st International Conference on Aspect-Oriented Software Development, AOSD 2002, pp. 41 – 55, 2002.

[32] J. Lamping. The interaction of components and aspects. Position paper. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97, 1997.

[33] C. Lopes. D: A Language Framework for Distributed Programming. PhD thesis, Northeastern University, 1997.

[34] C. Lopes, G. Kiczales. Recent Developments in AspectJ. Position paper. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98, 1998.

[35] H. Masuhara, G. Kiczales, C. Dutchyn. Compilation semantics of aspect-oriented programs. In AOSD Workshop on Foundations of Aspect-Oriented Languages, FOAL‘02, pp. 17 – 26, 2002.

[36] K. Mehner, A. Wegner. Assessment of Aspect Language Design. Position paper at Young Researchers Workshop, GCSE’99, 1999.

[37] K. Mens, C. Lopes, B. Tekinerdogan, G. Kiczales. Aspect-oriented Programming Workshop Report. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97, 1997.

[38] M. Mernik, V. Žumer, M. Lenič, E. Avdičaušević. Implementation of multiple attribute grammar inheritance in the tool LISA. ACM Sigplan Notices, vol. 34, no. 6, pp. 68-75, 1999.

[39] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer. LISA: An Interactive Environment for Programming Language Development. 11th International Conference on Compiler Construction, CC’02, LNCS, vol. 2304, pp. 1-4, 2002.

[40] M. Mernik, J. Heering, T. Sloane. When and How to Develop Domain-Specific Languages. CWI Technical Report SEN-E0309, 2003.

[41] M. Mernik, V. Žumer. Incremental Programming Language Development. Computer Languages, Systems and Structures, No. 31, pp. 1-16, 2005.

[42] J. Pryor, N. Bastan. A Reflective Architecture for the Support of Aspect Oriented Programming in Smalltalk. Position paper. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, 1999.

[43] H. Rajan, K. Sullivan. Need for Instance Level Aspect Language with Rich Pointcut Language. In AOSD Workshop on Software Engineering Properties of Languages for Aspect Technologies, SPLAT‘03, 2003.

[44] L. Seinturier. JST: An Object Synchronization Aspect for Java. Position paper. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, 1999.

[45] D. Spinellis. Notable design patterns for domain-specific languages. The Journal of Systems and Software, No. 56, pp. 91 - 99, 2001.

[46] T. Tourwe, A. Kellens, W. Vanderperren, F. Vannieuwenhuyse. Inductively Generated Pointcuts to Suppport Refactoring to Aspects. In AOSD Workshop on Software Engineering Properties of Languages for Aspect Technologies, SPLAT'04, 2004.

[47] D. A. Watt. Programming Language Concepts and Paradigms. Prentice-Hall, 1991.

[48] I. Welch, R. Stroud. Load-time Application of Aspects to Java COTS Software. Position paper. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, 1999.

[49] C. Zhang, H.-A. Jacobsen. TinyC^2: Towards building a dynamic weaving aspect language for C. In AOSD Workshop on Foundations of Aspect-Oriented Languages, FOAL'03, 2003.

BIOGRAPHY

Marjan Mernik received his M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently an associate professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He was a visiting professor in the Department of Computer and Information Sciences at the University of Alabama at Birmingham in 2004. His research interests include principles, paradigms, design and implementation of programming languages, compilers, formal methods for programming language description and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Tomaž Kosar received the BSc degree in computer science at the University of Maribor, Slovenia in 2002. He is currently a young researcher at the University of Maribor, Faculty of Electrical Engineering and Computer Science. His research for PhD degree is concerned with design and implementation of domain-specific languages. His research interest in computer science include also domain-specific visual languages, compilers, refactoring, and unit testing. He is a student member of the IEEE.

Viljem Žumer received his M.Sc. degree in computer science from the University of Ljubljana in 1977 and his Ph.D. degree in computer science from the University of Maribor in 1983. He is currently a professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is the founder of the Computer Science Department at the University of Maribor and in charge of many projects. His research interests include computer architecture and programming languages. He is a member of the IEEE.