Prototyping and Simulating Domain-Specific Languages for Wireless Sensor Networks

Daniel A. Sadilek
sadilek@informatik.hu-berlin.de

Humboldt-Universität zu Berlin
Institute for Computer Science
Rudower Chaussee 26
10099 Berlin, Germany

Technical Report

2007-09-21

The development of software for wireless sensor networks is involved and complex. This does not only impose much work on programmers but also prevents domain experts from directly contributing parts of the software. Domain-specific languages may help with these problems—if they are inexpensive to define, have a syntax that domain experts understand, and creating simulations for them is easy. We propose a language engineering approach that meets these requirements and thus allows rapid prototyping of domain-specific languages. We present our implementation based on Scheme and Eclipse EMF and present first experiences from the prototyping of a stream-oriented language for the description of earthquake detection algorithms.

Contents

1 Introduction

Motivation. A wireless sensor network (WSN) consists of spatially distributed, wirelessly communicating devices that are small, often battery-powered, and equipped with a radio transceiver, sensors, and a microcontroller.

The software development for WSNs is a difficult task: it is mostly done in low-level programming languages like C or even Assembler, testing developed software on the network is very costly, and the resources of the WSN's nodes are usually very constrained. A possibility to ease the development is the usage of special programming languages for WSNs [10] [16] [24].

Currently, our research group is working on SAFER [22], a project of the European Union, in which we develop technologies for earthquake early warning systems based on WSNs. An integral part of such a system is an earthquake detection algorithm. It runs on every WSN node and constantly processes data coming from the node's acceleration sensors. If an earthquake is detected on one node, messages are sent to the surrounding nodes in order to reach a consensus on the earthquake event. If the nodes agree, a warning message is disseminated in the whole network.

Developing an earthquake detection algorithm requires knowledge from seismologists.

In [21], we suggest enabling them to directly create parts of the system by providing them with a programming language they can understand. The above mentioned special programming languages for WSNs are tailored to WSN-programming in general and target software developers. As such, they are not appropriate for seismologists. Seismologists need a language tailored for the application domain "earthquake detection algorithm" specifically. For example, a seismologist with background in signal processing may want to describe a detection algorithm with streams coming out from sensor sources, going through filters and then into sinks.

Goal. We want to integrate domain experts (seismologists) into the WSN software development process by providing them with DSLs they understand easily. Therefore, the DSLs' concepts and concrete syntax must match the domain experts' cognitive space and intuition. Furthermore, the domain experts should be able to test the programs they have written in the DSLs by executing them.

We act on several assumptions from which requirements for the language engineering approach arise. We assume that ...

- the DSL's concepts as well as their concrete syntax will not be clear in the first place.

  This means that the DSL has to be developed *iteratively.* Especially in the beginning of the DSL development, we expect a *prototyping cycle* in which domain expert and software engineer frequently take turns at using and changing the DSL. Frequent changes mean that the DSLs' concepts must be *easily definable and changeable.*

- the DSL's range of application and thus its reuse will be limited.

  This means that developing *tools* like editors or interpreters for the DSL must *cheap.* For example, the language engineering approach should allow automatic creation of an editor from a declarative description of a purpose-built concrete syntax.

- deploying and testing programs on the target platform is very costly.

  This means that it's not feasible to test programs written in the DSL directly on the target platform—especially in the early rounds of the prototyping cycle when the DSL is used more for the purpose of evaluating the DSL itself than of developing the system. Therefore, it is necessary that the DSL can be *simulated.*

Paper structure. The rest of this paper is structured as follows. In Section 2, we discuss the suitability of the two paradigms metamodeling and metaprogramming to reach our goal, we exhibit their combination as our approach, and present the research questions associated with this approach. In Section 3, we present our evaluation project and describe our implementation. We discuss related work in Section 4 and conclude with our results, future work, and a summary in Section 5.

2 Approach

2.1 Characteristics of metamodeling and metaprogramming

How can we provide the domain experts with DSLs? At the moment, there is much interest in DSLs in the context of *metamodeling technologies* like Eclipse's EMF, GMF, and openArchitectureWare [25] or Microsoft's Domain-Specific Language Tools [18]—mainly, because these products allow the automatic creation of tools supporting purpose-built concrete syntax.

Metamodels miss *inherent operational semantics*. They only contain a description of the language's abstract syntax, i.e. which language concepts are available and how they can be combined. They miss a description of the language's operational semantics, i.e. how the language concepts should "behave" when executed. Executing a model conforming to a DSL's metamodel is only possible with an additional artifact describing the operational semantics: either some form of translation[1] or an interpreter.

The *translation* translates models conforming to the metamodel to an executable language, for instance C. When iterating through the prototyping cycle, the language concepts and their semantics change incrementally. To reflect these changes, the translation must be adapted. Then, the translation must be executed to update the model's representation in the executable language. This is quite a number of steps for one prototyping cycle. Also it involves at least three languages: a language for the metamodel, a language for the transformation, and the executable language. Furthermore, the potentially big semantic gap between DSL and executable language must be bridged by the translation. This can either be done by one complex translation or by introducing additional intermediate translations. Both options have drawbacks: one complex translation is hard to manage and additional intermediate translations complicate the prototyping cycle by introducing more steps.

The *interpreter* simulates a machine that can directly execute models conforming to the metamodel. When the language concepts or their semantics change, the interpreter must be adapted. Compared to the translation, an interpreter reduces the number of steps necessary for one prototyping cycle but writing an interpreter from scratch is a non-trivial task in itself.

With an interpreter, there are two options for executing models on the target platform: the interpreter could be ported to the target platform or a translation to a language executable on the target platform could be provided in addition to the interpreter. Both options imply additional effort: porting the interpreter raises problems regarding memory-management, performance, etc.; providing an additional translation means describing the language's semantics twice.

We can conclude that metamodels make the language prototyping cycle—as we expect it for WSNs—expensive because they miss inherent operational semantics. Additional effort is necessary for providing a translation or an interpreter.

What can be done? The idea of DSLs has not come with metamodeling but exists

[1]code generation, compilation, or transformation

since decades in *metaprogrammable languages*[2]. Although DSLs in metaprogrammable languages like Lisp and Smalltalk seemed to be buried in oblivion, the metaprogramming paradigm keeps getting more and more attention again. Language designers are steadily introducing more and more metaprogramming concepts into their languages. For instance, C++ compile-time template metaprogramming, which is a popular technique for generative programming, Java's reflection mechanism, and Ruby's run-time metaprogramming.

DSLs defined inside metaprogrammable languages provide inherent operational semantics. Defining and using domain-specific concepts in metaprogrammable languages is nothing more than a way to structure a program—and programs are executable. In a metaprogrammable language, the definition of a domain-specific concept can be done in multiple abstraction steps. These steps correspond to additional intermediate translations of metamodels but without complicating the prototyping cycle. Moreover, a metaprogrammable language provides a set of base concepts that don't have to be defined from scratch when a new DSL should be developed. For instance, a language engineer has already at hand basic datatypes, arithmetic operators or control flow statements.

So far, metaprogrammable languages seem better suited for prototyping DSLs for WSNs than metamodeling technologies. However, they have an important disadvantage compared to the metamodeling technologies: they don't allow the definition of a purpose-built concrete syntax. Depending on the domain, this may inhibit matching the DSL's syntax with the domain expert's cognitive space and intuition. In particular, this is the case if the domain expert expects a graphical representation of the DSL.

2.2 Proposed Approach

In the last section, we saw that both the currently famous metamodeling technologies and metaprogrammable languages have drawbacks regarding our purposes: Metamodeling technologies miss inherent operational semantics and a metaprogrammable language's concrete syntax may not match the domain expert's expectations. However, we think that these approaches have complementary qualities and a combination of both may have the characteristics we need.

We propose to use a metaprogrammable language that is executable on the target platform for prototyping the DSL. In this prototyping process, simulating programs built with the DSL will be necessary. Therefore, there has to be a simulation environment that is able to execute the metaprogrammable language and to provide mock objects for the target platform's special features like sensor or communication devices. In order to integrate domain experts, a metamodel will be created that reflects the structure of the DSL described within the metaprogrammable language. Based on this metamodel, tools that support purpose-built concrete syntax for the DSL can be described and generated.

With this approach, we have a description of the DSL including its operational semantics. Programs written in the DSL can be directly executed without an interpreter or a complex translation. This feature fulfills the requirement that the DSL can be simulated

[2]Languages that can work with their own code as data structures.

and it is the main difference of our approach to other metamodeling approaches.

Additionally, our approach allows us to reuse metamodeling facilities. For example, we can easily generate an editor that supports a purpose-built concrete syntax for the DSL.

2.3 Research Questions

There are five main research questions associated with our approach:

1. *Combination:* How can a metaprogrammable language be combined with metamodeling technologies? Conceptually, programs in a metaprogrammable language and models conforming to a metamodel must be aligned in some way. Is a modification of the metaprogrammable language necessary, e.g. of its type system? Technically, how are models/programs converted between the two representation forms?

2. *Uniform execution:* Is it possible to execute programs written in the DSL both in the simulation and on the small, resource-constrained WSN nodes without modifications? If not, which modifications are necessary?

3. *Operational semantics redundancy:* The domain-specific concepts need to have a definition[3] of their operational semantics for the target platform. Is it possible to reuse the definition done for the simulation?

4. *Performance:* How much performance loss is associated with our approach—in simulation and on the target platform? Is our approach feasible when real-time execution is required?

5. *Development efficiency:* Can our approach compete with other approaches regarding developer productivity (including maintainability, debuggability), and openness for other technologies (e.g. calling functions from existing libraries or integrating with given middleware structures)?

3 Implementation

In this section, we present the implementation of our approach. We selected an evaluation project, developed a toolset we named *Sminco* and prototyped a stream-oriented DSL.[4]

3.1 Evaluation Project

For the evaluation of our approach we chose the development of an earthquake early warning system running on a WSN. The final hardware platform will consist of hundreds

[3]given as code in the metaprogrammable language

[4]An extended version of this paper with more implementation details will be published as a technical report.

to thousands of WLAN-nodes with about 8 MB main memory and 32 MB flash memory that are distributed over a city. Unfortunately, this hardware is not yet at our disposal. Therefore, we used a single Netgear WGT634U WLAN-router as target platform for our evaluation. It runs OpenWrt Linux with a 2.6.16.13 kernel and has neither sensors nor warning devices.

For the simulation of earthquake detection algorithms, we have a database containing recorded data from acceleration sensors of several real earthquakes.

In the context of the EU-project SAFER, other working group members are developing with a three-step approach the same algorithms and protocols as we do. First, they model and simulate the communication protocols in SDL; then they integrate these protocols with the detection algorithm in a C++-based simulation, which accesses the databases with the recorded sensor data; finally, they create code for the target platform[5]. This allows us to compare our approach with this more conventional one.

3.2 Stream-Oriented DSL

We developed a stream-oriented DSL for the description of earthquake detection algorithms. Its main concepts are source, filter, and sink. Figure 1 shows an example usage of a simplified version of the DSL with purpose-built concrete syntax inside a generated graphical editor (Section 3.3). The shown model describes an earthquake detection algorithm running on one sensor node. Communication with other nodes is left out for the sake of simplicity.

The intention behind the model is as follows: Sensor readings from a sensor node's vertical (Z) *acceleration sensor* are piped through a filter called *STA/LTA detection* that realizes the earthquake detection. This filter lets pass sensor readings that are considered to be the beginning of an earthquake and blocks all others. When an earthquake is detected, usually multiple successive sensor readings pass the filter. Ultimately, the sensor readings will stream into a stream sink that generates an earthquake *detection warning*, e.g. by activating a warning horn whenever a sensor reading streams in. In order to keep the warning frequency at a reasonable level, an additional *time filter* is interposed between the STA/LTA filter and the detection warning sink. It lets pass one sensor reading and then blocks successive sensor readings for five seconds.

Figure 2 shows the metamodel of the stream-oriented DSL. It is kept extremely simple for this demonstration—as you can see the metamodel does hardly allow to formulate any other models than the example model in Figure 1.

3.3 Selected Tools

We based our implementation on the Eclipse Platform [25]. Accordingly, we use technologies from the Eclipse project for the metamodeling part: EMF as metamodeling framework and GMF for generating graphical editors.

[5]It is not yet clear whether the code for the target platform can be generated from existing artifacts or whether it must be hand-coded.

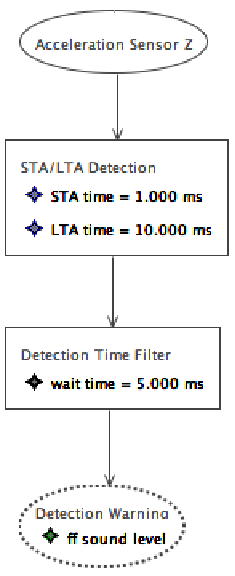


Figure 1: Screenshot of the earthquake detection algorithm model inside a GMF-generated graphical editor. Stream sources and sinks are represented as ellipses with a solid and a dashed line respectively. Stream filters are represented as rectangles.

We chose Scheme [13] as metaprogrammable language. It is small, clean and has a uniform concrete syntax allowing the representation of arbitrary domain-specific concepts. We use two implementations of Scheme: SISC runs on the JVM allowing easy integration with Eclipse; Chicken Scheme compiles Scheme programs to C source code that can be cross-compiled for the target platform. We selected SISC because we need an implementation that allows easy access to the Java world and provides support for *full continuations*[6], which we need for simulation. We selected Chicken Scheme because in a small performance evaluation we did, its results were satisfactory.

We measured the execution speeds of some Scheme/Lisp implementations that could be used on the target platform by calculating the Fibonacci number of 30. The results were obtained on the Netgear WGT634U WLAN-router and are given in Table 1. They show that the C implementation compiled with the switch `-O3` is the fastest one. The Chicken Scheme implementation is nearly as fast as the C implementation and the compiled code is approximately the same size if the Chicken runtime library is accessed as a shared object (SO) and debugging symbols are stripped from the executable. Gambit-C [8], although a high-performance Scheme implementation, performed badly—we did not investigate further the reasons for that. Hedgehog [27] is interesting because of its

[6]A continuation is a representation of the execution state of a program. Languages that support continuations provide means to capture a continuation in a variable and resume the execution at the state captured in the continuation. Full continuation support means that a continuation can not only be resumed once in a specific context but arbitrarily often and in any context.

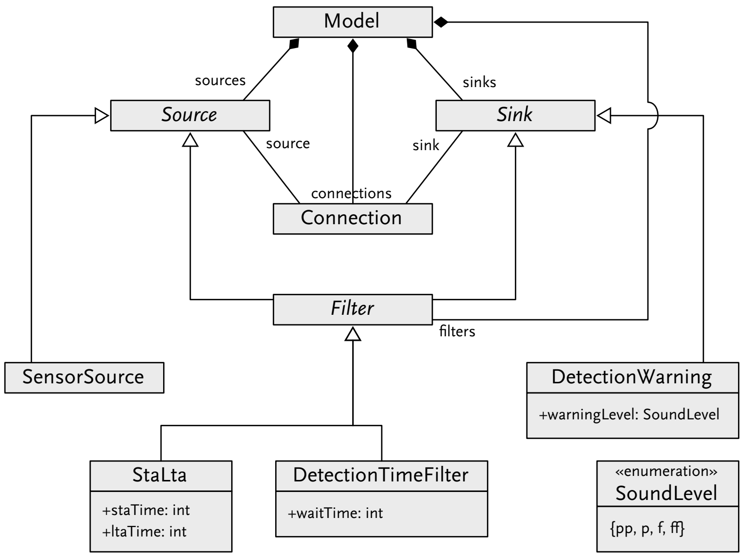


Figure 2: Metamodel of the simplified stream-oriented DSL.

Language and options	Code size [kB]	Time [s]
C -O3	8	0.41
Chicken	867	0.50
Chicken SO	13	0.50
Chicken SO, stripped	8	0.50
Hedgehog	0.172	8.13
Gambit-C	3800	12.15

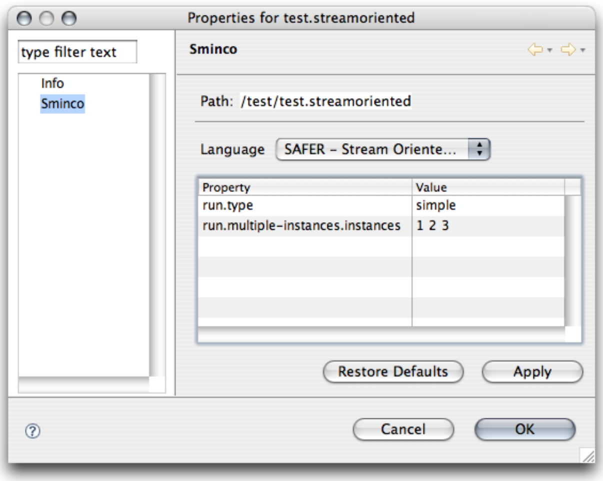
Table 1: Code size and execution time for calculating fib(30) on a Netgear WGT634U WLAN-Router.

small code size; it compiles a Lisp-like language to bytecode that is interpreted on the target platform.

It must be stressed that this performance evaluation is by no means comprehensive. We were seeking for a first pragmatic, working choice. Searching for the best possibility to execute DSLs on the target platform is subject to further research.

3.4 Plugin Architecture

We developed two Eclipse plugins: a main plugin and a DSL plugin for the stream-oriented language. The main plugin for Sminco provides the infrastructure for our approach, namely (i) an interpreter that is an instance of SISC extended for bridging EMF and Scheme and for the simulation of domain-specific concepts, (ii) a mechanism for caching intermediate results of simulations, and (iii) management facilities for controlling properties of the simulation and the target code generation.

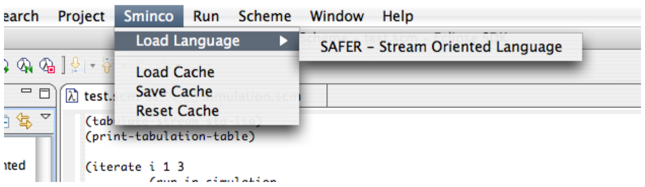


Path: /test/test.streamoriented

Language SAFER – Stream Oriente...

Property	Value
run.type	simple
run.multiple-instances.instances	1 2 3

Figure 3: Sminco properties dialog. Simulation and target code generation properties can be defined per file.



```
(print-tabulation-table)

(iterate i 1 3
```

Figure 4: Sminco menu. For every supported DSL, a menu item for loading its concepts is created.

For every DSL, an additional plugin must be developed providing the DSL's description. The description consists of (i) Scheme code for the DSL's concepts, (ii) support for the translation between EMF and Scheme, (iii) support for simulation, and (iv) a definition of the properties the user can change to control the simulation and the target code generation. The main plugin creates a menu item for loading the language's concepts for each DSL plugin (Fig. 4).

3.5 EMF/Scheme-Bridge

The most important part of Sminco is the bridge between EMF and Scheme. It answers the first research question by realizing the combination of the metaprogrammable language Scheme with the metamodeling technology EMF.

Programs represented as EMF-models can be translated to programs represented as a Scheme data structure. What is the nature of this data structure? In Scheme (and other Lisp dialects) both code and data are represented as S-expressions[7]. This metaprogramming property allows the data structure to be handled both as a mere data structure and to be executed. It can be executed either by evaluating it at runtime or by serializing it to a Scheme source file (see Section 3.6).

When developing a new language, the translation from EMF-models to S-expressions must be described for that language. There are two options where the knowledge about the syntactical differences between EMF-models and S-expressions can be encoded:

1. *Special syntactic forms.* Sminco provides a *standard translation* from EMF-models to Scheme S-expressions. If it is used, appropriate syntactic forms must be provided that allow the result of the standard translation to be interpreted as Scheme code.

2. *Custom translation.* Instead of using the standard translation, a custom translation can be provided that generates S-expressions that can be interpreted as Scheme code.

The standard translation translates all model objects uniformly, namely to a Scheme list with four elements for the object's class-name, unique id, attributes, and sub-objects. The list has the following structure:

```
Element         := '(' <classname> <id> '(' Attribute* ')'
                                        '(' Element*   ')' ')'
Attribute        := '(' <attribute-name> AttributeValue ')'
AttributeValue   := <literal-value> | ReferenceValue
ReferenceValue   := '(' 'ref' <ref-classname> <ref-id> ')'
```

Example 1 (Special syntactic forms)
Taking the model from Figure 1 as example, the result of the standard translation is the code shown in Figure 5. To make this code executable, special syntactic forms would have to be supplied for `Model`, `SensorSource`, etc.[8]

Example 2 (Custom translation)
If a custom translation is provided, the result of the transformation may be the code shown in Figure 6. As you can see, in this code, the dataflow is not given explicitly via additional `Connection` expressions but implicitly via the containment hierarchy of the expressions.

Which option is the better one—special syntactic forms or a custom translation—depends on which representation of a domain model in Scheme is most convenient. If a domain model can be conveniently expressed as a mere listing of its elements, providing special syntactic forms that match the standard translation's result is inexpensive. If

[7]S-expressions are, like XML, a way to represent semi-structured data. They consist of basic objects such as strings or numbers, the special atom `nil`, and pairs. Lists are composed of nested pairs with `nil` representing the end of the list.

[8]or just for `Model` with `SensorSource`, etc. being keywords

```
(Model 1 () ((SensorSource        2 (                            ) ())
             (StaLta              3 ((staTime 1000) (ltaTime 10000)) ())
             (DetectionTimeFilter 4 ((waittime 5000)             ) ())
             (DetectionWarning    5 ((soundLevel 'ff)            ) ())
             (Connection          6 ((source (ref SensorSource 2))
                                     (sink   (ref StaLta 3))) ())
             (Connection          7 ((source (ref StaLta 3))
                                     (sink   (ref DetectionTimeFilter 4))) ())
             (Connection          8 ((source (ref DetectionTimeFilter 4))
                                     (sink   (ref DetectionWarning 5))) ())))
```

Figure 5: Example model in its S-expression representation generated by the standard translation.

```
(detection-warning
  (detection-time-filter
    (sta-lta (sensor-source) 1000 10000)
    5000)
  'ff)
```

Figure 6: Example model in its S-expression representation generated by a custom translation.

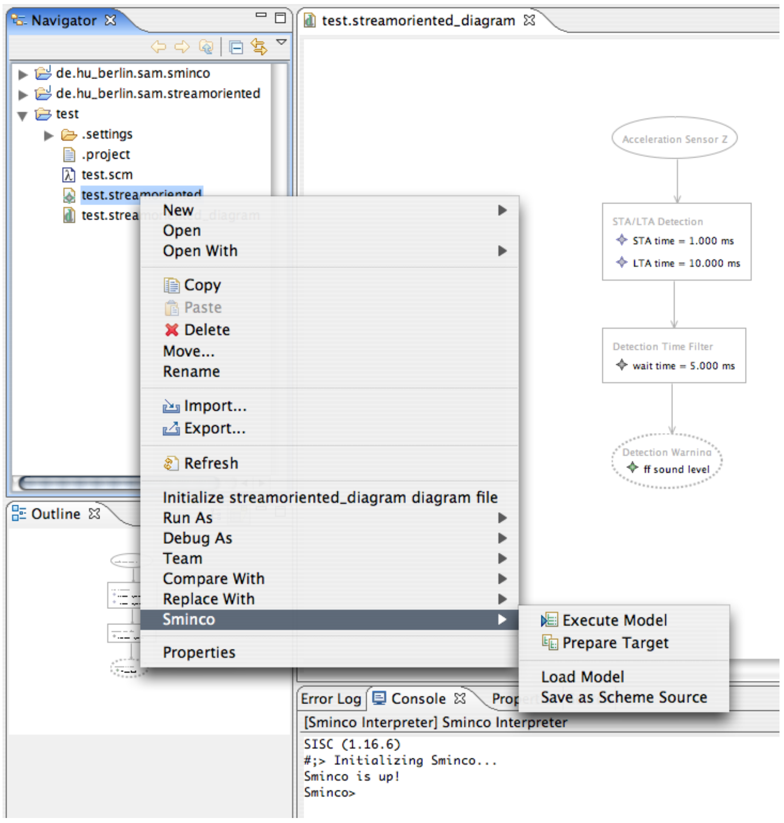
the domain model's representation needs to be more complex with nested expressions (like in Figure 6) or with multiple occurrences of one element, a custom translation is less expensive.

It must be stressed that the translation of an EMF-model to a Scheme S-expression is a mere syntactical one. The metamodel has been derived from the domain-specific concepts that were developed within Scheme. Therefore, the abstract syntax described by the EMF metamodel and the abstract syntax described by the domain-specific concepts in Scheme are identical. There is no semantic gap that has to be bridged by the translation, thus it can be kept simple.

In the research questions, we asked whether a modification of the metaprogrammable language would be necessary? It wasn't necessary for EMF and the Scheme implementation SISC. SISC provides a bridge to Java. From Java, EMF-models can be accessed either via a typed API of generated Java classes or via an untyped, reflective API. The combination of SISC's bridge to Java and the Java APIs to access EMF allowed us to represent EMF objects as Scheme variables. We use SISC's object-oriented type system with its bridge to EMF's typed, generated API to reference classes, to create new EMF objects, and for a type-dependent dispatch of Scheme code. We use EMF's untyped, reflective API to provide OCL-like expressions in Scheme for accessing EMF-models. These OCL-like expressions are a convenient way to describe custom translations.

3.6 Execution

As stated in the last section, a DSL program in its S-expression representation can be executed either by evaluating it or by serializing it to a Scheme source file, which can be



```
[Sminco Interpreter] Sminco Interpreter
SISC (1.16.6)
#;> Initializing Sminco...
Sminco is up!
Sminco>
```

Figure 7: Sminco context menu of a DSL program in its metamodel-based representation. In the background, the SISC console and a generated graphical editor for the stream-oriented language can be seen.

executed as usual.

To *simulate* a program, it is evaluated inside the development environment. For this, we created a simulation kernel for discrete event simulation. The kernel supports both event oriented and process oriented modeling and significantly depends on Schemes continuation support. It provides a special syntactic form `run-in-simulation` that can be called with the model's S-expression representation as body. Advancing the simulation time is accomplished by providing mock objects for special features of the target platform. These special features may be hardware features such as sensor or communication devices or OS-specific services like threading support. For example, if the target platform supports a `sleep` concept for threaded programming, this `sleep` is implemented to advance the simulation time accordingly. If the processing time of calculations is relevant for the simulation, calculation concepts like arithmetic operators can advance the simulation time, as well.

The simulation function is accessible by selecting "Execute model" from a model's context menu (Fig. 7).

Example 3
In the simulation, the stream source `sensor-source` *is implemented to read from the database with the recorded data of real earthquakes. The data is time-stamped and the simulation implementation of* `sensor-source` *advances the simulation time correspondingly. The stream sink* `detection-warning` *just writes to the console whenever a detection warning streams in.*

To *compile* a program for the target platform, it is serialized to a Scheme source file. Additionally, a scaffold for the compilation is generated containing a makefile and additional Scheme source files. These define the procedures that provide access to the platform's special features. The Scheme source files are compiled to C with the Chicken Scheme compiler and the C source files are cross-compiled for the target platform.

The compilation function is accessible by selecting "Prepare Target" from a model's context menu (Fig. 7).

Example 4
For testing on the target platform, the stream source `sensor-source` *is implemented to read from a TCP socket to which our desktop computer sends values. The stream sink* `detection-warning` *just writes to the console.*

Example 5
For a real deployment on the target platform, the stream source `sensor-source` *would be implemented to read from a real acceleration sensor and the stream sink* `detection-warning` *would trigger a warning device like a horn.*

We want to emphasize that no changes of the model in its S-expression representation are necessary to compile it for both simulation and the target platform. Providing appropriate mock objects for the platform's special features is sufficient.

3.7 Development Process

We used the development process that's depicted in Fig. 8. First, the special features that the platform offers have to be identified in order to map them to the simulation infrastructure. In our evaluation project, these special features were (amongst others) `sensor-source` and `detection-warning`. Then the domain-specific concepts are prototyped which involves the repeated creation or modification of domain-specific concepts, their use and simulation. In the stream-oriented DSL, the stream filter `sta-lta` was such a domain-specific concept. This way, the development of domain-specific concepts becomes a standard programming task in which in turn established development processes, e.g. Extreme Programming [2], and techniques[9], e.g. debugging and refactoring [9], can be applied.

After some of such main prototyping cycles that are done mainly by a software developer, the domain expert—in our evaluation project a geologist—can be involved by taking the prototyped domain-specific concepts, representing them in a metamodel (like

[9]depending on the choice of the metaprogrammable language

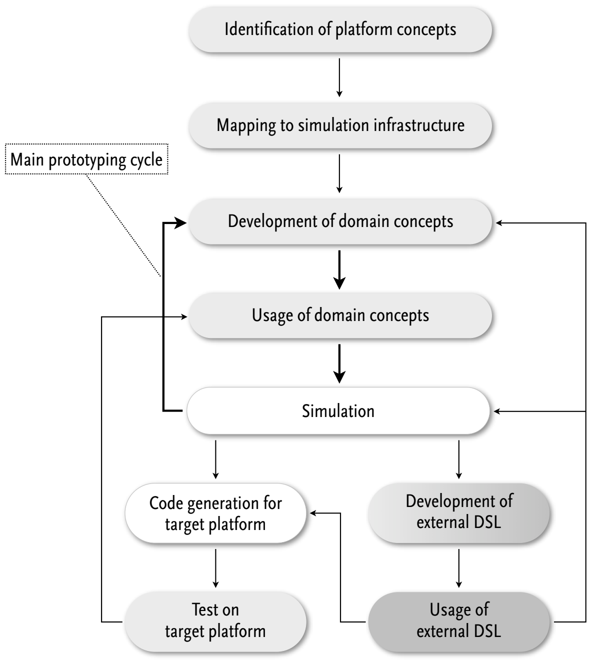

Identification of platform concepts
Mapping to simulation infrastructure
Main prototyping cycle
Development of domain concepts
Usage of domain concepts
Simulation
Code generation for target platform
Development of external DSL
Test on target platform
Usage of external DSL

Figure 8: Used development process. Light gray: work done by a software developer; dark gray: work done by domain experts; white: automated tasks. Less important steps back to former tasks are omitted.

in Fig. 2) and providing a purpose-built concrete syntax (like in Fig. 1). The domain expert can create and simulate a domain model and in case he finds the domain-specific concepts inadequate, he can prompt the software developer to change them accordingly.

Finally, code for the target platform can be generated, deployed, and tested.

4 Related Work

Ptolemy [5] supports the coupling of models with different operational semantics[10]. The semantics definition for the target platform is either done via full code generation or with Copernicus. Copernicus supports the partly reuse of the semantics definition done for the development environment by removing Ptolemy dependencies from Java class-files via byte-code rewriting. A subproject of Ptolemy named VisualSense [1] allows the modeling of WSNs. Viptos [7] combines VisualSense with the TinyOS-Simulator TOSSIM [17] in order to enable the transition from high-level modeling to target-code simulation and deployment. Although Ptolemy is advanced in many aspects, it is unsuitable for prototyping new domains because it does not manage the metamodel of domains explicitly. For a new domain it requires the user to manually supply a Java package with a

[10]In Ptolemy operational semantics is called "computational model"

set of classes.

GME [15] and MPS [11] work with an explicit representation of a model's metamodel but models in these systems are not inherently executable. In GME so-called "model interpreters" [12] must be supplied that can create code or other artifacts from traversing a model. Although they are called "interpreters" they actually serve for specifying transformations and are unsuitable for directly interpreting a model. The main feature of MPS is the possibility to easily create a concrete syntax with editor support for a new metamodel; but it has only support for form-based, textual syntax and it does not provide direct executability of models.

XMF-Mosaic [28], KerMeta [20], and the language modeling framework presented in [23] all provide direct executability by means of a core language. XMF-Mosaic has in its core an executable metamodeling language called XCore. With XCore successive layers of abstraction can be built similar to Scheme. KerMeta is an extension of Ecore, the meta-metamodel of Eclipse EMF. It adds functions and expressions and thus allows the description of operational semantics of a metamodel. The framework presented in [23] is similar to KerMeta as it allows the addition of functions to a metamodel. They differ in their storing of the runtime state. KerMeta maintains the runtime state implicitly in the metamodel-interpreter. The framework presented in [23] stores it explicitly in the model repository. In none of the three tools the core language can be executed outside the development environment; all tools require additional translations.

ASF+SDF (algebraic specification formalism + syntax definition formalism) [14] allows the declarative, algebraic definition of languages and their translation based on an abstract syntax tree. It also allows the definition of a concrete syntax. In [26] Visser proposes an approach based on SDF to include the concrete syntax of a metaprogrammed language into the metaprogramming language. The difference to our approach is that his goal is to ease the metaprogramming when metaprogrammed and metaprogramming language are not the same while we suppose them to be the same.

Of course, there are also languages intended specifically for modeling embedded control systems, e.g. Lustre, Esterel, and Signal [6, 4, 3]. But if their specific strengths like hard real-time support or good analyzability are not needed, DSLs are better suited for the integration of domain experts into the development process.

5 Conclusion

5.1 Results

We found the following answers to our research questions (Section 2.3).

1. *Combination:* We realized the combination of the metaprogrammable language Scheme and the metamodeling technology EMF as described in Section 3.5. We did not need to modify Scheme and found the translation between EMF-models and Scheme S-expressions to be a simple since mere syntactical one.

2. *Uniform execution:* For this question, we have a first indication of a positive answer. For the stream-oriented DSL, no changes of the model or its Scheme

representation as it is used for simulation are necessary to compile it for the target platform (Section 3.6).

3. *Operational semantics redundancy:* We could avoid redundancies in the description of the stream-oriented DSL's operational semantics for the simulation and for the target platform.
   We use the metaprogrammable language instead of translations to raise the abstraction from platform concepts to domain-specific concepts.[11] This raise of abstraction is exactly the definition of the DSL's operational semantics.[12] The metaprogrammable language Scheme is not only executable in the development environment but also on the target platform. This allows one definition of the DSL's operational semantics to be used both in simulation and on the target platform. Of course, development environment and target platform are different. Therefore, platform concepts like `sensor-source` must be bound differently. In the development environment, the platform concepts are bound to the simulation infrastructure, e.g. there `sensor-source` is implemented to read test data from a database and to advance the simulation time. On the target platform, the platform concepts are bound to the platform's infrastructure, e.g. there `sensor-source` is implemented to acquire values from the acceleration sensor with a specific frequency.

4. *Performance:* Using the metaprogrammable language provides strong abstraction capabilities implying performance penalties. We saw that our approach has worse performance than low-level coding or hand-coded code-generation in two aspects: (i) execution speed on the target platform and (ii) execution speed in the simulation.

5. *Development efficiency:* We don't have enough experience yet to benchmark the development efficiency systematically. Nevertheless, we are optimistic because of the experiences we had when prototyping the stream-oriented DSL. However, we already found an obstacle for our approach. Unfortunately, the tool support for Scheme is bad. Editing, code navigation, debugging, and profiling have to be done without the tool support one is accustomed to from Java, for instance.

Independent of our research questions, we found some interesting synergies arising from the combination of metaprogramming and metamodeling:

Bridging the gap in steps. In Section 2.1, we mentioned the abstraction gap between a DSL and its target language. We argued that the gap is difficult to bridge in one translational step and that a metaprogrammable language may allow us to use multiple steps. This was indeed the case for the stream-oriented DSL. We bridged the gap with four successive layers of abstraction, each building on its subjacent layer: (i) Scheme's primitives like `delay` and `force` (for delaying a computation), (ii) stream primitives

[11]Strictly speaking, we raise the abstraction from platform concepts *and* the metaprogrammable language's base set of available concepts to domain-specific concepts.

[12]in terms of the platform's and the metaprogrammable language's concepts

Approach	Subtask order and separation
Direct code generation	semantic translation + syntactic translation
MDA approach	semantic translation, syntactic translation
Our approach	syntactic translation, semantic translation

Table 2: Separation and order of translation subtasks.

like `stream-cons` (for constructing a stream) defined by Scheme's `delay`/`force` pair, (iii) *basic* stream operators like `stream-map` (for mapping all stream elements through an expression) defined by the stream primitives, and (iv) *higher* stream operators like `moving-average` (for calculating an arithmetic mean over a moving time window) defined by basic stream operators.

Clean separation of translation subtasks. The translation of a domain model (formulated in a DSL) to code for the target platform is usually done with one of *two approaches*: one approach is to generate code for the target platform from the domain model in one step[13]; the other approach is to transform the domain model to a platform model and then, in a second step, to generate code from the platform model. The latter approach is proclaimed by MDA [19] with the transformation from a platform-independent to a platform-specific model (PIM to PSM) and following code generation from the PSM.
The translation of a domain model to code for the target platform consists of *two subtasks*: one is to describe the semantics of the domain-specific concepts (the semantic translation); the other is to change the representation of a model, e.g. from XMI to Java (the syntactic translation).
The two approaches to translate a domain model to code for the target platform—direct code generation and the MDA approach—can be characterized by order and separation of the two subtasks—semantic translation and syntactic translation (see Table 2). Direct code generation intermingles semantic and syntactic translation in one step and therefore is hard to write and maintain. The MDA approach separates these two subtasks in two steps: first, the semantic translation is done from PIM to PSM without changing the syntactic domain; then, the syntactic translation is done from PSM to target code without changing the semantic domain.
Interestingly, we found our approach providing the same separation of subtasks as the MDA approach but in reverse order: first, we do a syntactic translation of a program from its metamodel-based representation to its Scheme representation; then, we do a semantic translation of the domain-specific concepts to base concepts of Scheme and the target platform by means of Scheme's abstraction facilities.

[13]e.g. by using a template language

5.2 Future Work

Execution on the target platform. In our evaluation project SAFER, we have deployed domain programs to the target platform only for small examples, yet. We will compile more complex programs that use special features of the target platform, for instance acceleration sensors and a warning horn.
We will try to support very small sensor nodes (with about 10 kB RAM and 32 kB flash memory) in application scenarios with heterogeneous hardware and frequent software updates. Under these conditions, compiling the domain programs to bytecode and executing them in virtual machines may be more appropriate than compiling them to native code. Our very next step will be to examine this virtual machine approach. Especially, we are interested in whether we can keep uniform execution and redundancy free operational semantics description.

Comparison. We want to broaden our understanding about what the principle commonalities and differences between our approach and Model Driven Engineering (including Model Driven Architecture), Language Driven Development and Generative Programming are. We want to identify criteria that determine for a given development problem which approach is best suited.

Other DSLs. We plan to apply our approach to other DSLs, e.g. a workflow description language and a fire spread modeling language. This will help us getting more evidence for the uniform execution and the redundancy free operational semantics description. Also, other DSLs will help us to evaluate the development efficiency of our approach.

Performance measurements. With the calculation of Fibonacci numbers we saw that, in general, our approach results in lower execution speed on the target platform than low-level coding but systematic performance measurements for more complex examples are still missing.

Language engineering. We evaluated our approach for one iteration of the prototyping cycle. In the future, we will investigate if our approach can be extended to a well-defined DSL engineering process that supports language evolution with instance co-evolution and systematic automated testing.

5.3 Summary

We identified the need to provide domain experts with DSLs in order to integrate them into the development process of software for WSNs (Section 1). We argued that the currently famous metamodeling technologies are not sufficient for this and proposed the usage of metaprogramming in combination with metamodeling technologies (Section 2).

We selected the combination of the Scheme programming language and Eclipse EMF and did a prototypical implementation of a toolset, Sminco, realizing their combination (Section 3). We have shown that a small example DSL for the stream-oriented description of earthquake detection algorithms can be developed with our toolset: this includes definition of DSL concepts, simulation, creation of purpose-built concrete syntax, and execution on the target platform for small examples.

We gave answers to our research questions (Section 5.1). As main results, we showed how Scheme can be combined with Eclipse EMF (1^{st} question), that models in our stream-oriented DSL created for the simulation can be executed on the target platform without modifications (2^{nd} question), and that redundancy in the description of a DSL's operational semantics for simulation and for the target platform can be avoided (3^{rd} question).

We think that our approach has advantages if domain experts should be integrated into the software development process by providing them with a DSL, the DSL's concepts are not yet clear and need to be prototyped, and simulation is essential because frequent deployment and testing on the target platform is too costly.

Lots of interesting questions are still open (Section 5.2) and our very next step will be to examine a virtual machine approach for the execution of developed DSLs on the target platform.

Acknowledgements

Thanks to Joachim Fischer, Markus Scheidgen, Sebastian Schuster, and the anonymous reviewers for comments on preliminary versions of this paper. This work was supported by a grant from the DFG (German Research Foundation, research training group METRIK).

References

[1] P. Baldwin, S. Kohli, E. A. Lee, X. Liu, and Y. Zhao. Modeling of sensor nets in Ptolemy II. In *IPSN '04: Proceedings of the third international symposium on Information processing in sensor networks*, pages 359–368, New York, NY, USA, 2004. ACM Press.

[2] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 2000.

[3] A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

[4] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. *Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java.* University of California at Berkeley, January 2007.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM*

SIGACT-SIGPLAN symposium on Principles of programming languages, pages 178–188, New York, NY, USA, 1987. ACM Press.

[7] E. Cheong, E. A. Lee, and Y. Zhao. Viptos: a graphical development and simulation environment for tinyOS-based wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 302–302, New York, NY, USA, 2005. ACM Press.

[8] M. Feeley. *Gambit-C, version 4.0 beta 18: A portable implementation of Scheme.* `http://www.iro.umontreal.ca/~gambit/doc/gambit-c.html`, June 2006.

[9] M. Fowler, K. Beck, J. Brant, and W. Opdyke. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[10] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[11] JetBrains. *Meta Programming System.* `http://www.jetbrains.com/mps/`, May 2007.

[12] G. Karsai. Structured Specification of Model Interpreters. In *ECBS: IEEE Conference and Workshop on Engineering of Computer-Based Systems*, volume 0, pages 84–90, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[13] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *hosc*, 11(1):7–105, 1998.

[14] P. Klint. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol.*, 2(2):176–201, 1993.

[15] Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.

[16] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(5):85–95, 2002.

[17] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire tinyOS applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM Press.

[18] Microsoft. *Microsoft Domain-Specific Language Tools.* `http://msdn2.microsoft.com/en-us/vstudio/aa718368.aspx`, May 2007.

[19] J. Miller and J. Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[20] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *MoDELS '05: Model driven engineering languages and systems (8th international conference)*, pages 264–278, Berlin, Germany, October 2005. Springer Verlag.

[21] D. Sadilek, F. Theisselmann, and G. Wachsmuth. Challenges for model-driven development of self-organising disaster management information systems. In J. Happe, H. Koziolek, M. Rohr, C. Storm, and T. Warns, editors, *IRTGW'06: Proceedings of the International Research Training Groups Workshop, Dagstuhl, Germany*, volume 3 of *Trustworthy Software Systems*, pages 24–26, Berlin, Germany, November 2006. GITO-Verlag.

[22] SAFER Project. *SAFER – Seismic eArly warning For EuRope.* `http://www.saferproject.net/`, May 2007.

[23] M. Scheidgen and J. Fischer. Human Comprehensible and Machine Executable Specifications of Operational Semantics. In *Third European Conference on Model Driven Architecture Foundations and Applications*, Lecture Notes in Computer Science, Haifa, Israel, June 2007. Springer-Verlag GmbH.

[24] K. Terfloth, G. Wittenburg, and J. Schiller. Facts — a rule-based middleware architecture for wireless sensor networks. In *COMSWARE 2006: First IEEE International Conference on Communication System Software and Middleware*, New Delhi, India, January 2006.

[25] The Eclipse Foundation. *Eclipse Platform.* `http://www.eclipse.org/`, May 2007.

[26] E. Visser. Meta-programming with Concrete Object Syntax. In *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, pages 299–315, London, UK, 2002. Springer-Verlag.

[27] L. Wirzenius and K. Oksanen. *Hedgehog Lisp 1.3.1.* `http://www.oliotalo.fi/hedgehog/hoglisp.html`, May 2007.

[28] Xactium Limited. *XMF-Mosaic.* `http://www.xactium.com/`, May 2007.