Prototyping Domain-Specific Language Semantics *

Daniel A. Sadilek
Humboldt-Universität zu Berlin, Computer Science Department, Germany
sadilek@informatik.hu-berlin.de

Abstract
Domain-specific languages (DSLs) need semantics. For an external, executable, metamodel-based DSL, this can be done in an operational or a translational way. In my dissertation, I develop a framework that allows both. It provides flexibility for semantics description in two axes: on the one axis, operational semantics is fixed and one can choose between different description languages (QVT, Java, Prolog, Abstract State Machines, and Scheme); on the other axis, Scheme is fixed and one can choose between operational and translational semantics. Using operational semantics, DSL program interpretation can be animated and debugged. Equivalence of operational semantics described with different languages can be tested by comparing execution traces.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors

General Terms Design, Languages

Keywords Language engineering, domain-specific languages, operational semantics, metamodelling

1. Introduction and Problem Statement
Domain-specific languages (DSLs) promise to increase developer productivity by raising the level of abstraction. DSLs use concepts of and a notation established in a specific domain. Thus, DSLs allow domain experts, who are non-programmers, to directly encode their domain knowledge about what a system under development should do.

But designing and defining a DSL is complex in itself; only when we manage to make this task as easy as possible, DSLs can fulfill their promise. The technologies involved in DSL definition and the process used depend on the type of the DSL. In my work, I consider external DSLs with execution semantics in the modelware technological space (i.e., DSLs that are independent of other (host) languages, that are used to create programs, and whose abstract syntax is defined with metamodels).

Usually, the execution semantics of such a DSL is given by defining a code generator. The typical process for creating the DSL puts emphasis on the generated code (Kelly and Tolvanen 2008): (1) Programmers write target code manually at least 3 times. (2) They factor out the commonalities to create a domain-specific framework. (3) They define a DSL that resembles the framework’s domain concepts. (4) They write a code generator targeting the domain framework. (5) Domain experts start creating and executing DSL programs.

This is appropriate for projects where a domain framework or code with reoccurring patterns is already available. But if such a code base is missing, domain experts cannot start working with the DSL before much target code has been programmed manually. What they may already know about the necessary domain concepts is not managed explicitly. Instead, the domain concepts flow implicitly into the code—transported by the domain knowledge, which the domain experts explain to the programmers. Later, the programmers have to recover the domain concepts from the code.

The goal of my dissertation is to develop a language prototyping framework that supports a reversed process for DSL development, which integrates domain experts more tightly and in which domain concepts are managed explicitly. It looks as follows: (1) Domain experts and programmers define the DSL concepts. (2) Programmers prototype the operational semantics of the DSL concepts. (3) The operational semantics description is used to derive two DSL interpreters: one can simulate DSL programs in the development environment, the other can execute them on a target platform. (4) Domain experts use the DSL to create DSL programs; they can execute them in the development environment and on the target platform. In this reversed development process, domain experts are integrated earlier than in the typical one because they take part in defining and testing the domain concepts.

The problems developing such a language prototyping framework are: How to describe the operational semantics? Multiple description languages are possible, which one to

* This work is supported by grants from the Deutsche Forschungsgemeinschaft, Graduiertenkolleg METRIK (GRK 1324/1).
choose? How can DSL interpretation based on operational semantics be visualised? How can DSL interpretation be brought to a target platform? How to represent DSL programs on the target platform? The reversed development process requires an operational approach. If performance is critical, how can we perform the transition to a translational approach?

2. A Language Prototyping Framework

The framework I develop is called EPROMISE\(^1\) and it comprises two sub-frameworks. The first one, EPROVIDE (Sadilek and Wachsmuth 2008), allows to describe the semantics of a metamodel-based DSL in an operational fashion. In EPROVIDE, we represent configurations of the operational semantics as models and define the space of all possible configurations with a metamodel. Consequently, a model-to-model transformation realises the transition relation that maps a configuration to its successor. Thus, executing a DSL program results in a configuration changing over time. With metamodel-based technologies, an editor for configurations can easily be defined. This allows for animated DSL program execution and debugging.

EPROVIDE is extensible with languages to describe the transition relation. Currently supported are QVT, Java, Prolog, Abstract State Machines, and Scheme. This allows a language engineer to choose a language that matches his skills and requirements. For each new description language, one has to decide how the configurations can be accessed physically and how they should be represented with the language’s data structures. For Scheme, these problems are solved by the second sub-framework, ESEMANTICS.

ESEMANTICS (Sadilek 2007) combines the Scheme programming language with the Eclipse Modelling Framework (EMF). It supports both operational and translational semantics: it allows one to use Scheme to describe the transition relation, and it supports the translation of DSL programs from a model representation to a Scheme representation.

DSL programs translated to a Scheme representation can be executed in two ways: in a simulation and on a target platform. For the simulation, I implemented a discrete-event simulation kernel managing a simulation time and supporting multiple, communicating instances of a program. For the target platform, I use a standard Scheme compiler.

By integrating ESEMANTICS with EPROVIDE, EPROMISE provides flexibility for semantics description in two axes: on the one axis, operational semantics is fixed and one can choose between different description languages; on the other axis, Scheme is fixed and one can choose between operational and translational semantics. The point where the axes cross is operational semantics described with Scheme.

3. Status and Future Work

I started my PhD work in August 2006 and plan to finish it in August 2009. At this point, I can think of multiple additional features that can be integrated into EPROMISE.

Compiling Operational Semantics. Currently, I am working on compiling Scheme-based operational semantics descriptions to a target platform like it is already possible with translational descriptions. My approach to compile a DSL program uses three artefacts: (1) the operational semantics of the DSL, which is programmed using a model interface consisting of generated, metamodel-specific Scheme procedures; (2) classes of a Scheme object system that provide the same model interface and that are generated from the DSL’s metamodel; (3) generated initialisation code that instantiates the Scheme classes to build an object structure resembling the DSL program. These three artefacts can be compiled with a standard Scheme compiler.

Testing Semantics Equivalence. A user may first want to prototype operational semantics with a specific language, say ASMs because they are formally founded and some properties of the semantics can be proofed. Later, he may want to execute programs with this semantics on a target platform. With EPROMISE, he could re-implement the operational semantics in Scheme, which would allow for execution on the target platform. As both the ASM-based and the Scheme-based semantics work on the same data structure (an EMF model), their equivalence could be tested by comparing execution traces of example DSL programs.

4. Evaluation

I use three example DSLs to evaluate my approach: a Petri net DSL, a DSL for specifying tests of continuous system, and a stream-oriented DSL. With the Petri net DSL, I can quickly test and illustrate my ideas. The test specification DSL is an extension of the existing language TTCN to support continuous data streams. The stream-oriented DSL can be used to describe earthquake detection algorithms. Currently, my working group develops an earthquake early warning system in which the detection algorithm is programmed in C++. Thus, I will be able to compare the performance of a DSL-based solution with the manual C++ implementation.

References


---

\(^1\) Eclipse-based framework for prototyping semantics to interpret, simulate, and compile DSL programs.