Versatile Language Semantics with Reflective Embedding

Tom Dinkelaker

Technische Universität Darmstadt
Hochschulstr. 10, 64289 Darmstadt, Germany
dinkelaker@informatik.tu-darmstadt.de

Abstract

Often, for one programming language, various implementations exist that have subtle but important variations in their syntax and semantics. While current technology provides good support for syntax variability in form of syntax extensions, there is only limited support for semantic variability in language implementations. My thesis is about a novel approach for adaptable language implementations that uses a meta-object protocol to embed language abstractions into the host language and that uses reflective techniques to adapt the language implementation. The techniques developed in my thesis open up several possibilities for adaptation in language implementations not addressed by related work. Similarly to the role of a meta-object protocol in general-purpose languages for adapting object-oriented abstractions, the meta-object protocol allows to adapt language abstractions of advanced language features, such as the abstractions of aspect-oriented programming and domain abstractions in domain-specific languages.

Categories and Subject Descriptors D.3.3 [*Software Engineering*]: Language Constructs and Features—Classes and Objects, Frameworks

General Terms Design, Languages

Keywords Embedded Domain-Specific Languages, Meta-Object Protocols, Aspect-Oriented Programming

1. Motivation

Often, for one programming language, various implementations exist that have subtle but important variations in their syntax and semantics. For example, there is a myriad of *domain-specific languages* (DSLs) for state machines, or there are various implementations of the *aspect-oriented programming* (AOP) [5] paradigm for the same base language. Variations in languages exist because of the diversity in applications. Actually in every problem domain or programming paradigm, there can be a dis-consensus on the right syntax and semantics.

In traditional language development, for every new syntax and semantics, a new language is implemented from scratch resulting in high development costs. Current language development approaches target to improve these costs, e.g., using *extensible compilers or run-times* as well as *component-based*, *model-driven*, or *generative language development*. While current technology provides good support for syntax variability in form of syntax extensions, there is only limited support for semantic variability in language implementations. In general, meta protocols are not available for most languages, particularly for new languages, such as DSLs.

DSLs facilitate writing programs in a certain application domain by providing direct means to express domain-specific abstractions and operations. Although using a DSL results in more declarative code, DSL programs may also suffer from tangling and scattering in the presence of domain-specific crosscutting concerns. To improve modularization in DSL programs, aspect-oriented features could be used to modularize crosscutting concerns. But, while AOP infrastructures exist for most general-purpose languages, AOP is in general not available for arbitrary DSLs, since aspects must be composed in a special way.

Embedded domain-specific languages [4] have been proposed to avoid the high development costs for DSLs. Roughly, an embedded language is implemented as a library in a hosting language. On the one hand, a new language can be implemented without implementing a new parser and compiler. And on the other hand, the most host language's features are reused for the embedded language. Thus, the development cost is significantly reduced and new language features can incrementally be added. While it is commonly agreed that embedded DSLs are the fasted way to implement a DSL, it remains an open question whether this approach can be used to embed advanced language features, such as aspects.

The motivation of this thesis is to combine the research in embedded DSLs with the research of advanced language features, such as *reflective programming* [7, 6] and *aspect-*

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.
ACM 978-1-60558-768-4/09/10.

oriented programming [5]. By using reflective programming in DSLs and aspect-oriented languages, we would like to leverage the same flexibility that is provided by reflective programming for general-purpose languages to these languages. By using DSLs in reflective programming, we would like to leverage domain-specific reflective languages that allow more declarative and therefore allow robust adaptations.

In the following, the research hypothesis is summarized in Sec. 2, the plans for an evaluation are presented in Sec. 3, and finally, Sec. 4 reports future work.

2. Thesis

My thesis is:

> *Adopting reflective techniques for embedding languages enables languages that are open for syntactical and semantics extensions at the application level, and open for composability.*

My research combines the approaches of *reflective programming* [7, 6] with embedding languages [4]. The approach is therefore called *reflective embedding*. The approach uses reflective features of the host language in form of a *meta-object protocol* (MOP) [6] that is used to enable open syntax, semantics, and compositions of languages. Reflective embedding is demonstrated in the *Groovy* language, but it could also be implemented in other languages providing similar reflective features, such as *Ruby* or CLOS.

To enable an open syntax, reflective embedding uses the MOP to embed language abstractions into the host language. A language is decomposed into language components. A language component consists of (1.) an interface that defines syntax abstractions (i.e., the keywords), (2.) an implementation of this interface that binds the syntax abstractions to (3.) a language model that provides the concrete semantics. These artifacts are packed together as a library in the host language. In the language components, each of the above three parts can be replaced to adapt the implementation. At runtime, the MOP implicitly maps the language abstractions used in programs to method calls on meta-level classes in the language model.

To enable open semantics, reflective embedding uses reflective features to extend the implementations of existing language abstractions for alternative semantics. For example, alternative language semantics can enable an interpretation that provides certain guarantees, an optimized execution, or other semantic analyzes of the same program. In a nutshell, reflection is used to extend the language semantics by overriding the methods that are called on the language model classes for executing the semantics.

To enable open compositions, reflective embedding uses reflective features to compose several language components. In a nutshell, extensible composition operators are provided for composing syntax and semantics of language components as if the composed parts would have been implemented as one language component. Internally, in the composition operators, the MOP delegates the usage of a keyword in a program that is given in the composed syntax to the right language component. Different styles of composition are supported. A language component can be extended using an inheritance mechanism. Several components can be composed using black-box composition and gray-box composition.

3. Evaluation

For a qualitative evaluation, the flexibility of the concept is validated by solving existing problems in four case studies.

Case Study 1: Versatile Semantics for DSLs. The semantics of a given DSL is subject to evolution and to disconsensus on the right semantics. For example, a state machine can be realized either as a Moore or a Mealy automaton. Similarly, the execution semantics of UML state machines has several variations [1], such as various implementation strategies for transition selection and event consumption. Despite this, semantics variations in DSL implementations have only been little explored by means of extensible DSL compilers and interpreters. The need for supporting application-specific semantic variability for DSLs is comparable to the need for adapting the semantics of a *general-purpose language implementation* in a particular application targeted by meta-object protocols [6].

Reflective embedding enables meta-level architectures for DSLs. In a nutshell, while a language designer implements a DSL as a library using reflective embedding, another language designer in the user domain can adapt the language implementation using the MOP. The adaptability is enabled because the language is embedded in a host language and the MOP of the host language makes the language abstraction adaptable. We instantiate the approach by building a DSL for state machines that supports variations in state machine semantics that have been demanded previously in literature [1] but that are currently not supported in related DSL approaches.

Case Study 2: Versatile Semantics for AOP. Reflective embedding can be used to embed abstractions of a programming paradigm, such as AOP. Additionally when using reflective embedding, the paradigm semantics stays open for extensions. A problem with the existing AOP technology is that alternative semantics for aspect-oriented abstractions can be defined only by compiler experts using extensible aspect compiler frameworks and infrastructure. Application developers are prevented from tailoring the language semantics in an application-specific manner.

To address this problem, in [2], we present a new architecture for aspect-oriented languages with an explicit meta-interface to language semantics, called a *meta-aspect protocol*. We demonstrate the benefits of such a meta-level architecture for AOP by presenting several scenarios in which programs use the meta-interface of the language to tailor its semantics to a particular application execution context.

Case Study 3: Enabling AOP for DSLs. The reflective embedding of domain abstractions and programming paradigm abstractions can be combined to improve the modularization of crosscutting concerns in DSL programs. Like programs written in general-purpose languages, programs written in DSLs may also suffer from tangling and scattering in the presence of domain-specific crosscutting concerns. An open problem is that there is no adequate generic approach to enable dynamic AOP for DSLs.

In [3], we present a framework that supports aspect-oriented features for domain-specific base languages. We use reflective techniques to extend the interpretation of DSL implementations, such that *domain-specific join points* are intercepted and control is transferred to the framework that composes in domain-specific aspects. Using this framework to implement domain-specific aspect languages has several advantages. First, the framework facilitates the implementation of new aspect languages because large parts of aspect-oriented semantics can be reused. Second, both base programs and advice can be written in different DSLs. Third, the framework can compose aspects into DSL programs even as late as at runtime.

Case Study 4: Domain-specific Meta-Protocols. A well-known critique on meta-object protocols is that they are too powerful in that they allow to perform program and language adaptations that may result in incorrect programs.

To address this problem, we propose to use domain-specific abstractions on top of meta-protocols. Instead of using the full power of the protocol, the programmer uses a domain-specific language to specify adaptations on a more abstract level. These abstract adaptation specification is then actually performed through the implementation of that domain-specific language. This DSL internally calls the underlying meta-protocol that then performs the more technical operations. Because of that abstract specification and because the specification can be analyzed in the DSL implementation before actually executing the adaptation, the obtained domain-specific meta-protocol allows to develop more robust adaptations. We have instantiated a simple domain-specific language for specifying the detection and resolution of aspect interactions that is implemented on top of our meta-aspect protocol.

For a quantitative evaluation, the plan is to evaluate the development and the runtime costs of reflective embedding.

Development Costs. The plan is to measure the costs for implementing new languages and compare them to the costs of traditional stand-alone implementations. For evaluating the approach's quality for developing DSLs, we are developing an open source project that develops the same DSL (for implementing state machines) using different traditional implementation technologies, and we compare the qualities of the obtained language implementations with the one obtained using our approach. For evaluating the approach's quality for developing new aspect-oriented languages, we will measure the development costs of implementing existing domain-specific aspect languages, such as *COOL* and *RIDL*, and compare them to the costs of implementing the languages from scratch. Besides general metrics, such as lines of code, the reuse when growing a DSL implementation is measured, e.g., when incrementally adding new domain abstractions or when reusing aspect-oriented features from the AOP framework and comparing it to what it would cost for a stand-alone implementation.

Runtime Costs. The runtime costs of the DSLs in the case studies are measured using benchmarks and compared with related implementations. For DSLs without aspect-oriented features, as for DSLs in general no adequate benchmarks are available, new benchmarks will be designed to measure the runtime overhead imposed by the indirections necessary for enabling versatile language semantics by comparing execution time to stand-alone DSLs. For measuring the overhead imposed by the meta-aspect protocol, the execution times will be compared to the ones of existing aspect-oriented compilers and run-times without versatile AOP semantics. For DSLs with aspect-oriented features, we will compare to existing domain-specific aspect language implementations.

4. Future Work

Currently, we are comparing reflective embedding to other language implementation techniques, such as other embedding approaches, component-oriented language implementations, and monads. Improving performance by using a specialized meta-object protocol is future work.

References

[1] F. Chauvel and J.-M. Jézéquel. Code Generation from UML Models with Semantic Variations Points. In *UML MoDELs*, volume 3713 of *LNCS*, 2005.

[2] T. Dinkelaker, M. Mezini, and C. Bockisch. The Art of the Meta-Aspect Protocol. In *International Conference on Aspect-Oriented Software Development (AOSD.09)*, 2009.

[3] T. Dinkelaker, M. Monperrus, and M. Mezini. Untangling Crosscutting Concerns in Domain-specific Languages with Domain-specific Join Points. In *Workshop on Domain-specific Aspect Languages (co-located with AOSD)*, 2009.

[4] P. Hudak. Modular Domain Specific Languages and Tools. In P. Devanbu and J. Poulin, editors, *International Conference on Software Reuse*, pages 134–142. IEEE Press, 1998.

[5] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[6] G. Kiczales, J. d. Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[7] P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.