Towards Agile Language Engineering

Daniel A. Sadilek Markus Scheidgen
Guido Wachsmuth
Stephan Weißleder

Humboldt-Universität zu Berlin
Unter den Linden 6
D-10099 Berlin, Germany
{sadilek|scheidge|guwac|weissled}@informatik.hu-berlin.de

December 15, 2008

Abstract

Language engineering is software engineering concerned with computer languages. Agile language engineering is the result of adapting agile principles to language engineering. An agile language engineering process is geared to ever changing requirements and fosters higher language quality, software quality, and developer productivity. It provides short iteration cycles, intensive user integration, control over frequent changes, and continuous delivery of valuable language tools. In this paper, we point out technological premises for agile language engineering; and we sketch how they are met by well-known as well as upcoming techniques and tools from language engineering.

1 Introduction

Languages change [16]. A language description, as any other piece of software, is designed, developed, tested, and maintained. But above all, the purpose and scope of languages change and languages have to be adapted. This applies especially to domain-specific languages that are designed for a

specific purpose without exact prior knowledge about this purpose. Language tools should not be built from scratch every time a language changes. Also, models and programs written in a language are valuable assets that must be preserved and adapted when the language changes. Language engineering [24, 9] addresses the quality of languages and the productivity of language development. In this paper, we address the problem of enabling language engineering to facilitate frequent changes.

Software engineering methodologies and techniques can be classified in a continuum spanning from *predictive* to *adaptive*. A predictive process is thoroughly planned and adapts hardly to changing requirements. In contrast, adaptive processes can easily adapt to changing requirements—the future is not planned in detail. Still, these processes can be well-defined. In 2001, the Agile Manifesto [7] defined principles for *agile software development*. Agile software development lies on the adaptive side of the continuum and is based on evolutionarily changing the software product in short iterations. The user is steadily involved and there is a usable, valuable software product at the end of each iteration.

In this paper, we apply agile principles to language engineering. *Agile language engineering* provides short iteration cycles, intensive user integration, control over frequent changes, and continuous delivery of valuable language tools.

In the following section, we introduce some necessary terminology. In Section 3, we present the agile principles for language engineering. In Section 4, we describe how an *agile language engineering* process may look like in general, and we demonstrate by an exemplary language how it can be realised with metamodelling technologies in particular. We discuss related work in Section 5 and conclude with our contribution and future work in Section 6.

2 Preliminaries

In this section, we give a short introduction into the foundations of *agile language engineering*. We discuss the different aspects of a language, several formalisms for describing a language, and domain-specific languages.

2.1 Language Aspects

Generally, a language comprises two aspects: the appearance and the meaning of language instances. Descriptions of computer languages reflect these aspects: The *concrete syntax* of a language describes the appearance and structure of language constructs. The *abstract syntax* specifies only the structure of language constructs. The *semantics* of a language describes the meaning of language constructs with respect to a semantic domain. These descriptions are closely related: Semantics are expressed in terms of the concrete or abstract syntax. Since both concern the structure of the language, concrete and abstract syntax interdepend. Language tools are derived from language descriptions—partly automatically generated, partly by hand. Keeping the different language descriptions, language instances, and language tools in sync is one of the serious challenges to language engineering.

2.2 Language Formalisms

Existing language engineering technology relies on different language models to describe language aspects. The underlying formalism (grammars, graph grammars, or metamodels), can be used to classify existing language engineering environments into *grammar ware*, *graph grammar ware*, and *model ware* [24]. In this paper, we touch on all formalisms but concentrate on model ware.

Grammars and related formalisms are the traditional way to specify the concrete and the abstract syntax of textual computer languages. Various formalisms exist to express the semantics of such languages in a denotational, axiomatic, or operational way. This can be done in terms of either the abstract or the concrete syntax of the language.

Graph grammars extend the idea of grammars to graphs. The (graphical) syntax of a language can be described by the production rules of a graph grammar [39]. The semantics of graphical languages can be defined operational or as a transformation to another language with triple graph grammars.

A *metamodel* is an object-oriented specification of the abstract syntax of a language. They are used in Model Driven Engineering (MDE) where models are the primary engineering artefacts. For textual languages, a grammar and a mapping between syntax trees and language instances can describe the concrete syntax of the language [4, 59] as done for the OCL standard [36] or for

SDL [17]. For graphical languages, a mapping between language constructs and graphical elements defines the concrete syntax. Language semantics can be expressed in denotational or operational semantics by model transformations.

2.3 Domain-Specific Languages

A domain-specific language (DSL) is a special type of language used to increase the productivity of software developers. In contrast to general purpose languages, a DSL provides domain-specific terms that cannot be applied generally but only in a limited domain. DSLs allow for more precise and readable expressions than general purpose languages. This enables domain experts with less programming experience to participate in software engineering.

The domain-specific nature of a DSL makes it subject to frequent change. Typically, concepts, concrete syntax, and semantics of a DSL are not clear in the first place. Prototypical language tools should be delivered to the domain experts as early as possible. Usage of these prototypes as well as communication between domain experts and language engineers reveal necessary changes. Changes should then result in new prototypes. Especially in the beginning, several changes can be expected. Therefore, an iterative development process is desirable. Language concepts must be easy to define and easy to change.

Usually, the application range and reuse of a DSL is limited. Nevertheless, a wide variety of language tools is needed: Editors are used to create language instances; compilers or interpreters are needed to execute or simulate language instances; debuggers and analysis tools are other useful tools for software developers. Therefore, the development of language tools for a DSL must be efficient. This suggests generic or generative solutions.

Agile language engineering as discussed in this paper is particular useful for the development of DSLs. It provides short iteration cycles, intensive user integration, control over frequent changes, and continuous delivery of language tools.

3 Agile Principles for Language Engineering

The Manifesto for Agile Software Development [7] embraces the fundamental principles underlying agile software development methods. In this section, we

examine the questions that arise from applying these principles to language engineering. We discuss these questions in the following section.

Agile software engineering emphasises individuals and interaction, particularly *user collaboration*. Face-to-face conversation is considered the most efficient and effective way to convey information. This comprises both, engineer-engineer and user-engineer conversation. What roles are involved in *agile language engineering*? What are the typical scenarios? How are roles allocated in these scenarios? What are the consequences for the engineering process? We study these questions in Section 4.1.

Agile software engineering requires an *engineering process*. So does *agile language engineering*. How could an *agile language engineering* process look like? Which steps does it consist of? How are the roles assigned in the process? We propose a process answering these questions in Section 4.2.

In agile software engineering, *working software* is the primary measure of progress. It should be delivered frequently and as early as possible. *Working software* means that the user can execute and use the delivered software. Languages are not executable and usable themselves. What does *working language* mean? How can we make a language work? In Section 4.3, we investigate these questions in detail.

Another principle in agile software engineering is the continuous delivery of *valuable software*. Testing ensures the quality of software and is an integral part of an agile process. What does testing mean for languages? When is a language test correct? What are appropriate test cases? How can a test suite be evaluated? We discuss these questions in Section 4.4.

Agile software engineering welcomes changing requirements, even late in development. Furthermore, *frequent change* is encouraged in order to ensure technical excellence, good design, and simplicity. Automated refactoring guarantees certain preservation properties and helps to control changes in agile processes. How can we control language change? What are the implications of the interdependencies of language aspects and tools? What impact does language change have on language instances? We consider these questions and stepwise language adaptation as a way to control language change in Section 4.5.

4 Agile Language Engineering Applied

In this section, we describe how *agile language engineering* can be realised using tools from the field of model ware. Thus, we answer the questions formulated in the last section. In each of the following subsections, we first discuss one of *agile language engineering*'s aspects in a general fashion and then exemplify it. For this, we use a running example from the earthquake detection domain.

Example 1 (Domain earthquake detection) *At the present time, our research group is working on SAFER [46], a multidisciplinary project of the European Union involving institutions from computer science, multiple geologic disciplines, meteorology, and disaster management. In this project, we develop technologies for earthquake early warning systems. An integral part of such a system is an earthquake detection algorithm, which constantly processes data coming from an acceleration sensor. Developing an earthquake detection algorithm requires knowledge from seismologists. In [43], we suggest enabling seismologists or other* domain experts *to directly contribute to a system model by providing them with a language they understand easily. This calls for a DSL with concepts that the seismologists know and with a concrete syntax that matches their intuition.*

4.1 Language Engineering Roles

Agile language engineering is concerned with three roles: *Domain experts* (DE) possess the knowledge necessary to develop software in a specific domain. For example, they can formulate requirements for the software. Often, domain experts are also users of the software. The software is implemented by *software engineers* (SE). In the context of *agile language engineering*, a software engineer is also a *language user* (LU = SE), namely the user of a DSL, which should ease software development The DSL itself is developed by a *language engineer* (LE).

Of course, each of these roles cannot only be played by one person but also by a group of people. Moreover, one person or group can also play multiple roles at once; we distinguish two development scenarios:

1. *DE, [SE = LE]:*
   Software engineers develop the language they use. Typically, the used language is a so-called *internal DSL* that is defined inside a flexible, metaprogrammable host language. Here, language engineering is an integral part of software engineering. The software engineer instruments abstraction mechanisms of the host language to create software more efficient, maximising reuse. This is for example the case in *language-oriented programming* [15].

   In this scenario, domain concepts are first realised as software artefacts—expressed with the DSL as it looks so far. If a domain concept has proved its value, it is incorporated into the DSL.

2. *[DE = SE], LE:*
   Domain experts develop (parts of) the software themselves; they are integrated into the software engineering process. For this, they need a language. Typically, the used language is a special *external DSL* with limited application scope. An external DSL, in contrast to an internal DSL, is an independent language that can have a concrete syntax built specially to match the domain experts' intuition.

   In this scenario, new domain concepts are identified by the domain experts. Given that the DSL—due to its limited application scope—does not allow to express the new concepts with existing ones, they are directly incorporated into the DSL by the language engineer.

Both scenarios can benefit from *agile language engineering* because languages are continuously extended during software development [54] in both scenarios.

Example 2 (Role assignment) *In the earthquake detection example, the domain expert is a seismologist. As already stated, we want to enable him to directly express earthquake detection algorithms with a DSL. This corresponds to the second scenario (DE = SE, LE): By using a DSL, the seismologist also acts as a software engineer; but he depends on a language engineer to define and extend the DSL he uses.*

4.2 An Agile Language Engineering Process

Realising the vision of *agile language engineering* requires an engineering process. Extreme programming [6] is an established process for agile soft-

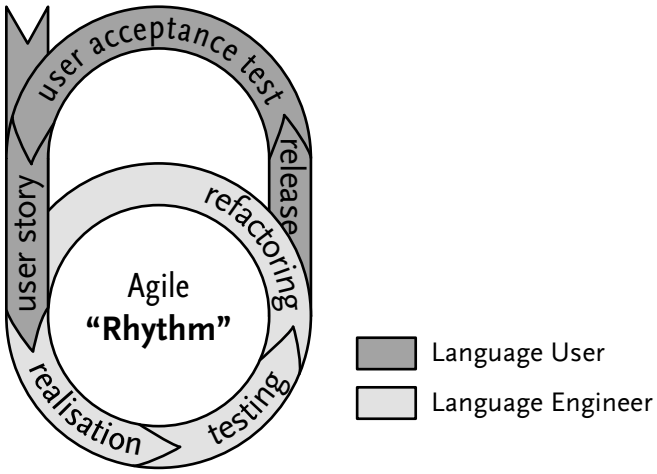

Figure 1: Agile language engineering process.

ware engineering. Its most distinctive feature compared to more predictive engineering processes, like the waterfall model, is the fast succession of small iterations. In each iteration, the software product is extended with a new function. This function is tested with automated tests and then presented to software users in acceptance tests. They can express their change requests in the form of user stories, which are considered in the next iteration. Inspecting architecture and design of the software is a regular task. This leads to changes of already implemented parts of the software. Therefore, refactoring is an integral part of agile software engineering.

We propose to apply a very similar process for *agile language engineering* (Figure 1). The differences lie in the tasks performed in the process steps and in the artefacts created. User stories given by language users act as input into the process. User stories are about what language users want to model with the language, how models should look like, what they should mean, and how tools should behave. From these stories, language engineers derive a realisation of the language. Such a realisation contains declarative models that language tools can be generated from automatically. Additionally, such a realisation can contain manually implemented tools or manual modifications of generated tools (cf. Section 4.3). Like with agile *software* engineering, the realisation is tested with automated tests and then presented to language users in acceptance tests. However, it is more complex to test

a language than testing ordinary software: all language aspects (abstract syntax, concrete syntax, semantics) and all tools for the language must be tested (cf. Section 4.4). If errors are found or change requests arise, the language must be adapted. This, again, is more complex for languages than for ordinary software. The specifications of the different language aspects are interdependent and already existing language instances and tools must be co-adapted when the language changes (cf. Section 4.5).

Example 3 (Creating the first model sketch) *The stream-oriented language is developed iteratively. A seismologist, who plays the role of a language user, and a computer scientist, who plays the role of a language engineer, start the first iteration with discussing some example uses of the new language. For the beginning, they concentrate on one specific detection algorithm called* STA/LTA *[48]. The seismologist sketches his ideas in an informal ad-hoc concrete syntax on a whiteboard (Figure 2). It is part of a larger user story about how this model is used, e.g. how it is executed.*

The intention behind the model is as follows: Streams come out of sensor sources, go through filters and then into sinks. Sensor readings from an acceleration sensor *are piped through a filter (STA/LTA) that realises the earthquake detection. The filter forwards sensor readings that are considered to be the beginning of an earthquake and blocks all others. The frequency of sensor readings is limited by another filter,* detection time filter*, before they stream into the stream sink* detection warning*. This stream sink generates an earthquake warning whenever a sensor reading streams in, for example by activating a warning horn.*

The seismologist wants to control timing properties of the STA/LTA detection and of the detection time filter. Furthermore, he wants to control the sound level of the detection warning. For this, seismologist and language engineer include corresponding parameters in the model.

In the next step (Section 4.3), the language engineer will derive the first version of a stream-oriented language *from the model sketch. In a later step (Section 4.4), the model sketch itself will be represented formally and will be used to test the metamodel of the language.*

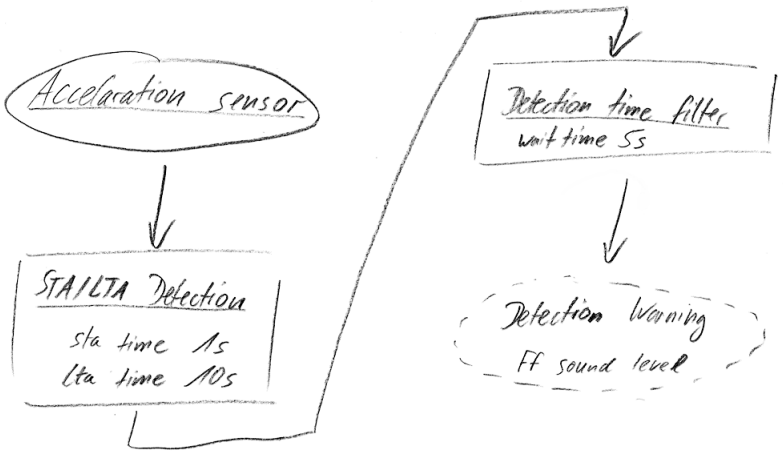

Figure 2: First sketch of an earthquake detection algorithm.

4.3 Make a Language Work

Agile engineering is about working software. We define a language as *working*, if tools exists that enable users to create, analyse, transform, and run instances of that language. The concrete set of tools depends on the nature of the language. If we have to produce a working language right from the beginning, we need technology that allows us to create language tools fast and based on potentially incomplete information. Consequently, we need toolkits and frameworks that allow us to describe language tools on a high level of abstraction, without the need to fill in all the details right from the beginning.

Existing language engineering frameworks consist of meta-languages and meta-tools. Meta-languages allow to describe distinct language aspects. Meta-tools can be used to automatically derive language tools from these descriptions in a generic or generative way. The frameworks thereby allow an efficient generative engineering of language tools. To create a full tool-chain of language tools, we have to use several frameworks in combination.

Examples for language frameworks for grammar-based language engineering are the ASF+SDF Meta-Environment [52], LDL [40], and The Eclipse IMP [22]. Model-based frameworks are usually based on a metamodelling language similar to OMG's MOF recommendations. Important work in this area are DSL-development frameworks like GME [3] or XMF [14] (originated in the MMF approach [13]), and metaprogramming facilities like MPS [15],

Kermeta [50], MetaEdit+ [34], AToM3 [35], AMMA [42], or openArchitectureWare [1]. Other frameworks are based on graph grammars and graph transformations: Graph REwrite And Transformations (GReaT) [12], DiaGen [53], or Tiger [49]. These frameworks allow to generate editors, analysers, interpreters or simulators, and compilers or code-generators.

Example 4 (Metamodel for the stream-oriented language) *The language engineer in our example needs to create an editor and a simulator for the stream-oriented language. We choose a metamodel based approach for this example and use existing metamodel based frameworks to build a graphical editor and simulator. First, the language engineer derives a first metamodel version based on the first model sketch in Figure 2. This metamodel contains basic stream-oriented concepts that are necessary for the development of an earthquake detection algorithm. The language engineer works agile and iteratively and, thus, constructs a minimal metamodel with only indispensable elements like that in Figure 3. At this stage, the metamodel is kept so simple that it hardly allows to formulate any other model than that in Figure 2.*

Example 5 (Editor for the stream-oriented language) *In order to create an editor that supports a graphical concrete syntax, the language engineer uses the Eclipse Graphical Modelling Framework GMF [20]. He models the graphical concrete syntax with meta-languages of GMF. He defines the symbols (boxes and circles) and their connections (arrows) in a "graphical model" and maps them to classes and associations of the metamodel in a "mapping model". From these models of the concrete syntax, GMF generates an editor that allows to create and manipulate models. Figure 5 shows a model created with this editor.*

Example 6 (Simulator for the stream-oriented language) *To build a simulator, the language engineer uses the techniques described in [47]. In that paper, the authors present a framework that allows to augment a metamodel with a semantics description. Based on this description, the framework can generate a simulator for models that are instances of the metamodel. The language engineer simply has to describe the language's operational semantics with operation signatures and implementations for these operations. The operation signatures are placed in the existing metamodel classes. The operation implementations are defined with an action language, similar to UML activities. Based on operation signatures and implementations, the framework*

can generate a model simulator that executes the operation implementations on a given model and thereby interprets the model based on the described semantics.

The metamodel in Figure 3 shows such operation signatures and further utility classes necessary to describe the operational semantics. Figure 4 shows the implementation of the operation `StaLta:consume(value)`*. When simulating the model from Figure 5, this operation is called by the acceleration sensor for each measurement of seismic activity. This simulates a data stream from the sensor to the connected STA/LTA filter. The filter computes a* short term average *and a* long term average *over the arriving values. To realise this, a* `StaLta` *instance uses two instances of the utility class* `MovingAverage` *to store values and compute averages over a user-defined number of measurements (*`time`*). The size of the averages is provided by the language user through* `StaLta`*'s attributes. After calling* `MovingAverage:move(value)` *on both averages,* `StaLta:consume(value)` *checks if seismic intensity raises by comparing both averages. A significant raise indicates an earthquake. In such cases, the filter calls* `consume(value)` *on the connected detection warning sink. Based on the metamodel and the operation descriptions, we can derive a simulator for early warning systems described in this version of the stream-oriented language.*

In the example, both editor and simulator are automatically generated from language descriptions based on the according frameworks. This allows to build language tools fast, but also requires later manual work to integrate tools tightly. On the one hand, we need the flexibility to choose from several frameworks depending on concrete syntax (e.g. graphical or textual syntax) and language semantics (e.g. operational or translational semantics). On the other hand, not every framework is already integrated with every other framework. Therefore, integrating frameworks or generated tools will be a necessary burden of applying agile language engineering. Furthermore, we cannot expect that all tools can be fully automatically generated. Specific language requirements will always require manual alterations of the generated tools.

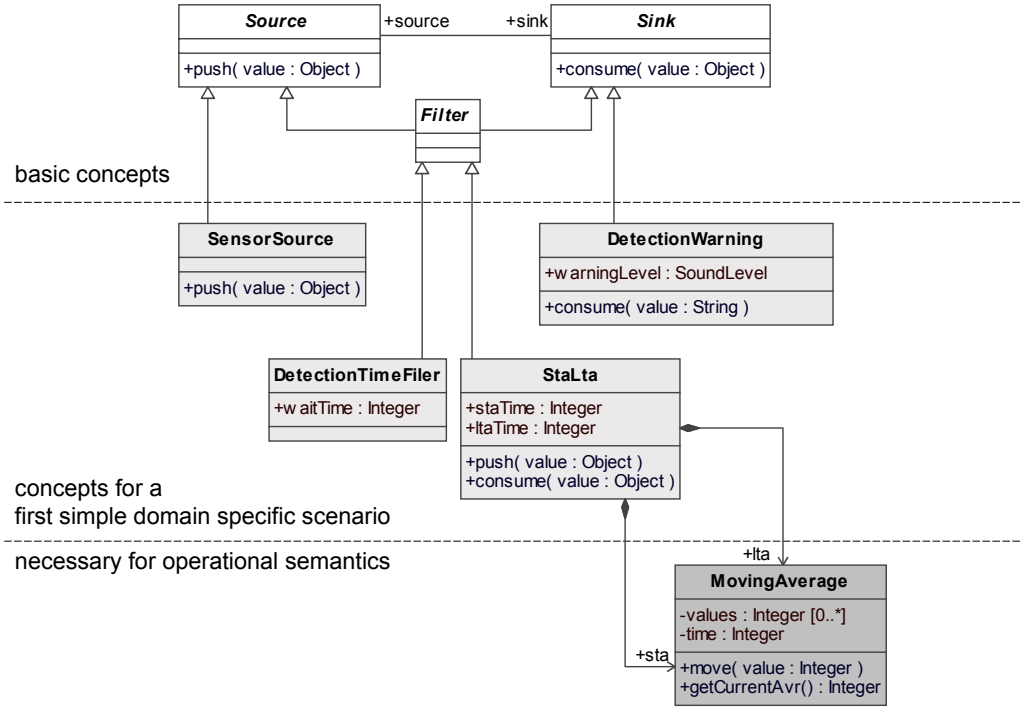

Figure 3: A metamodel with operations for the stream-oriented language.

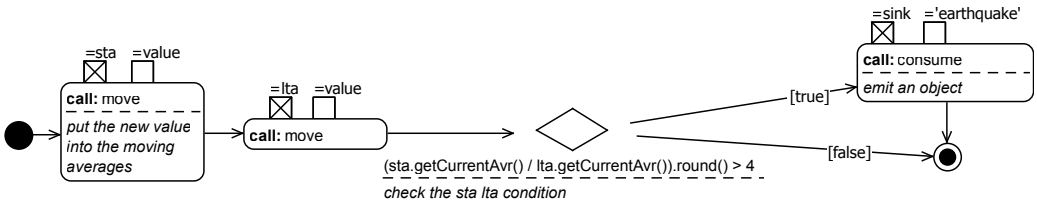

Figure 4: Behaviour description for the operation `StaLta.consume(value)` in the metamodel.

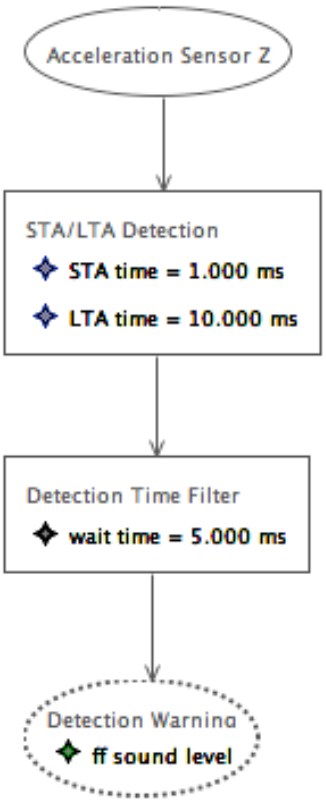

Figure 5: Screenshot of an editor generated with GMF. It shows the first model expressed in the stream-oriented DSL.

4.4 Test Support for Languages

Testing is one of the most important techniques to assess the quality of a system [8]. The test execution is sub-divided in test cases. The basic structure of each test case is as follows: A sequence of test input stimuli is fed into the system under test (SUT). Then, the actual behaviour of the SUT is compared to the expected behaviour. If they are different, then the test case detected a fault. The set of all test cases for a SUT is a test suite. Coverage criteria are used to define a certain level of quality for a test suite [51]. They can also be used to define so-called test goals that are used for test case generation [21].

There are several ways to integrate testing in the overall engineering process. The Agile Manifesto proposes *early testing*. For example, this is realised in extreme programming [6], where a test is written prior to the function of the SUT it tests.

4.4.1 Testing in Agile Language Engineering

In this section, we suggest how to apply the mentioned aspects of testing to *agile language engineering*. In the case of a language, the SUT consists of the language description and language tools. We test the different aspects of a language separately: we create tests for (1) the abstract syntax, (2) the concrete syntax, (3) the semantics, and (4) the user acceptance of the language tools. Note that these tests are executed in each iteration of the *agile language engineering* process.

(1) The abstract syntax specification of a language has to be tested in order to validate its expressiveness. One possibility to do this is by giving positive and negative example instances of the language. Each example instance forms one test case. When executing test cases, it is checked whether positive (negative) example instances do indeed (not) comply to the abstract syntax of the language. A tool realising this approach is MMUnit [45]. It provides an editor for specifying example instances of the language under test. MMUnit is an Eclipse plug-in based on EMF. It generates JUnit test cases that can be easily integrated in an overall testing process.

(2) The concrete syntax of a language manifests in an editor. Consequently, tests for the concrete syntax description are GUI tests. These GUI tests can be based on a capture-replay-mechanism, which records the manually executed test cases of the domain expert to repeat them automatically.

This technique has drawbacks if the design of GUI components is changed often. Another way to implement automatic GUI tests is the automatic generation of test cases by analysing the concrete syntax description and the used GUI generators. There are some tools that support automatic GUI testing, e.g. Abbot [2].

(3) The tests for the semantics of a language depend on how the semantics is described. Here, we deal with two kinds of semantics descriptions: *interpretative* and *transformative*. With an *interpretative* approach, language instances can be interpreted and executed directly. For testing, we create an example instance and input data, feed them into the interpreter, and compare expected and actual behaviour. With a *transformative* approach, language instances are translated into an instance of another language, the target language. This transformation has to be tested, e.g. with an automated approach [10]. If the target language's instance is executable, the result of the transformation can be tested, as well—like with an interpretative approach. For some approaches of semantics descriptions, test cases can be automatically derived. For instance, when activity diagrams and OCL describe the semantics, we can use approaches that generate test cases from UML flow graphs and OCL [29, 58].

(4) All of the preceding test approaches can be automated to a certain extent. The user acceptance tests, however, have to be performed manually by a user. Informal aspects like the design (layout, colour) of dialogues can influence the acceptance of language tools.

4.4.2 Evaluation of Tests

A test suite has a certain fault detection ability—depending on the test cases included. Since testing cannot prove the absence of faults, this fault detection ability is used as a notion for quality. The quality of the test suite is measured with coverage criteria. Although there is no proof for a relationship between coverage criteria and fault-detection ability, the satisfaction of certain criteria is widely accepted as sufficient—even for safety-critical systems. Coverage criteria are classified based on their foundation: structural [30], functional [30], or fault coverage [19, 38], for instance. The language aspect dictates the coverage criteria:

(1) Tests of the abstract syntax of a language target language structure. Consequently, for such test cases we apply structural coverage criteria, e.g.,

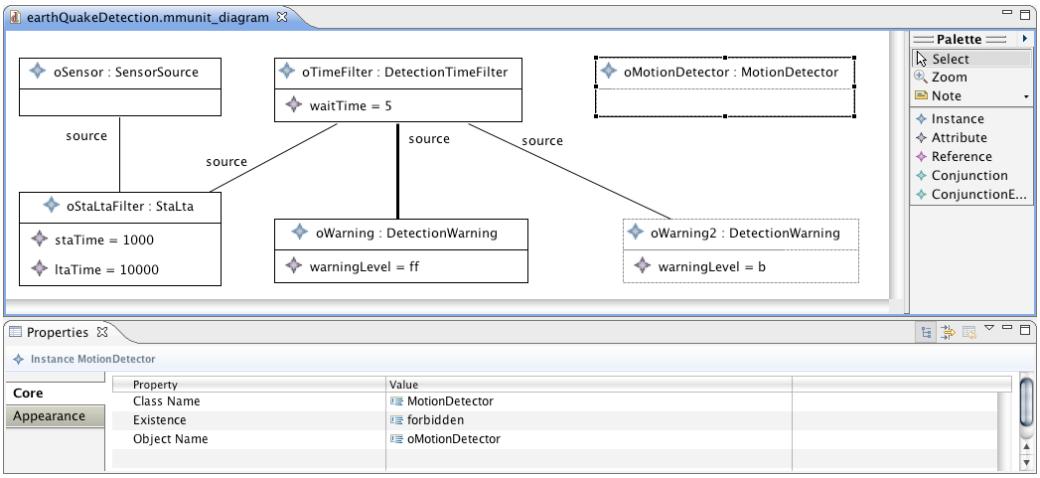

Figure 6: A screenshot of a test model in MMUnit.

Model Fragment Coverage Criterion [10], *Association-end Multiplicity Criterion*, or *Class Attribute Criterion* [5].

(2) Coverage criteria for concrete syntax tests depend on the nature of the syntax. For instance, for a graphical concrete syntax, coverage criteria for GUI tests can be used: e.g., *Event Coverage* and *Invocation Coverage* demand that all components of a GUI are invoked at least once [33].

(3) Coverage criteria for language semantics tests depend on the nature of the semantics. Executable languages require functional criteria [30], for instance. One such criterion is *Modified Condition Decision Coverage* [11]. It is widely accepted, e.g., required for software quality in airborne systems and equipment certification (standard RTCA/DO-178B). For other kinds of semantics specifications (e.g., rule-based), other coverage criteria are appropriate: e.g., *Rule Coverage Criterion* or *Context-Dependent Rule Coverage Criterion* [26].

(4) As mentioned above, user acceptance depends on informal aspects like the design of language tools. Therefore, coverage criteria cannot be applied.

Example 7 (Use of MMUnit to test the abstract syntax) *In our example, the language engineer uses MMUnit to test the abstract syntax of the stream-oriented language. The stream-oriented language allows to describe connected sources and sinks. One restriction is that each source must have exactly one sink. To test this restriction, the language engineer defines positive and negative example instances of the stream-oriented language in a so-called*

test specification (Figure 6): The test specification contains arbitrary and forbidden elements—marked with a dashed and a thick border, respectively. The test specification enforces that each source has exactly one sink by declaring that oWarning2 *(a second sink of* oTimeFilter*) is forbidden. Since the necessary multiplicities between* Source *and* Sink *are missing in the designed metamodel (cf. Figure 3), MMUnit does not reject the forbidden model. This is a fault. Consequently, the metamodel has to be corrected.*

Example 8 (Application of structural coverage criteria) *We use the coverage criterion* Class Coverage *with the MMUnit test specification: it covers 100 % of the non-utility classes of the metamodel from Figure 3. Now, we apply the coverage criterion* Class Attribute Coverage*: Because an instance of class* Sink *can reference an instance of class* Source *and there are 3 non-abstract sub-classes of* Sink *and 3 of* Source*, there are 9 possible combinations for an instance of* Sink *referencing an instance of* Source *via the attribute* source*. Since the test specification uses only 3 of them, it reaches only about 33 %.*

Example 9 (Application of functional coverage criteria) *We use coverage criteria for a certain kind of flow graph: activity diagrams (Figure 4). For testing the stream-oriented language, domain expert and language engineer create a test set-up for* `StaLta::consume(value)`*. This set-up contains a sequence of low input values. Consequently, domain expert and language engineer observe that the STA/LTA condition is always evaluated to false but never to true. Therefore, the test uses just 5 out of 6 transitions and reaches about 83 % Transition Coverage. In the following, domain expert and language engineer add a high test input value to the test input sequence so that the STA/LTA condition is evaluated to true. Therefore, the test covers 100 % of the transitions.*

4.5 Language Adaptation

In an agile development process, change is encouraged. This holds for *agile language engineering* as well. In this section, we discuss causes and locations for language changes, the meaning of change to languages, and possibilities to support change in an *agile language engineering* process.

4.5.1 Language Change

Once a language is used, either in projects or in its tests, it becomes subject to change. Actually, development is permanent change: Alternative designs are explored in order to meet requirements. Well-known solutions are customised for particular problems. New requirements are implemented and old requirements may change. Thus, the language has to change. Redesign arises due to a better understanding or to facilitate reuse. Errors are discovered and corrected. Thereby, the language is typically changed stepwise in a manual ad hoc fashion [25].

Change affects all aspects of a language: For instance, language usage might reveal new requirements on the concrete syntax. As most requirements relate to language concepts, the abstract syntax is subject to frequent change due to evolving requirements. The same holds for language semantics since requirements related to a language concept are commonly concerned with its meaning as well. Thereby, changing one aspect of a language affects the others. This holds particularly with regard to the abstract syntax of a language. Here, changes are usually propagated to the highly related concrete syntax and to semantics descriptions typically expressed in terms of abstract syntax constructs.

Furthermore, language change propagates along the meta-dimension [16] causing inconsistencies between the language and its instances. In general, language instances co-change with the syntactical aspects of the language. Additionally, instances might co-change with language semantics in order to preserve the original semantics.

In addition to instances, co-change affects language tools. These tools contain some aspects of a language implicitly. For example, parsers define the concrete and abstract syntax implicitely. Editors include at least a concrete syntax definition. Additionally, an editor might include an abstract syntax for outline views and static semantics for semantic analyses. Tools have to co-change with language specifications concerned with those aspects. For purely generic approaches, co-change is a non-issue. Here, tools are fully specified by the language description itself.

In other approaches, tools may be changed independently by hand causing inconsistencies between tools and language specifications. These inconsistencies often cause irremediable erosion where language specifications or tools are not longer updated [16].

4.5.2 Agile Adaptation

As we mentioned before, language change arises permanently. In *agile language engineering*, language change is accepted and systematically managed. Thereby, language erosion can be avoided.

For software engineering, automated transformations for code refactoring [18, 37] enables agile processes by handling dependencies and widespread changes. A refactoring guarantees behaviour preservation and states explicit properties of a change performed in terms of pre- and postconditions. Nowadays, automated refactoring is integrated in modern IDEs for various programming languages.

For *agile language engineering*, transformational *language adaptation* is essential. An adaptation defines several pre- and postconditions as well as its effect in terms of a transformation step. Each adaptation ensures particular semantics preservation and instance preservation properties [25, 56]. These properties define the quality of change and indicate the need for co-adaptation. Tool support for adaptation exists for grammar-based [28] and model-based language descriptions [55].

Each adaptation indicates co-adaptation needed for instances and other language definitions. In most cases, co-adaptation can be achieved automatically. Automated co-adaptation can be provided for language instances [27, 56] and semantics descriptions [31, 32]. Hence, language descriptions and instances are kept in sync and language erosion is prevented. Furthermore, transformational adaptation makes language change explicit and provides documentation and traceability. For some change, manual adaptation remains feasible. In this case, we advice small changes in order to keep co-adaptation manageable and to avoid language erosion.

Once a language description is adapted, either by hand or by tools, the development cycle starts over. In order to make the language run, other language descriptions and language tools might need co-adaptation. This is achieved by automated co-adaptation steps, by manual adjustment of generic solutions, or by completely manual co-adaptation. Then, the new language needs to be tested again. The language instances in the test set have to be co-adapted. Thereby, some instances might get obsolete or new test cases might be needed. Change, adaptation, and co-adaptation remain manageable due to small adaptation steps and frequent repetition of the development cycle.

Example 10 (Adaption of the stream-oriented language) *The stream-oriented language can be extended in several ways: First, connections between*

sources and sinks become explicit. Therefore, the language engineer turns the corresponding association into a class. The adaptation of the metamodel and the co-adaptations of the semantics description and of language instances are achieved completely by automated transformations. Then, sources are allowed to connect to more than one sink and vice versa. This is done by generalising the corresponding association with help of an automated transformation. Language instances are automatically co-adapted. Since the language engineer has to define the meaning of possibly new instances, language semantics need to be extended manually. In a further step, the language user wants to integrate warnings originating from several sensors by a consensus algorithm. Therefore, the language engineer introduces consensus filters as a new language concept. Again, adaptation is achieved automatically. It does not invalidate language instances. Semantics are extended manually to specify the meaning of the introduced concept. Next, the language engineer increases the expressivity of the language by introducing filters with user-definable expressions. The language user experiments with these new filters and recognises the need for internal states and explicit buffers in these filters. Like for the introduction of consensus filters, both adaptations do not invalidate existing language instances. Semantics need to be extended manually for the new language constructs.

5 Related Work

5.1 Community Process

Most multi-purpose languages are developed within a community process. Multi-purpose languages are used by a larger community. A community process allows to react to developments and flows within that community. Examples are the development of UML recommendations within the OMG or the advancement of Java through the Java community process. Community processes are heavyweight processes. Specification and implementation of language and language tools are separated. Scope and size of languages and process cause many problems. The needs of many people have to be unified. This requires time and lots of compromises. Backward compatibility is always a limiting factor. There are often wanted or unwanted discrepancies between language specification and tools from different vendors implementing it. But, the scope and size of community processes are also its biggest advan-

tage. For example, the bigger market for multi-purpose languages allows for manual implementation of very sophisticated tool-support. *Agile language engineering*, as proposed in this paper, is the contrary form of language engineering and relies on cheaply deriving tool-support from declarative language models.

5.2 Internal DSLs

Using internal DSLs almost feels like writing in a new language—although you are just using a normal multi-purpose programming language. There are particular programming languages that are suited for internal DSLs, because these languages provide a very flexible concrete syntax and extensive metaprogramming facilities. Example languages are Ruby, Smalltalk, or Lisp. All these languages are vividly used for internal DSLs.

Besides existing programming languages, there are several metaprogramming languages based on MOF-like metamodelling mechanisms, that can be used for internal languages or are tightly connected to external language development. Such languages are used in the XMF framework [14], the Meta Programming System (MPS) [15], and Kermeta [50].

Internal DSLs are an inevitable consequence of agile engineering and continuous refactoring. Engineering with internal DSLs can be understood as a weak form of *agile language engineering* that requires the language user to be a software engineer.

5.3 Language Workbenches

In contrast to internal DSLs, external DSL development requires to build languages with their own syntax and all tools necessary. To ease the development of such languages and allow rapid prototyping for languages, language workbenches and language frameworks offer sets of tools and metalanguages to describe different language aspects and create tools from these descriptions.

Examples for these workbenches and frameworks based on grammars are the ASF+SDF Meta-Environment [52], LDL [40], and The Eclipse IMP [22]. Model-based frameworks are GME [3] or XMF [14] (originated in the MMF approach [13]), and metaprogramming facilities like MPS [15], kermeta [50], MetaEdit+ [34], AToM3 [35], AMMA [42] and ATL [23], or openArchitectureWare [1]. Other workbenches and frameworks are based on graph gram-

mars and graph transformations are Graph REwrite And Transformations (GReaT) [12], DiaGen [53], Moses [41], or Tiger [49].

These workbenches and frameworks can be used for *agile language engineering* or to realise the vision of *language-oriented programming* [15]. Language-oriented programming combines multiple domain-specific languages and their development with the actual software development. When an abstraction is identified, it is directly integrated into the project's DSLs and used right away. Creating DSLs becomes a daily habit as creating APIs, libraries, or classes is in today's software engineering. *Agile language engineering* can complement agile software engineering using language-oriented programming.

6 Conclusion

6.1 Contribution

In this paper, we applied agile principles to language engineering. We pointed out the different roles in language engineering, typical development scenarios, and consequences for user collaboration throughout the engineering process.

We exemplified *agile language engineering* for the development of a stream-oriented language from the domain of earthquake detection. Based on this example, we showed how *agile language engineering* can be applied by utilising existing language engineering techniques. We investigated solutions to efficiently develop language tools. We are concerned with the testing of language aspects and the evaluation of language tests. Stepwise language adaptation was discussed as a way to control frequent language change and to avoid erosion between language specifications, tools, and instances. In general *agile language engineering* can allow to efficiently engineer languages that change frequently. Therefore, this approach is particular useful for developing DSLs.

Thereby *agile language engineering* presents only engineering process principles and the general course of action. For every language, specific language frameworks have to be combined to find a specific agile language engineering process. We want to stress that the presented examples and uses meta-languages and tools only constitute an example application of agile language engineering principles.

6.2 Future Work

The agile principles discussed and applied to language engineering in this paper are only a first step towards *agile language engineering*. Now we need formal studies, involving real world languages. This does not only concern the language engineering itself, but also the software engineering that is based on the developed languages. For user collaboration, further investigations about the interactions between agile language engineering processes and agile software engineering processes are needed. Based on the general process presented in this paper, more concrete agile processes have to be designed and evaluated.

We listed some of the growing number of language frameworks and workbenches. To allow the efficient development of highly integrated tool chains, we need to find generic interaction and integration mechanisms between different language engineering frameworks and concrete language tools. Language workbenches are a beginning, but the possibilities to integrate tools created with different technologies are limited. The integration of frameworks and tools for several language aspects also influences automatic tests for language tools and adaptation and automated co-adaptation. Furthermore, frameworks for automated tests and co-adaptation have to be developed for all the used meta-languages and must also be applicable to manually altered generated language tools.

References

[1] openArchitectureWare, 2008. See `http://www.openarchitectureware.org`.

[2] *Abbot*, 2008. See `http://abbot.sourceforge.net/doc/api/abbot/doc-files/about.html`.

[3] Aditya Agrawal, Gabor Karsai, and Akos Ledeczi. An End-to-End Domain-Driven Software Development Framework. In *OOPSLA'03*. ACM Press, 2003.

[4] Marcus Alanen and Ivan Porres. A Relation between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, TUCS, 2004.

[5] Anneliese Amschler Andrews, Robert B. France, Sudipto Ghosh, and Gerald Craig. Test adequacy criteria for uml design models. *Softw. Test., Verif. Reliab.*, 13(2):95–127, 2003.

[6] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[7] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development. February 2001. See `http://agilemanifesto.org/`.

[8] Boris Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[9] Jean Bézivin and Reiko Heckel, editors. *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings*, 2005.

[10] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *ISSRE'06*, pages 85–94. IEEE, 2006.

[11] John J. Chilenski and Steven P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. In *Software Engineering Journal*, volume 9, pages 193–200, September 1994.

[12] Alexander Christoph. Great: Uml transformation tool for porting middleware applications. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 212–219. CSREA Press, 2004.

[13] Tony Clark, Andy Evans, Stuart Kent, and Paul Sammut. The MMF Approach to Engineering Object-Oriented Design Languages. In *LDTA'01*, April 2001.

[14] Tony Clark, Andy Evans, Paul Sammut, and James Willans. *Applied Metamodeling, A Foundation for Language Driven Development*. Xactium, 2004. See `http://www.xactium.com`.

[15] Sergey Dmitriev. Language Oriented Programming: The Next Programming Paradigm. *onBoard*, (1), November 2004. See `http://www.onboard.jetbrains.com/is1/articles/04/10/lop/`.

[16] Jean-Marie Favre. Meta-model and model co-evolution within the 3D software space. In *ELISA'03*, pages 98–109, September 2003.

[17] Joachim Fischer, Michael Piefel, and Markus Scheidgen. A metamodel for sdl-2000 in the context of metamodelling ulf. In Daniel Amyot and Alan W. Williams, editors, *SAM*, Lecture Notes in Computer Science, pages 208–223. Springer.

[18] Martin Fowler, Kent Beck, John Brant, William F. Opdyke, and Donald B. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, June 1999.

[19] Phyllis G. Frankl and Elaine J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, 1993.

[20] *Eclipse Graphical Modeling Framework*, 2008. See `http://www.eclipse.org/gmf`.

[21] I-Logix. *Rhapsody Automatic Test Generator, Release 2.3, User Guide*, 2004.

[22] IBM Research. *The Eclipse IMP*, 2008. See `http://www.eclipse.org/proposals/imp`.

[23] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Model Transformations in Practice*, volume 3844, pages 128–138. Springer, 2005.

[24] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *TOSEM*, (3):331–380, jul 2005.

[25] Ralf Lämmel. Grammar adaptation. In José Nuno Oliveira and P. Zave, editors, *FME'01*, volume 2021 of *LNCS*, pages 550–570. Springer, 2001.

[26] Ralf Lämmel. Grammar testing. In *FASE'01*, volume 2029 of *LNCS*, pages 201–216. Springer, 2001.

[27] Ralf Lämmel and Wolfgang Lohmann. Format evolution. In *RETIS '01*. OCG, 2001.

[28] Ralf Lämmel and Guido Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. *ENTCS*, 44(2), 2001.

[29] Leirios. LTG/UML. See `http://www.leirios.com`.

[30] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akadamischer Verlag, 2002.

[31] Wolfgang Lohmann and Günter Riedewald. Towards automatical migration of transformation rules after grammar extension. In *CSMR '03*. IEEE, 2003.

[32] Slavisa Markovic and Thomas Baar. Refactoring OCL annotated UML class diagrams. In Lionel C. Briand and Clay Williams, editors, *MoDELS'05*, volume 3713 of *LNCS*. Springer, October 2005.

[33] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for gui testing. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 256–267, New York, NY, USA, 2001. ACM.

[34] Meta Case. *MetaEdit+*, 2008. See `http://www.metacase.com`.

[35] The Modelling, Simulation and Design lab (MSDL), School of Computer Science of McGill University, Montreal, Quebec, Canada. *AToM3 A Tool for Multi-Formalism Meta-Modelling*, 2008. See `http://atom3.cs.mcgill.ca/index.html`.

[36] Object Management Group. *Object Constraint Language Specification, version 2.0*, May 2006.

[37] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.

[38] Alexandre Petrenko, Gregor v. Bochmann, and Ming Y. Yao. On fault coverage of tests for finite state specifications. *Computer Networks and ISDN Systems*, 29(1):81–106, 1996.

[39] Jan Rekers and Andy Schürr. A graph grammar approach to graphical parsing. In *IEEE 11th International Symposium on Visual Languages*, pages 195–202, sep 1995.

[40] Günter Riedewald. The LDL — language development laboratory. In Uwe Kastens and P. Pfahler, editors, *CC'92*, volume 641 of *LNCS*, pages 88–94. Springer, October 1992.

[41] Jorn Janneck Robert Esser. Moses: A tool suite for visual modeling of discrete-event systems. In *HCC'01*, pages 272–279. IEEE Computer Society, 2001.

[42] Davide Di Ruscio, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, and Alfonso Pierantonio. Extending AMMA for Supporting Dynamic Semantics Specifications of SDLs, 2006.

[43] Daniel Sadilek, Falko Theisselmann, and Guido Wachsmuth. Challenges for model-driven development of self-organising disaster management information systems. In *IRTGW'06: International Research Training Groups Workshop*. GITO-Verlag, November 2006.

[44] Daniel A. Sadilek and Stephan Weißleder. MMUnit (Unit-Tests for Meta-Models). See `http://mmunit.sourceforge.net`.

[45] Daniel A. Sadilek and Stephan Weißleder. Towards Automated Testing of Abstract Syntax Specifications of Domain-Specific Modeling Languages. In *Domain-Specific Modeling Languages (DSML'08) - associated with Modellierung 2008*, March 2008.

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

[47] Markus Scheidgen and Joachim Fischer. Human comprehensible and machine processable specifications of operational semantics. In *ECMDA'07*, LNCS. Springer, 2007.

[48] Samuel W. Stewart. Real time detection and location of local seismic events in central california. In *Bull. Seism. Soc. Am.*, volume 67, pages 433–452, 1977.

[49] Gabriele Taentzer. Tiger EMF transformation. http://tfs.cs.tu-berlin.de/emftrans, 2007.

[50] Triskell Team. *Triskell Meta-Modelling Kernel*. IRISA, INRIA, 2008. See `http://www.kermeta.org`.

[51] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[52] Mark van den Brand, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter Olivier, Jeroen Scheerder, Jurgen Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *CC'01*, LNCS. Springer, 2001.

[53] Gerhard Viehstaedt and Mark Minas. Diagen: A generator for diagram editors based on a hypergraph model. In Amihai Motro and Moshe Tennenholtz, editors, *NGITS*, pages 0–, 1995.

[54] Eelco Visser. Domain-specific language engineering. A case study in agile DSL development. In Ralf Lämmel, Joao Saraiva, and Joost Visser, editors, *GTTSE'07*, LNCS. Springer, 2007. to appear.

[55] Guido Wachsmuth. An adaptation browser for MOF. In Danny Dig and Michael Cebulla, editors, *WRT'01: First Workshop on Refactoring Tools*, pages 65–66, 2007.

[56] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In Erik Ernst, editor, *ECOOP'07*, volume 4609 of *LNCS*. Springer, 2007.

[57] Stephan Weißleder. ParTeG (Partition Test Generator). See `http://parteg.sourceforge.net`.

[58] Stephan Weißleder and Bernd-Holger Schlingloff. Deriving Input Partitions from UML Models for Automatic Test Generation. In *LNCS Volume on Models in Software Engineering (MoDELS 2007)*, 2007.

[59] Manuel Wimmer and Gerhard Kramler. Bridging grammarware and modelware. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, Lecture Notes in Computer Science, pages 159–168. Springer.