Delft University of Technology
Software Engineering Research Group
Technical Report Series

Declaratively Defining Domain-Specific Language Debuggers

Ricky T. Lindeman, Lennart C. L. Kats, Eelco Visser

Report TUD-SERG-2011-012a

TUD-SERG-2011-012a

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

This paper is a pre-print of:

Ricky T. Lindeman, Lennart C. L. Kats, Eelco Visser. Declaratively Defining Domain-Specific Language Debuggers. In Ewen Denney, Ulrik Pagh Schultz, editors, Generative Programming and Component Engineering, 7th International Conference, GPCE 2011, Portland, OR, USA, October 22-23, 2011, Proceedings. ACM, 2011.

```
@inproceedings{Lindeman-GPCE-2011,
  title = {Declaratively Defining Domain-Specific Language Debuggers},
  author = {Ricky T. Lindeman and Lennart C. L. Kats and Eelco Visser},
  year = {2011},
  booktitle = {Generative Programming and Component Engineering,
               7th International Conference, GPCE 2011, Proceedings},
  editor = {Ewen Denney and Ulrik Pagh Schultz},
  date = {October 22-23, 2011},
  location = {Portland, OR, USA},
  publisher = {ACM},
}
```

Declaratively Defining Domain-Specific Language Debuggers

Ricky T. Lindeman Lennart C. L. Kats Eelco Visser
Delft University of Technology
r.t.lindeman@student.tudelft.nl, l.c.l.kats@tudelft.nl, visser@acm.org

Abstract

Tool support is vital to the effectiveness of domain-specific languages. With language workbenches, domain-specific languages and their tool support can be generated from a combined, high-level specification. This paper shows how such a specification can be extended to describe a debugger for a language. To realize this, we introduce a meta-language for coordinating the debugger that abstracts over the complexity of writing a debugger by hand. We describe the implementation of a language-parametric infrastructure for debuggers that can be instantiated based on this specification. The approach is implemented in the Spoofax language workbench and validated through realistic case studies with the Stratego transformation language and the WebDSL web programming language.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.5 [*Testing and Debugging*]: Debugging aids; D.2.6 [*Software Engineering*]: Programming Environments; D.3.4 [*Processors*]: Debuggers

General Terms Languages

Keywords Debugging, Domain-Specific Language, Language Workbench, Spoofax

1. Introduction

Domain-specific languages (DSLs) increase developer productivity by providing specialized syntax, semantics, and tooling for building software or writing specifications within a certain domain. They provide linguistic abstractions for common tasks in a domain, eliminating low-level implementation details and boilerplate code. DSLs are most effective when supported by specialized tooling including, but not limited to, an integrated development environment (IDE) with editors tailored for the language, a debugger, a test engine and a profiler [24].

The development of a new DSL without specialized tools is no easy undertaking. A compiler or interpreter for a new language requires a parser, data structures for abstract syntax trees, and likely traversals, transformations, type checkers, and so on. To increase the productivity of users of the DSL, IDE support should also be implemented. This entails editor services, ranging from syntactical services, such as syntax highlighting and an outline view, to semantic services such as content completion and refactoring. Also, a debugger can be built for the IDE, which further increases the productivity of DSL users by allowing them to spot runtime problems and providing insightful information regarding the runtime execution flow.

This paper focuses on the development of debuggers for DSLs. Program comprehension is vital to effectively debug applications during software maintenance [20]. Therefore, a debugger that models program behavior at the correct level of abstraction will aid the DSL program comprehension process, which in turn has a positive effect on the maintenance of DSL programs.

One of the most common tasks during software maintenance is debugging. Unfortunately, DSL debuggers are often not available as the implementation effort of building a new debugger for a small language by hand is often prohibitive. First, the runtime system of the DSL must support stepping and state inspection, which is often not included in the initial DSL design. Second, the accidental complexity of the debugger IDE API makes it hard to integrate a DSL debugger into the IDE framework. Furthermore, the differences between DSL implementations make it difficult to reuse debug runtime components.

DSL engineering and language workbenches DSL engineering software assists in the development of new DSLs. Examples include parser generators, meta-programming languages and frameworks, and tools and frameworks for building the IDE for a DSL. Language workbenches are a new breed of DSL engineering tools [6] that integrate software for most aspects of language engineering into a single development environment. Language workbenches make the development of new languages and their IDEs much more efficient, by *a)* providing full IDE support for language development tasks and *b)* integrating the development of the language compiler/interpreter and its IDE. Examples of language workbenches include MPS [23], MontiCore [13], Xtext [5], and our own Spoofax [11].

A key goal of DSL engineering software is to provide a layer of *abstraction* over general-purpose programming languages and APIs that makes language engineers more efficient in building DSLs. This abstraction can be provided by graphical user interfaces, such as wizards and configuration screens that are provided by most language workbenches. It can also be provided as a linguistic abstraction, by introducing a new high-level language for defining (some aspect of) a DSL. Language engineers can then write high level *language definitions* rather than handwrite every compiler, interpreter and IDE component. Particularly successful are parser generators, that can generate efficient parsers from declarative syntax definitions [12].

According to Deursen [19] it is not easy to maintain DSL implementations, for example when the requirements of the DSL change. Language workbenches solve the maintainability issue for both DSL development and DSL by enabling the definition of DSL support tools using high level declarative languages.

Generating a debugger from a language definition Although the idea of generating a debugger from a language definition is not

GPCE'11, October 22–23, 2011, Portland, Oregon, USA.

new [10], we developed a new, low-effort approach for the development of debuggers for DSLs. In this paper we propose a generic debugger generation framework for DSLs that abstracts over the complexity of writing a debugger by hand. The framework broadly consists of three components: First, a specialized, declarative specification language for coordinating debug events in a language-specific fashion. Second, a debug instrumentation tool that instruments DSL programs based on the specification. Third, a generic infrastructure for debuggers that is designed in four separate layers in order to maximize reusability between different DSL implementations.

The contributions of this paper are as follows:

- the introduction of a high level debugger specification language
- the introduction of a four-layer infrastructure for integrating debuggers into an IDE while maximizing reuse
- the implementation[1] of these components as part of the Spoofax Language Workbench [11] and the Eclipse IDE.

We validate our approach through two case studies on two languages that were previously defined for Spoofax. First, Stratego, a program transformation language [2], and second WebDSL, a DSL for modeling web applications with a rich data model [8].

Outline This paper is organized as follows. Section 2 describes the general architecture of a debugger and discusses different implementation techniques of DSLs and their debuggers. We then show how debuggers for DSLs can be specified as part of a language definition in Section 3, which can be used for a debug instrumentation processor, as described in Section 4. We describe the runtime architecture of our approach in Section 5. Case studies are described in Section 6, and we reflect on our work, compare it to related work, and provide concluding remarks in Sections 7 and 8.

2. Debuggers

Debuggers provide an important facility for code understanding and maintenance that is often considered to be a vital part of IDEs. Bugs in software are an unfortunate but inevitable part of software engineering reality. Locating (and trying to solve) bugs is an essential part of the development and maintenance process of software. Debuggers make this process more efficient. This section will describe the general architecture of a debugger and the difficulties that arise when trying to create a debugger for a DSL.

2.1 General Architecture

Most debugger implementations consist of the following components:

- A *debugger front-end* that allows developers to control the execution flow and inspect the execution state
- An *execution context* for the language, i.e. a certain position in the source code and usually a stack trace
- An *execution state* for the language, i.e. some view of the program's state such as local variable values
- *Debug events* that are fired as the debugged program reaches a certain state

A debugger front-end interacts with the execution model of a language and is able to inspect the execution state of an application in different execution contexts during the execution of an application. A typical execution context for a language such as Java has the following scope hierarchy:

- Application: the top level scope.
- Threads: an application can be single threaded or multithreaded.
- Stack frames: for each subroutine or function call a new stack frame is created.
- Instruction pointer: the current active statement.

Debuggers can suspend the application at fixed points in the execution flow, as the application runtime fires an event indicating that it reached such a point. Using conditional and unconditional breakpoints the debugger can then decide to suspend the execution and allow developers to inspect the execution context and state.

The execution state can be inspected in terms of language-specific data structures. The availability and value of a variable is related to the previously defined scope in which the variable is defined and used.

To be able to inspect the execution context and state, a debugger has to know how the runtime state and the source code of a language are related. This means that a debugger depends on both the syntax and semantics of a language. Using debug information that contains relevant metadata (such as line number and filename) and a debug runtime library that matches the debug information to the runtime behavior of a program a debugger is able link the syntax and semantics.

2.2 Debuggers for DSLs

Debugger implementations must be specialized for the syntax and semantics of a language. Automating part of the implementation effort requires high-level specification of this variability in syntax and semantics. The remainder of this section discusses how differences in execution models, programming paradigm, and implementation approach of DSLs affect the implementation and architecture of debuggers.

2.2.1 Executability

A key design aspect relevant to debugging is the execution model of DSLs. Mernik et al. specify four kinds of executability [14]:

- DSL with well-defined execution semantics. A debugger for this kind of DSL is straightforward to implement.
- Input language of an application generator [4, 16].
- DSL not primarily meant to be executable but nevertheless useful for application generation.
- DSLs not meant to be executable [24].

As long as the semantics of the DSL are specified and the host language provides support for suspending the runtime execution, either via a debugger or emulated in the host language debug runtime library, it is possible to generate a debugger for a DSL that fits into one of the first three categories.

Non-executable DSLs are often used to formally describe domain-specific data structures. The semantics of the information is encoded in a domain-specific notation, also called jargon. Domain experts can use jargon to communicate in a non-ambiguous way. Debugging of data is not possible, but a graphical visualization (or perhaps multiple graphical visualizations) can help to comprehend the data structure.

2.2.2 Programming Paradigm

Related to the executability is the programming paradigm the DSL is based upon. Imperative and declarative programming are two high-level contrasting paradigms that influence the implementation of a debugger. Imperative programming is a style of programming in which the programmer has to explicitly specify the computations

[1] Available with nightly builds of Spoofax 0.6.1 via `http://www.spoofax.org/`.

that change a program state. In contrast, declarative programming only requires programmers to specify the logic of a computation without actually describing the control flow. DSLs often take elements from both paradigms.

A debugger for an imperative language is rather straight forward as the debugger should just follow the control flow. Although the computations and control flow is hidden when executing declarative programs, whether or not to also emit debug events to debugger to model the internal state of the computation depends on knowledge level of the users of the DSL. In addition, changing the implementation of the internal computations to emit debug events may be undesirable or impossible.

2.2.3 DSL Implementation Approach

The implementation approach of a DSL has a large impact on the DSL debugging capabilities. The effort required for debugger implementation is influenced by the availability of debugging capabilities of the runtime platform used, e.g. suspending the execution when demanded by the user and the ability to pass the execution state to the IDE. Therefore, in this section we will discuss various implementation approaches and the issues that arise when such an implementation technique is used.

DSLs can be classified as external and internal DSLs. External DSLs have their own syntax, whereas internal DSLs rely on the syntax of the host language [7]. For external DSLs, we can also distinguish DSLs relying on code generators, preprocessors, and interpretation. In this paper we focus on external DSLs, as they have their own specialized syntax a translation step that makes it possible to provide a domain-specific debugger in a partially automated fashion. For completeness, we also discuss internal DSLs and the challenges in creating domain-specific debuggers for those languages.

Compiled DSLs Compiled DSLs are fully designed for and dedicated to particular application domain. The DSL has its own syntax and is translated to a program in some target language by generating code for their host language, typically a general-purpose language such as Java. Implementing a debugger for these DSLs requires the extension of the generator with generation of debugging metadata in the output code, as described in Section 2.1. When not directly supported by the runtime system of the host language, the generator should also generate additional code to fire events. It also requires the implementation of an actual debugger front-end that processes this information, shows it in the IDE, and allows IDE users to control the execution flow.

Domain-specific language extension Domain-specific language extension is a technique that extends the host language with a domain-specific guest notation. The syntax of the host language can be extended by adding new syntactic constructs using marco-like extension, by adding a preprocessing step that transforms the guest notation back to the host language, or by implementing a proper extension of the host language compiler.

Since language extensions are assimilated to their host language, one approach to debugging language extensions is to debug at the host language level. This can lead to problems such as the execution context and state of the extension does not necessarily naturally align with the host language. Only by instrumenting the extension at the source level rather than at the target level it is projected, can extensions be effectively debugged.

Interpreted DSLs Interpreted DSLs have their own syntax and semantics. DSL programs are executed by a separately written interpreter that operates directly on the source code (or an abstract representation). Furthermore, interpreted DSL programs can be run on any platform as long as an interpreter is available. Calling an interpreter from a general purpose language is possible if they are written in the same language. Even interaction between the interpreter and the host language is possible but comes at the price of a more complicated interpreter implementation.

One of the advantages of an interpreter is that it directly operates on the program structure, this makes it easy to retrieve metadata at the DSL abstraction level. However, adding debug actions such as setting breakpoints and adding stepping supports requires adding a execution control component to the interpreter.

Internal DSLs and application frameworks Internal DSLs rely purely on the syntax and semantics provided by a general-purpose host language such as Ruby or Scala. They are distinguished from traditional libraries and application frameworks by their use of programming techniques such as fluent interfaces, operator overloading, and meta-programming capabilities provided by the host language such as template meta-programming and implicits. These features give the libraries a “language”-like feel to it, while still maintaining full integration and compatibility with the host language. Providing specialized, domain-specific tool support and statically checking internal DSLs is more difficult since these DSLs are really using general-purpose language constructs rather than specialized constructs.

For debugging, internal DSLs tend to fully rely on the debugger of their host language. While it is convenient to get a debugger for “free” with this approach, the debugger is not specific to the application domain. In particular, does not show the execution flow and data structures in a domain-specific fashion. To do this requires analysis of the source code to determine what are the “DSL parts” and integration of specialized domain-specific debugging facilities with a general-purpose debugger.

First, it is hard to distinguish between GPL and DSL execution. Second, the data structures in the GPL may not match the domain data structures. And finally, although the DSL and the GPL can share the same syntax, the semantics may differ.

2.3 Summary

In theory, creating a debugger for a DSL does not differ much from generating a debugger for a traditional general purpose language. Just as with DSLs program, programs written in a general purpose language are translated to a lower level language. For instance, Java is translated to Java bytecode and C# to CIL. It is the task of the debugger to transform the low-level runtime state back to a suitable higher level representation.

Using the execution model, the programming paradigm and the implementation approach as a means to classify a DSL shows us that there is a great variety between them. Also, each classification has its own issues regarding the creation of a debugger. An automated approach to building debuggers abstract over these issues in a language independent way.

The chosen implementation platform influences the amount of effort required and the approach to the implementation of, a DSL debugger. However, not the distance between the DSL and the implementation platform is critical but rather the distance between the host language and DSL execution model.

3. Declarative Debugger Specification

We propose a language independent debugger implementation framework that abstracts over the issues raised in Section 2.2 regarding the executability, the programming paradigm and the implementation approach of a DSL. The framework broadly consists of three components: (1) a debugger specification language called SEL that uses the language definition to define the DSL debugging model, (2) a language-parametric debug instrumentation tool and (3) a generic debugger runtime infrastructure that communicates with IDE debug services.

As a basis for our approach we use an event-based debugging model to model DSL execution flow in a generic way. The SEL specification maps the language-specific syntax and semantics to this generic event-based debugging model. The instrumentation tool interprets the specification to augment DSL programs in executable form with debugging information. Augmented DSL programs interoperate with the generic debugger infrastructure, which provides the glue to connect an instrumented DSL program to the IDE debug services and changes the program model in reaction to the received debug events.

This section describes the event-based debugging model and the specification language (1) that describes the relation between the DSL program and debug events. Section 4 discusses the implementation of the debug information extraction and event generation used by the instrumentation tool (2). And in Section 5 the language independent debugger runtime infrastructure is discussed. The infrastructure consists of the implementation of the host language dependent event-sending mechanism as well as the language independent debugger is discussed.

Debug event classes We use a set of four language-independent debug event classes to capture the runtime program behavior, following Auguston [1]. First, the `step` event models the execution order of DSL statements. However, a single statement does not have to be atomic, it can consist of multiple nested-statements. For example, a call to a subroutine can generate multiple step events at a different level of granularity. Next, the `enter` and the `exit` event model this hierarchical relation between step events. The last event is the `var` event indicating a variable is declared at the current level in the hierarchy created by the `enter` and `exit` events.

The event-based approach eliminates the need to support the generation of DSL metadata in the actual code generator component. Writing the specification requires syntactic and semantic knowledge about the DSL as language independent debug events (`enter`, `exit`, `step`, `var`) have to be linked against matching syntax constructs.

Syntax event linking (SEL) While the debug events are the same across various DSLs, the actual instrumentation strategy differs per DSL implementation. We abstract over this implementation by specifying the relation between the semantic behavior (modeled with debug events) and the syntax constructs of a DSL. The syntax-event linking language (SEL) is used to describe this relation.

The SEL specification language describes where to inject a debug event, and transformation strategies that define how to generate debug events in DSL syntax and how to extract relevant debug information such as line number, filename and current method name from the DSL program.

A SEL specification consists of multiple event definitions. A single event definition is structured as follows:

```
event eventClass at pattern
                 creates generator
                 from extractor
```

A rule of this form specifies a pattern to match one or more syntactic constructs (`pattern`), used to inject one of the four debug events (`eventClass`). The actual injection is done using transformations that are specified in the Stratego transformation language [2], using the transformations indicated by the `generator` and `extractor` names. The generator is a transformation that generates a small DSL code fragment that represents a debug event, while the extractor extracts the debug information from the selected syntax construct. Multiple definitions can exist for the same `eventClass` as long as the patterns do not overlap. This section will continue with a detailed explanation on the basic SEL syntax. We discuss the generation and extraction transformations in Section 4.

Syntax pattern matching To find specific AST nodes in a parsed program syntax construct patterns can match against a syntax construct in two ways: (1) using a syntactic category (sort) and (2) a specific AST node (constructor).

The syntax construct pattern is defined as follows:

```
Sort.Constructor
```

`Sort` and `Constructor` both point to existing identifiers in an SDF specification. Furthermore, either a `Sort` or `Constructor` can be ignored by using an underscore in the pattern. For example, `Statement._` matches against all constructors that are generated by the Statement sort, `Statement.VarDef` matches against all VarDef constructor generated by the Statement sort and `_.FunctionDecl` matches against all FunctionDecl constructors regardless of the sort.

An example SEL specification In Figure 1 a small DSL program with functions and statements is shown (left), together with its abstract syntax representation (right). Note that we show abstract syntax trees using a textual representation, using prefix constructor terms for tree nodes and indicating lists tree nodes with square brackets. Using graphical rectangles, the figure highlights function declarations and statements in the abstract syntax, marking the tree nodes that are important for the debugger.

Figure 2 shows SDF syntax definition rules[2] and a SEL specification mapping the functions and statements of Figure 2 to events. The dotted line shows how an enter debug event is matched against the `FunctionDecl` constructor using a syntax construct pattern with an explicit constructor. The solid line shows how a step debug event is matched against a category of syntactic constructs, containing the `FuncCall`, `VarAssign` and `Return` constructor using the `Statement` sort. `gen-func-enter` and `gen-stat-step` reference transformations that generate a debug event that fits into the DSL syntax. `extract-function-debug-info` and `extract-statement-debug-info` reference transformations that extract debug information from the matched syntax constructs.

4. DSL Program Instrumentation

One of the main tasks for runtime support of debugging is the incorporation of debugging information into executable DSL programs. This information can be incorporated by hand by adapting the translation step of DSLs, as outlined in Section 2.2. In this paper we use a reusable, language-parametric debug instrumentation tool instead, which uses the SEL specification to incorporate the necessary debug information.

Tool chain The debug instrumentation tool acts as a preprocessor that runs before the code generation or interpretation step of a DSL. Inspired by aspect-oriented programming , it weaves debugging information into DSL programs, again forming new, valid DSL programs. These can then be further processed by the standard code generator or interpreter used for the DSL.

Figure 3 shows the tool chain for DSL compilation or interpretation with debug information, based on the debug instrumentation tool. The first tool in the chain is the parser, which uses the syntax definition of the DSL to parse the input program. It produces an abstract syntax tree, which is used as the input of the instrumentation tool. The tool augments the abstract representation of the DSL program with debug information using the SEL specification. The result is used as the input for the standard processing pipeline of the language.

[2] Note that in SDF, production rules take the form *p* -> sort* {`cons(`*constructor*`)`}, indicating a pattern on the left-hand side and the syntactic category and abstract syntax constructor on the right [21].

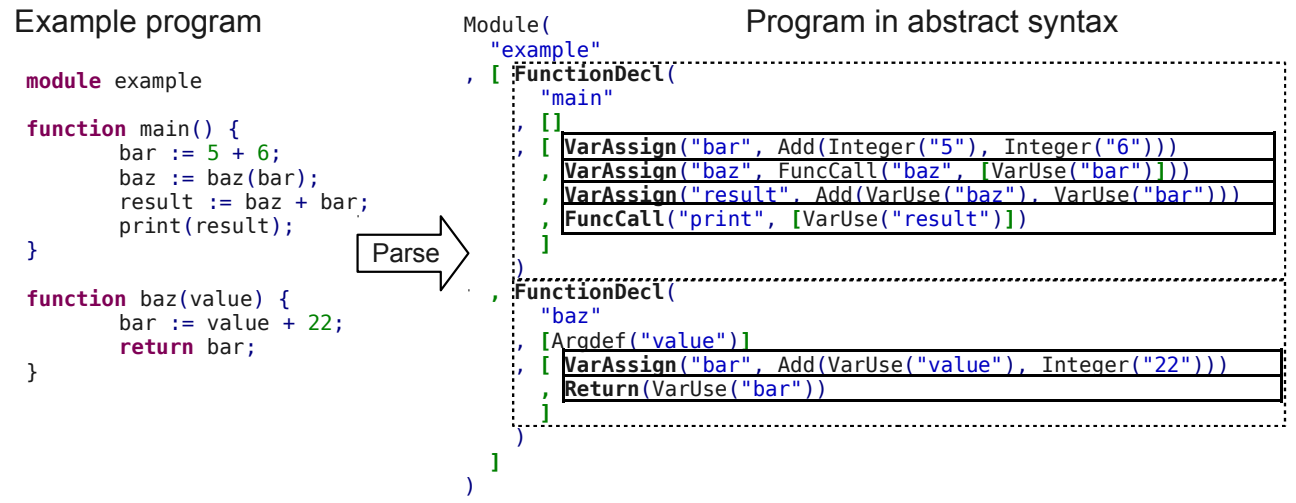

Figure 1. An example DSL Program in concrete and abstract syntax.

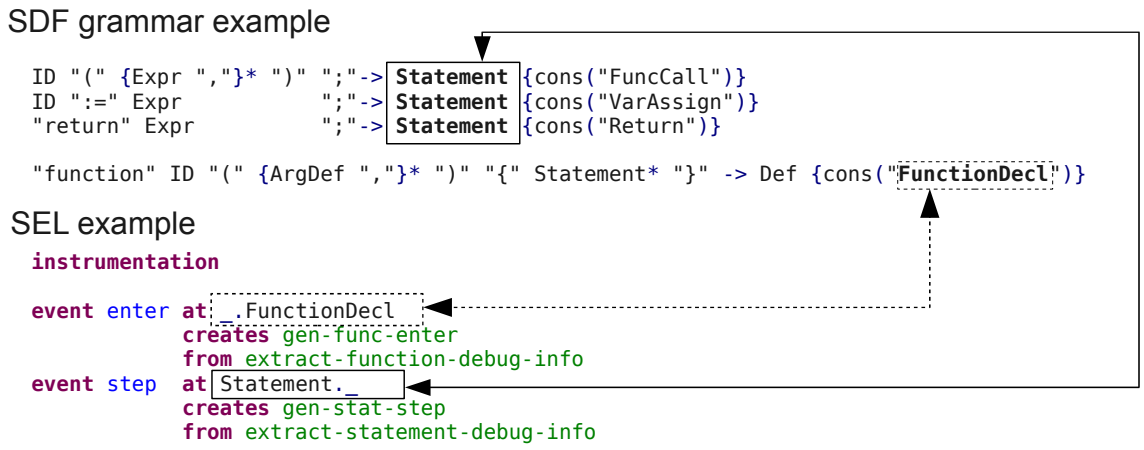

Figure 2. Relation between an SEL specification and a DSL grammar. The dotted line shows how a constructor pattern is matched against one type of constructor. The solid line shows how a sort pattern is matched against a syntactic category (sort).

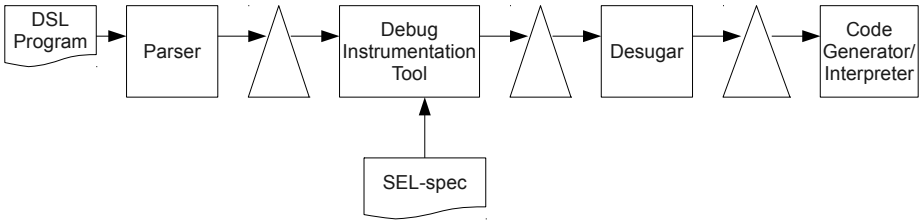

Figure 3. The tool chain for instrumentation and execution of DSL programs

The normal processing pipeline of DSLs usually starts with a desugaring step that normalizes DSL constructs to their core form. It is important that the debug instrumentation is performed before this desugaring step, to allow the debugger to operate on the original source code without losing any syntactic sugar. Finally, after desugaring, the DSL program is used for code generation or interpretation.

Instrumentation by preprocessing The instrumentation tool processes DSL programs based on a bottom-up traversal over the abstract representation. If a constructor matches a pattern in the SEL specification then the matching extraction transformation is called to extract debug information. The extractor then returns a tuple containing debug information such as the name of a local variable declaration and the line number[3] of a statement. Then the generator

[3] Note that we maintain position information in memory for the abstract representation, even though this is not shown in the textual rendition of Figure 1.

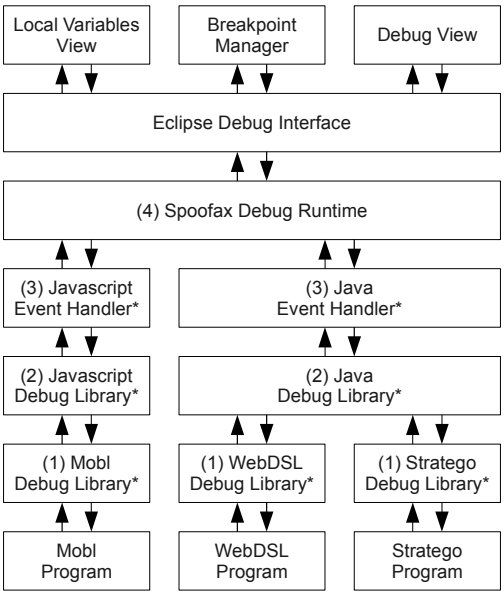

Figure 4. An overview of the integration of runtime components inside the Eclipse IDE, showing the components for the Mobl, WebDSL and Stratego DSLs. Components marked with an asterisk are specific to a DSL or DSL runtime platform.

transformation is called with the event type and the debug information as its arguments and generates a debug event statement.

`step` events are inserted before the matched statement, `var` events are inserted after the matched statement and `enter` events are inserted at the start of the method body. Inserting `exit` events is a bit more complicated when the DSL allows multiple return statements at non-fixed locations in a method body. The `exit` event cannot be inserted after the return statement, because the `exit` event will never be reached as the execution flow is already returned to the method caller. Also, just adding the `exit` event before the return statement would change the ordering of events because the return statement can contain an expression that calls another method. As an example, consider the following return statement:

```
return expression;
```

which should be transformed to something like this:

```
var temp := expression;
event(exit-event, debug-information);
return temp;
```

DSLs that support Java-like exceptions and try-finally blocks simplify the multiple exit points issue. To ensure that the `exit` event is always called for normal as well as exceptional exits the body of a method should be surrounded with a try-finally block and the `exit` event should be placed in the finally block.

Instrumenting code at the DSL level gives us the advantage that a DSL program instrumented with debug events can run on any backend for which a native debug runtime library is implemented. The implementation of the native debug library and how the debugger receives the debug events are discussed in the next section.

5. Debugger Runtime Infrastructure

In this section we discuss our implementation architecture. The architecture consists of four layers: a DSL-level debug library (1), a platform-level debug library (2), a platform-level event handler library (3), and a shared, IDE-specific core library (4). These components and their integration with the Eclipse IDE and the DSL program are illustrated by Figure 4.

By splitting up the infrastructure components into four layers, we maximize reuse: when a new DSL is developed for a platform that was previously targeted, only the DSL-level library has to be implemented (1). If a new platform is targeted, e.g. JavaScript, then the two, reusable platform-level libraries (2 and 3) should also be implemented. The common core library component (4) is neither language nor platform specific and does not require implementation unless a different IDE is used.

5.1 DSL-level Debug Library

The debug events that were added by the instrumentation tool of Section 4 are calls to the DSL debug library component. The functions in this library correspond to the four debugging event classes. They are implemented by simply forwarding the call to the platform-level debug library. The calls allow easy identification of the locations in the host locations that correspond to debug events, thus eliminating a reverse engineering step that maps fragments of generated code back to DSL code.

5.2 Platform-level Debug Library

The platform-level debug library is a lightweight library component that marshalls debug events to the platform-level event handler in the IDE. For example, for DSLs that are executed on the Java Virtual Machine (JVM), it forwards the event data from that JVM to the JVM in which the IDE runs. For DSLs that run on other platforms, or DSLs that are interpreted, a similar form of marshalling can be applied.

5.3 Platform-level Event Handler

The platform-level event handler is responsible for controlling the runtime system in which the DSL is executed. It also passes on events from the platform-level debug library to the IDE-level debug library.

Where the two previously discussed libraries are executed in the execution context of the DSL program, the platform-level event handler is a library component that operates as part of the IDE. As such, with Spoofax and Eclipse being based on Java, it is implemented in Java.

Most execution platforms provide native support for debugging. For example, the Java platform provides an API to control breakpoints and reflect over the execution state of a running Java application. This API can be used from the platform-level event handler to control the runtime of the running DSL program. For example, in our Java implementation of this library, we simply use the JVM API to set a breakpoint in the platform-level debug library to suspend it. The JVM API is also used to efficiently reflect over the runtime state, e.g. to inspect local variables.

For platforms that do not provide native debugging support, we can emulate the suspending behavior by using the platform-level debug library to pause until resumed by the IDE. Similarly, state can be inspected by manually marshalling it from the DSL level to the event handler.

5.4 IDE-level Debug Library

The IDE-level debug library is the largest component in our infrastructure, and can be shared between all DSLs and all DSL platforms. The component integrates with the Eclipse debug perspective UI, and reuses the Eclipse data structures that model the program state and the user interface elements that act has a graphical frontend for this model.

Each of the four debug event classes is processed and communicated to the Eclipse API. The `step` event is used to change the

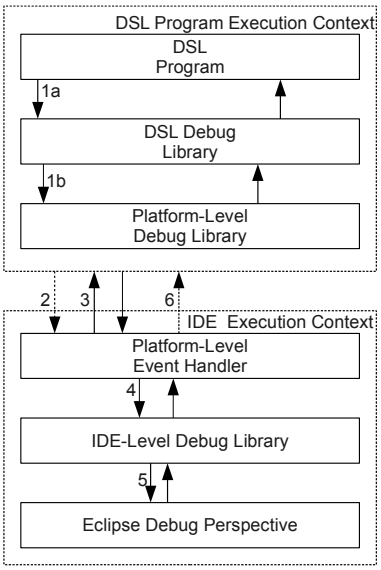

Figure 5. DSL program execution flow.

location of the instruction pointer in the top level stack frame. The `enter` event adds a new stack frame or introduces a new local scope, and the `exit` event removes the top stack frame or removes the current local scope. The `enter` and `exit` events also specify whether to push or pop a stack frame, or to only use a local scope. Finally, a `var` event declares a new variable in the current scope.

5.5 DSL Program Execution Flow

Figure 5 shows the control flow between the runtime components, the sequence is as follows:

1. (a) The DSL Program executes a debug-event sending statement, which (b) calls the matching method in the platform-level debug library.
2. A breakpoint is hit in the host language, DSL execution is suspended and the platform-level debug handler is notified of the event.
3. Using reflection, the debug information attached to the event is extracted from the suspended program.
4. The debug event is passed on to the IDE-level debug library and the DSL program state is updated.
5. The debugger compares the DSL program state against the DSL breakpoints set by the user.

At this point two execution paths are possible: If a DSL breakpoint is hit, the DSL program stays suspended and the IDE jumps to the corresponding line in the DSL program and waits for the user to select the next action (which will be discussed in the next paragraph). If no DSL breakpoint is hit, the execution is resumed (step 6 of the sequence).

When the DSL program is suspended the user can inspect the program state that was created and can take one of the following actions: *terminate*, *resume*, *step over*, *step into*, or *step out*. The implementation of the *terminate* and *resume* action is trivial. A *step* command will also resume the program execution but it will suspend the execution either when a breakpoint was hit (canceling the step request) or when the desired program state is hit. For a *step over* action, the execution is suspended once the next `step` event in the same stack frame is received or when the current stack frame is popped from the stack. For a *step into* action, the execution is suspended at the first step event that is received in the first child stack frame. For a *step out* action, the execution is suspended at the first step event originating from the parent stack frame.

Event class	Sort Category	Production Count
`step`	Strategy	58
`enter/exit`	RuleDef	3
`enter/exit`	StrategyDef	6
`var`	Strategy	58
`var`	RuleDef	3
`var`	StrategyDef	6

Figure 7. Syntax productions and events in Stratego.

6. Case Studies

We implemented our framework as part of the Spoofax language workbench [11]. To validate the implementation we performed two case studies. The first case study is performed with the Stratego transformation language and the second case study is performed with the WebDSL web programming language. A screenshot of the Stratego debugger is shown in Figure 6.

For each case study we will motivate why it was chosen and discuss the relevant issues raised in Section 2.2, followed by an overview of the components that have to be implemented in order to generate a fully working debugger.

6.1 Stratego

Stratego [2] is a transformation language used to transform program definitions using rewrite rules and traversal strategies. Stratego is actively used as a software analysis and generation tool in projects such as Spoofax [11], WebDSL [22] (a web programming DSL) and Mobl [9] (a DSL for mobile web application development).

By designing a debugger for Stratego, transformations written with the language can be stepped through and inspected. Since Stratego is also the basis for editor services in Spoofax [11], the same applies to editor services such as content completion.

Following to the categories specified in Section 2.2, we can identify Stratego as an imperative programming language with well defined execution semantics, which makes it relatively straightforward to recreate the program state during debugging with the proper debug information. When we consider the implementation approach, Stratego supports the compiled as well as the interpreted approach both available with a Java or C back-end, for this case study we focus on the compiled and the interpreted implementation approach with a Java back-end as it simplifies the implementation of the platform-specific debug library. Adapting the Java code generator to support the generation of debug information is undesirable, due to the maturity and complexity of the stratego compiler, making the debug instrumentation approach a viable solution to include debug information in Stratego programs.

Stratego debug events Stratego is a (non-pure) functional language based on the notion of strategy definitions and rules, which roughly correspond to functions in other languages, and strategy expressions, which roughly correspond to statements. Stratego also uses local variables that are assigned in match patterns. Together, these notions map naturally to the four debug events. Figure 7 provides an overview of the number of syntactic productions and their events.

Using the SEL language we can define the debug instrumentation preprocessor. First, we have to determine which syntax constructs correspond to which debug events. Second, we have to specify how we can extract the debug information from the abstract

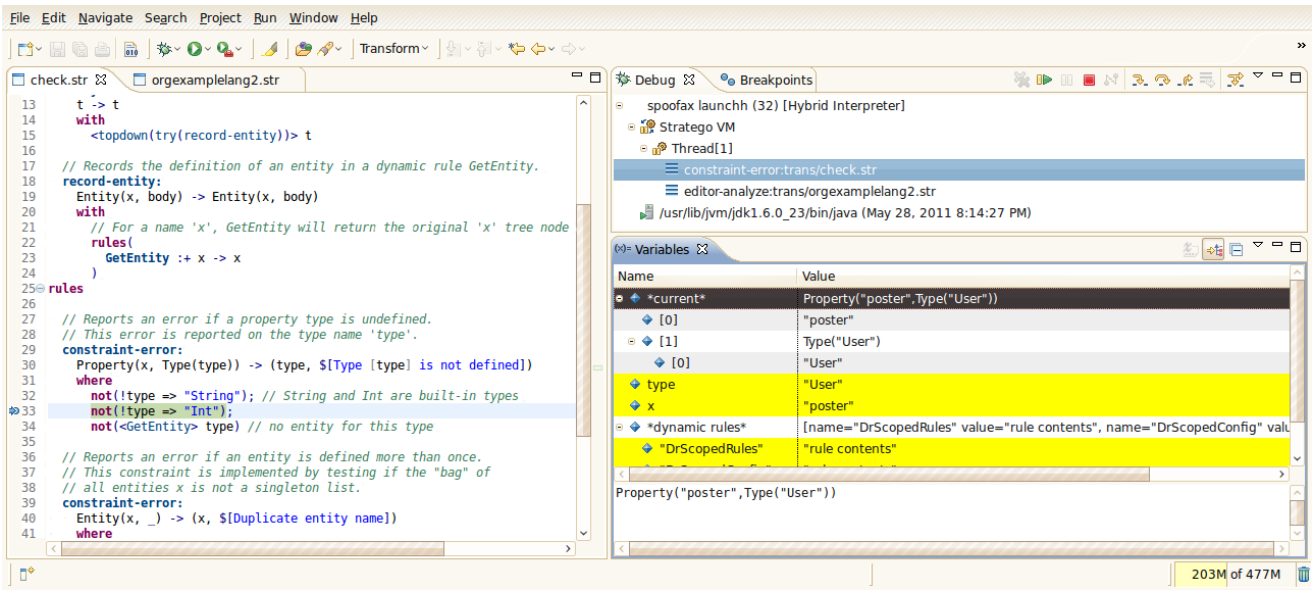

Figure 6. A screenshot of the Stratego debugger integrated in Spoofax/Eclipse. Left: Stratego editor with the highlighted current statement. Top right: Debug view showing the program state. Bottom right: Variables view showing the active variables.

```
instrumentation

event enter at StrategyDef._
            creates gen-strategy-enter
            from extract-strategy-debug-info
event exit  at StrategyDef._
            creates gen-strategy-exit
            from extract-strategy-debug-info
event enter at RuleDef._
            creates gen-rule-enter
            from extract-rule-debug-info
event exit  at RuleDef._
            creates gen-rule-exit
            from extract-rule-debug-info
event step  at Strategy._
            creates gen-step
            from extract-step-debug-info
event var   at Strategy._
            creates gen-var
            from extract-var-info
event var   at StrategyDef._
            creates gen-strategy-var
            from extract-strategy-var-info
event var   at RuleDef._
            creates gen-rule-var
            from extract-rule-var-info
```

Figure 8. SEL for Stratego

syntax tree. And finally, we have to determine how the debug information is stored.

Figure 8 shows a part of the SEL specification for the Stratego language. The `StrategyDef` and `RuleDef` sorts are the syntax constructs that correspond to strategy and rule definitions, these definition will fire an `enter` event when called and fire an `exit` event when the execution returns to the caller. The `Strategy` sort captures all statements that will generate a `step` event. The `Strategy` sort can define new variables, but variables are also used as parameters in rule and strategy signatures. The `var` event has to be matched against the `Strategy` sort but also against `RuleDef` and `StrategyDef` because parameters are allowed.

```
gen-strategy-enter:
  s -> Seq(SCallT("enter", ...), s)
```

Figure 9. Generation transformation for an `enter` event.

Figure 9 is an example of a Stratego rewrite rule that implements the `gen-strategy-enter` generation transformation referred to in Figure 8. Stratego rewrite rules have the form r`:` p_1 `->` p_2 and rewrite a term pattern p_1 to a pattern p_2. This particular rule rewrites a strategy expression tree node to one that is preceded by an `enter` event.

Instrumentation Stratego programs can be parsed as an abstract syntax tree including source code locations annotations. The location info is included for every event, the `enter` and `exit` events require the name of the rule or strategy and the `var` event requires the name of the variable as well as the value because Stratego does not allow variables to be redefined.

Rules and strategies usually have a single exit point, but if a rewrite rule fails the execution is returned to the caller just like a Java exception. Stratego supports a try-finally block only using a different notation, thus the try-finally approach from Section 4 is used to make sure exit events are always fired.

Debugger runtime Stratego allows native method calls to Java which makes it possible to implement the DSL debug library which in turn calls a Java implementation of the platform-level debug library. For instance, the `enter()` rule of Figure 9 is implemented as a Java method call to `enter(String name, ...)` in the platform-level debug library.

Reflection This case studies show that it is possible to reconstruct the stratego program state using debug events, but the actual event sending/receiving mechanism depends on the platform used by the DSL. Because we use the Java backend version of Stratego we can reuse the Java debugger for suspending the runtime and inspecting the execution state. The Java Debug Runtime component then uses reflection to extract the debug information from the suspended Java program to change the program state. The program state is then

used to determine if a Stratego breakpoint was hit using the strategy described in Section 5.4.

6.2 WebDSL

WebDSL is a DSL for web applications with a rich data model [8]. WebDSL is used as a subject to a case study because it contains imperative as well as declarative code. Defining a debugger for the imperative part of WebDSL is straightforward as it is based on the common execution model containing functions and statements. However, it is a much greater challenge to define a debugger for the declarative parts of the language. Therefore this case study will focus on defining a debugger for the declarative part of WebDSL.

WebDSL debug events WebDSL uses a declarative language for page definitions, with some support for control statements, to build the user interface of a webpage. WebDSL distinguishes page and template declarations, pages are complete page definitions while templates can be used to define reusable user interface components. Furthermore, template definitions can be locally redefined in a page or template definition only for the active definition.

During the evaluation of a page definition, the web page is outputted incrementally to a stream. These definitions can be debugged, but that would actually result in debugging the creation of the UI, not debugging the UI proper. Nevertheless, we use `enter`/`exit` events for page definitions and templates (parts of pages that can be included).

Template and page definitions are a mixture between control flow statements, basic user interface components and calls to template definitions. Not only do the debug events serve as actions that change the runtime state, a trace of debug events also show how a webpage is build. The `enter` event is fired as a new nested user interface element is created, while the `exit` event makes the current level final. The `step` event either models a control sequence, a call to a template that generates a subelement of the user interface or a call to generate a basic user interface element.

Debugger runtime WebDSL is implemented based on Java, which means we can reuse both the platform-specific event handler and the platform-specific debug library we also used for Stratego. A minor difference is that WebDSL applications are hosted in an Apache Tomcat environment, which must be configured before a debugger can be attached to it.

7. Discussion and Related Work

This paper presented a generic debugger generation framework for DSLs that abstracts over the complexity of writing a debugger by hand. Our work follows in a line of previous research aimed at more efficiently developing debuggers for custom languages. In this section we reflect over our work and highlight the differences to related work.

Applicability to different types of DSLs A key characteristic for debuggability of DSLs is their executability. Despite the fact that the relation between the executability of a language and the possible debugging capabilities is rather vague, it is possible to create a debugger as long as executable code is generated or code is executed by an interpreter.

Although the difference between a imperative and declarative programming language is precise, languages usually take elements from both paradigms. A debugger for a pure declarative language is hard to define as the execution model is hidden from the user. Nevertheless, the debugger can be used to show relationship between declarations is constructed.

We only evaluated our approach for interpreted and compiled DSLs. Debuggers for DSLs implemented as language extensions require special attention, as we only instrument the DSL parts and not the general-purpose parts of a program. Depending on the context in which the DSL extension is applied, additional effort is needed to integrate the debugger with a debugger for the host language.

Performance Debugging incurs additional runtime overhead in both the program build process and the runtime. Debug instrumentation is a one time penalty during code generation, while the debug runtime introduces extra overhead during program execution. The overhead we experienced in our case studies was acceptable, but a general strategy to avoid the overhead can be to employ separate debug and release builds to minimize the overhead in deployed applications.

The debug instrumentation preprocessor has to traverse each input file only once to generate a debug instrumented DSL program. Therefore, the performance overhead is linear with respect to the size of the code base of a DSL program. The debug instrumentation overhead can be decreased significantly if the DSL supports separate compilation. A change to a single file does not have to result in a rebuild of the entire DSL program, because the preprocessor can operate on a single source file without being aware of the complete application code base.

The DSL program execution performance is affected by debug instrumentation because of the extra debug statements, adding extra method invocation overhead, and because the debugger has to inspect the program state with events. While our implementation architecture may incur larger performance overhead than common with debuggers natively supported on an execution platform, it is common to see a certain degradation in performance even in the most optimized natively supported debuggers.

Tools for building debuggers The Meta-Environment [17] was one of the first tools that supported the generation of language aware editors and debuggers for end-users. The debuggers were generated using a generic debugging framework called TIDE [18] which also used debug events to create the program state in the debugger. TIDE relies on the language developer to implement a single, handwritten debug adapter component to link the DSL runtime to the TIDE system. In contrast, we use a layered architecture in order to minimize the implementation effort. TIDE also requires the code generator or interpreter to be adapted by hand, where we provide the SEL language to weave in the required changes.

Wu et al. [25] describe a grammar-driven technique to build a debugging tool generation framework from existing DSL grammars. Similar to our approach, they allow language developers to link grammatical constructs to events. However, their approach is aimed at DSLs that are specified as ANTLR grammars with semantic actions for execution. Using a technique they call grammar weaving [15], they weave debug instrumentation into these semantic actions. In contrast, our approach relies on a separate debug instrumentation tool that is not dependent on the implementation approach of the DSL. Wu et al. also provide a debugging framework at the IDE level, but they lack the layered architecture that allows for further reuse across DSLs in our approach.

The Meta Programming System (MPS) [23] is a language workbench that uses projectional editing rather than free text editing. It is notable because of its built-in support for debugging for languages that are based on its Base Language (BL), a host language inspired by Java. Most DSLs in MPS are defined as extensions of BL. By mapping to BL statements and expressions, they can use the BL debugger. MPS does not currently provide an API or infrastructure for debugging based on other languages. In contrast, our approach is independent of a particular base language, and uses a preprocessor to maintain independence of the implementation technique of the language. We also provide a reusable infrastructure for implementing debuggers for other platforms.

Libraries for building debuggers Where we use a tool-centric approach to generate parts of a debugger implementation, there is also related work in the form of libraries for building debuggers in an efficient manner. The Eclipse platform itself provides the Language ToolKit (LTK) library, which provides a layer of abstraction over common language-oriented operations based on the tooling Eclipse provides for the Java language. A similar library provided for it is the Dynamic LTK (DLTK) library (eclipse.org/dltk), aimed at dynamic languages. The IDE Metatooling Platform (IMP) [3] is a combination of a library and a set of wizard for scaffolding. While these libraries provide a framework for debuggers that abstracts over the traditional low-level implementations and provide hooks for IDE integration, they do not offer the layered architecture that we apply for DSLs and still require manual implementation of instrumentation DSL programs.

8. Conclusions and Future Work

Debuggers are an important tool in modern software engineering practice. Full tool support with debugging facilities ensures optimal productivity of developers with DSLs. The debugging framework presented in this paper ensures that debugging support can be implemented for DSLs with a minimum of effort, by abstracting over the accidental complexity of standard debugging APIs and providing an infrastructure that maximizes reuse across multiple DSL implementations.

Future work is needed to further specialize debuggers for domain-specific languages. Directions include more sophisticated visualizations of domain-specific data structures, the addition of domain-specific event classes to match the execution model of some DSLs, and providing infrastructure for hot code replacement.

Acknowledgments

This research was supported by NWO/JACQUARD projects 612.063.512, *TFA: Transformations for Abstractions*, and 638.001.610, *MoDSE: Model-Driven Software Evolution*.

References

[1] M. Auguston. Building program behavior models. In *ECAI Workshop on Spatial and Temporal Reasoning*, pages 19–26, 2007.

[2] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[3] P. Charles, R. M. Fuhrer, S. M. S. Jr., E. Duesterwald, and J. Vinju. Accelerating the creation of customized, language-specific ides in eclipse. In S. Arora and G. T. Leavens, editors, *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009*, 2009.

[4] J. C. Cleaveland. Building application generators. *Softw.*, 5(4), 1988.

[5] S. Efftinge and M. Voelter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[6] M. Fowler. Language workbenches: The killer-app for domain specific languages? http://www.martinfowler.com/articles/languageWorkbench.html, 2005.

[7] M. Fowler. Domain specific languages. http://martinfowler.com/dslwip/, dec 2009.

[8] D. M. Groenewegen, Z. Hemel, and E. Visser. Separation of concerns and linguistic integration in WebDSL. *Software*, 27(5), September/October 2010.

[9] Z. Hemel and E. Visser. Declaratively programming the mobile web with mobl. In *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011*. ACM, 2011.

[10] P. Henriques, M. Pereira, M. Mernik, M. Lenic, J. Gray, and H. Wu. Automatic generation of language-based tools using the lisa system. *Software, IEE Proceedings* -, 152(2):54–69, april 2005.

[11] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463. ACM, 2010.

[12] L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 918–932. ACM, 2010.

[13] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In R. F. Paige and B. Meyer, editors, *Objects, Components, Models and Patterns, TOOLS EUROPE 2008*, volume 11 of *LNBIP*, pages 297–315. Springer, 2008.

[14] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *Computing Surveys*, 37(4):316–344, 2005.

[15] D. Rebernak, M. Mernik, H. Wu, and J. G. Gray. Domain-specific aspect languages for modularising crosscutting concerns in grammars. *IEE Proceedings - Software*, 3(3):184–200, 2009.

[16] Y. Smaragdakis and D. Batory. Application generators. *Encyclopedia of Electrical and Electronics Engineering*, 2000.

[17] M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In R. Wilhelm, editor, *Compiler Construction, CC 2001 , April 2-6, 2001, Proceedings*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.

[18] M. van den Brand, B. Cornelissen, P. A. Olivier, and J. J. Vinju. Tide: A generic debugging framework - tool demonstration. *ENTCS*, 141 (4):161–165, 2005.

[19] A. van Deursen and P. Klint. Little languages: little maintenance? *Journal of Software Maintenance*, 10(2):75–92, 1998.

[20] I. Vessey. Toward a theory of computer program bugs: An empirical test. *Int. Journal of Man-Machine Studies*, 30(1):23–46, 1989.

[21] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[22] E. Visser. WebDSL: A case study in domain-specific language engineering. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II, Int. Summer School, GTTSE 2007*, volume 5235 of *LNCS*, pages 291–373. Springer, 2007.

[23] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. In M. van den Brand, B. Malloy, and S. Staab, editors, *Software Language Engineering, SLE 2010*, LNCS. Springer, 2010.

[24] D. S. Wile. Supporting the DSL spectrum. *CIT. Journal of computing and information technology*, 9(4):263–287, 2001.

[25] H. Wu, J. Gray, and M. Mernik. Grammar-driven generation of domain-specific language debuggers. *Software: Practice and Experience*, 38(10):1073–1103, 2008.

TUD-SERG-2011-012a
ISSN 1872-5392