Declaratively Defining Domain-Specific Language Debuggers

Version of July 19, 2013

Ricky T. Lindeman
Declaratively Defining Domain-Specific Language Debuggers

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ricky T. Lindeman
born in Nootdorp, the Netherlands
Declaratively Defining Domain-Specific Language Debuggers

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 thesis 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.

Thesis Committee:

Chair: Prof. dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor: Dr. L.C.L. Kats, Faculty EEMCS, TU Delft & Cloud9
Committee Member: Dr. ir. J.S.S.M. Wong, Faculty EEMCS, TU Delft
Committee Member: Dr. K.V. Hindriks, Faculty EEMCS, TU Delft
Preface

This thesis marks the end of my student career at the Technical University of Delft during which I learned a lot of interesting stuff and met a lot of interesting people both in and outside the lecture rooms. Of course, this thesis would not be possible without the guidance of prof. Eelco Visser and Lennart Kats from the Software Engineering Research Group. I would like to thank them for inviting me in the world of domain-specific languages, program transformations and language workbenches. Their valuable feedback and discussions lifted this thesis to higher level. I would also like to thank them for giving me the opportunity to write a paper that got accepted at the Generative Programming and Component Engineering conference in 2011 and for letting me present it at the conference. I had a blast! Finally, I would like to thank my friends, family and girlfriend for providing the necessary distraction and the occasional gentle nudge to get back to work. Thanks!

Ricky T. Lindeman
Nootdorp, the Netherlands
July 19, 2013
# Contents

**Preface** iii

**Contents** v

**List of Figures** vii

## 1 Introduction

1.1 Problem Statement ................................. 1
1.2 Methodology, Scope and Evaluation Criteria .......... 2
1.3 Contributions .................................. 4
1.4 Outline ..................................... 5

## 2 Background

2.1 Debugger Architecture .............................. 7
2.2 Domain-Specific Languages ......................... 10
2.3 DSL Engineering and Language Workbenches .......... 11
2.4 Stratego .................................... 13

## 3 Case Study: Implementing a Debugger From The Ground Up

3.1 The Problem .................................. 15
3.2 Stratego Debugger Design ......................... 15
3.3 Implementation Details ........................... 20

## 4 Debuggers for DSLs

4.1 DSL properties and the Impact on Debugger Design .. 25
4.2 Reflection on the Debugger Design .................. 28

## 5 Language-Independent Debugger Implementation Framework

5.1 Declarative Debugger Specification .................. 32
5.2 DSL Program Instrumentation ....................... 35
5.3 Debugger Runtime Infrastructure .................... 38
# Contents

6 Case Studies 43
   6.1 Stratego ................................................................. 43
   6.2 WebDSL ................................................................. 49

7 Evaluation Criteria Results 57
   7.1 Usability of the Generated Debuggers .......................... 57
   7.2 Usability of the Debugger Construction Framework ........... 70
   7.3 Language-Parametricity of the Framework ........................ 71

8 Discussion and Related Work 75
   8.1 Discussion .................................................................... 75
   8.2 Related Work .............................................................. 76

9 Conclusions and Future Work 79
   9.1 Conclusion .................................................................... 79
   9.2 Future Work ................................................................. 80

Bibliography 83
# List of Figures

1.1 DSL debugger requirements. ................................................. 3
1.2 Evaluation criteria. ......................................................... 4
2.1 Overview of the architecture of a debugger. .......................... 8
2.2 A screenshot of the Java debugger integrated in Eclipse. .......... 9
2.3 Implementation of the map, bottomup and topdown traversal strategies. ......................................................... 14
3.1 Two approaches to include debug events in a Stratego program. ......................................................... 18
3.2 A fragment of a Stratego program and its debug instrumented counterpart. ......................................................... 19
3.3 The Stratego toolchain with instrumentation step. .......................... 21
3.4 Basic Stratego fragments before and after instrumentation. ......................................................... 21
3.5 Execution flow for Stratego Debug Events. ................................................. 22
3.6 A screenshot of the Stratego debugger integrated in Spoofax/Eclipse. ......................................................... 23
5.1 Architectural overview of the language-independent debugger implementation framework. ......................................................... 32
5.2 An example DSL program in concrete and abstract syntax. ......................................................... 34
5.3 Relation between an SEL specification and a DSL grammar. ......................................................... 34
5.4 The toolchain for instrumentation and execution of DSL programs. ......................................................... 35
5.5 Exit event insertion using return expression extraction. ......................................................... 36
5.6 Exit event insertion using a try-finlly block. ......................................................... 36
5.7 The runtime infrastructure for a language-independent debugger. ......................................................... 38
5.8 An overview of the integration of the debugger architecture components. ......................................................... 39
5.9 DSL program execution flow. ......................................................... 41
6.1 Syntax productions and events in Stratego. ......................................................... 44
6.2 A part of the SEL specification for Stratego. ......................................................... 45
6.3 Extractor transformation for an enter and enter event. ......................................................... 46
6.4 Generation transformation for the enter event. ......................................................... 46
6.5 Generation transformation for the exit event. ......................................................... 47
6.6 Stratego debug library fragment. ......................................................... 48
LIST OF FIGURES

6.7 Native implementation of Stratego debug library. 48
6.8 Java platform-level debug library. 49
6.9 A WebDSL fragment with an event trace and the generated HTML. 50
6.10 A fragment of the SEL specification for WebDSL. 51
6.11 Syntax productions and events in WebDSL. 51
6.12 Generation transformation for the step event. 52
6.13 WebDSL post-instrumentation implementation. 52
6.14 WebDSL runtime debug library native class definition. 53
6.15 WebDSL runtime debug library for templates. 54
6.16 A fragment of the Java back-end debug runtime library for WebDSL. 54
6.17 Debug instrumented WebDSL fragment. 54
6.18 A screenshot of the WebDSL debugger debugging a YellowGrass application. 55

7.1 StrategoXT debug instrumentation duration per file plotted against the term-size of the source file. 59
7.2 Stratego debug instrumentation duration per phase and percentage of total duration of various projects. 60
7.3 YellowGrass debug instrumentation duration per file plotted against the term-size of the source file. 61
7.4 WebDSL debug instrumentation duration per phase and percentage of total duration of various projects. 61
7.5 StrategoXT build duration for normal and instrumented code. 62
7.6 Comparison of StrategoXT build steps duration for normal and instrumented source code. 62
7.7 Project compilation duration on normal code and instrumented code. 63
7.8 YellowGrass build duration on normal code and instrumented code. 63
7.9 Measurements on generated StrategoXT artifacts. 64
7.10 Stratego lifted call arguments example. 65
7.11 Measurements on generated YellowGrass artifacts. 65
7.12 Source code of tiny-module used in Figure 7.13. 66
7.13 Stratego Compiler runtime event overhead for the compilation of the tiny-module. 66
7.14 The Stratego compile time measurements for the EntityLang-project broken down by compiler step. 67
7.15 YellowGrass runtime event overhead per page request. 68
7.16 Call stack structure for higher order non-instrumented strategies. 69
7.17 LOC of the debugger components. 70
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 [56].

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. Language workbenches (LWBs) increase the productivity of DSL developers by integrating these tools in a single IDE. To increase the productivity of the users of the DSL, DSL-specific IDE support should also be implemented. This entails the implementation of editor services ranging from syntactical services, such as syntax highlighting and an outline view, to semantic services such as content completion and refactoring. In addition, 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 insight into the runtime execution flow.

1.1 Problem Statement

This thesis focuses on the development of debuggers for DSLs. First we should distinguish between the verb debugging and the noun debugger. Debugging is a software maintenance task which occurs when a developer wants to locate a fault in the program. A debugger on the other hand is a support tool that can be used to find the cause of the bug in the source code. The program that is being debugged is called the debuggee.

Program comprehension is vital to effectively debug applications during software maintenance [49]. DSLs improve program comprehension by using specialized notations and abstractions from the domain [29]. Thus, a DSL debugger will have a positive effect on the maintenance of DSL programs.

Although debugging is one of the most common tasks of software maintenance, DSL
1. Introduction

daspers 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.

In this thesis we will explore the possibility of creating a debugger for a DSL with a minimum of programming effort. We take into account the rise of language workbenches that integrate the development and use of DSLs into a single IDE environment [15]. Using high-level language definitions developers can specify the grammar of a DSL and various IDE components. We will try to apply this approach to defining a debugger for a DSL.

It is important that the approach can be applied to other DSLs and that the usability of the debugger should be comparable to a hand-written conventional debugger in terms of compile-time overhead, run-time overhead and functionality. Therefore the research question of this thesis is:

*Can we generate a DSL debugger that a) adds a run-time and compile-time overhead which is comparable to a conventional debugger with the same set of functions, b) using an approach that reduces the programming effort and c) is applicable to other languages?*

In the next section we will elaborate more about the methodology, scope and the evaluation criteria that will guide us to an answer of the research question.

1.2 Methodology, Scope and Evaluation Criteria

The Software Engineering Research Group\(^1\) (SERG) at the Delft University of Technology has developed several DSLs (such as WebDSL [17] and Mobl [21]) and has a language workbench under active development called Spoofax [26], (implemented as an Eclipse plug-in). Spoofax uses a mixture of SDF [20, 50] (syntax definition language) and Stratego [5] (a transformation language) to implement DSLs and DSL support tools. The IDE editor services (such as syntax highlighting, code completion and refactoring) that are available in Spoofax can be configured for each DSL using a declarative, rule-based editor descriptor languages. Under the hood Spoofax uses a language-parametric framework to support these editor services.

By building a DSL debugger generation framework on top of the Spoofax framework we can take advantage of these reusable language-parametric components and use them to implement language-parametric debugger components. But before we can determine which components of a debugger can be made language-parametric or not, we should have a clear understanding of the architecture of a debugger and the implementation effort required to build one.

\(^1\)http://swerl.tudelft.nl/bin/view/Main/WebHome
### Debugger requirements

- Support adding and removing line breakpoints before and during program execution.
- Support suspend, resume, terminate and disconnect actions.
- Support step into, step over and step out actions.
- Show the stack frames of a suspended application.
- Show the available variables per stack frame when the application is suspended.
- Selecting a stack frame should navigate to the corresponding location in the source code.

Figure 1.1: DSL debugger requirements.

Therefore, the first step will be to implement a debugger written from scratch, with the requirements listed in Figure 1.1 that integrates into the Eclipse IDE [45] to get a deeper understanding of how debuggers are implemented. The implementation of the debugger is driven by three design problems:

DP 1. How can we extract relevant debug metadata from source code?
DP 2. How can we send debug events with the debug metadata during the program runtime?
DP 3. How should the debugger handle those debug events?

The implementation of these design problems will be of major importance further on in our research when we use the handwritten debugger as a basis for a language-independent debugger generation framework[2]. We will implement this handwritten debugger for the Stratego language, since it is a language used often at SERG. Stratego is used to develop new DSLs in Spoofax and also parts of Spoofax are implemented using Stratego. A Stratego debugger would be a welcome addition to the developer’s support tool set.

After we have an understanding of the complexity of writing a debugger by hand we discuss how the properties of a DSL influence the design of a debugger and what components of a debugger can be made language-specific or language-independent while taking into account the three debugger design problems. This discussion helps us to determine what kind of DSLs can be supported by our DSL debugger generation framework.

Next, we explore the possibility of generating certain debugger components from a high-level language definition and implementing language-independent and language-parametric components to enable reuse and reduce implementation effort. The language-parametric and language-independent components together form a DSL debugger generation framework integrated in the Spoofax language workbench.

Finally, we evaluate the final DSL debugger generation framework with respect to the three criteria groups:

1. Usability of the generated debuggers
2. Usability of the debugger construction framework
3. Language-parametricity of the framework

We will refer to these three design problems using DP 1, DP 2 and DP 3.
1. Introduction

Usability of the generated debuggers

(EV1) How complete are the generated debuggers compared to a handwritten debugger?
(EV2) How easy is it for the DSL application programmer to active the debugger?
(EV3) What is the debug instrumentation duration overhead?
(EV4) What is the compile-time duration overhead of instrumented code versus non-instrumented code?
(EV5) What is the file-size overhead of the generated binaries when debug instrumentation is applied (in bytes)?
(EV6) What is the run-time duration overhead of debug instrumented programs?
(EV7) How much effort and time does it take to switch from normal to debugging mode and back?
(EV8) What is the accuracy of origin tracking in the presence of program transformations?
(EV9) Is it possible to debug programs that link to separately compiled libraries (which may not be instrumented)?

Usability of the debugger construction framework

(EV10) How much is the programming effort (in LOC) reduced comparing a handwritten debugger versus a generated debugger?
(EV11) What is the ratio of language-independent code in the framework to language-specific code (in LOC) for a DSL debugger?
(EV12) How easy is it to include the debug instrumentation phase in an existing toolchain?

Language-parametricity of the framework

(EV13) How many components are language-independent?
(EV14) How many components are language-parametric?
(EV15) To what extent is the approach applicable to other languages?

Figure 1.2: Evaluation criteria.

Figure 1.2 shows the criteria for each criteria group. These three groups overlap with conditions stated in the research question. Part a of the research question maps to “Usability of the generated debuggers”, part b maps to “Usability of the debugger construction framework”, and part c maps to “Language-parametricity of the framework”. We will perform two case studies that describe the implementation of a debugger for Stratego as well as WebDSL to get the desired measurement results for the evaluation criteria.

1.3 Contributions

The main scientific contribution of this thesis is a DSL debugger generation framework approach with language-independent and language-parametric components. The framework can be split into three subcomponents.
1. A language-parametric debug instrumentation tool driven by a high-level specification language called SEL.

2. A language-parametric debug event sending and handling library that enables reuse at the DSL implementation level.

3. A language-independent debugger front-end integrated into the Eclipse IDE.

The debugger infrastructure and the debugger specification are built on top of the Spoofax framework. Furthermore, we have implemented debuggers for Stratego and WebDSL and evaluated their performance.

This research resulted in a new approach for DSL debugger development. Therefore we decided to describe our findings in a paper to make it available for other researchers. The paper was published at the tenth Generative Programming and Component Engineering (GPCE’11) conference in Portland, Oregon, USA [33].

1.4 Outline

In Chapter 2 we discuss background theory about debuggers, DSLs, LWBs and the Stratego language. The case study in Chapter 3 describes how to implement a debugger from the ground up to gain more insight in the development of a debugger. This chapter focuses on the implementation of a debugger and not about the debugger generation. In Chapter 4 we motivate how a debugger design depends on DSL properties and reflect how the debugger from the previous chapter can be adapted to abstract over these issues. Chapter 5 introduces a language-independent debugger implementation framework that abstracts over DSL-specific issues. Actual debugger implementations for Stratego and WebDSL are presented in Chapter 6 and in Chapter 7 we evaluate our framework using the previously presented evaluation criteria and case studies. We reflect over our work and highlight the differences to related work in Chapter 8. Finally in Chapter 9 we draw conclusions and list future work.
Chapter 2

Background

The topic of this thesis is the result of merging three fields of computer science research: debuggers, domain-specific languages and language workbenches. In this chapter will give some background information about those areas. First we will discuss the general architecture of debuggers. Then we will give an overview of general DSL usage and development, followed by an overview of language workbenches.

2.1 Debugger Architecture

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.

Figure 2.1 shows a general debugger architecture overview which consists of the following components:

- The debuggee is the program that is being debugged.
- A debugger front-end that allows developers to control the execution flow and inspect the execution state.
- A debugger back-end is usually part of the virtual machine (VM) or interpreter running the debuggee, it receives requests from the front-end and answers with a response.
- An execution context of the debuggee, i.e. a certain position in the source code and usually a stack trace.
- An execution state of the debuggee, 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.
2. Background

A debugger front-end interacts with the execution model of a language via the back-end 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 multi-threaded.
- **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 depends on the execution state 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 the line number and the 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.

In addition to translating source code directly to machine code, a compiler can also perform optimizations on the generated code. This can become an issue when debugging a program with optimizations applied. For example, an optimization could cause the order of statements to be changed to make optimal use of registers. Now, does the person using the
2.1. Debugger Architecture

Zellweger [59] identified two categories of debuggers: expected behavior debuggers and truthful behavior debuggers. An expected behavior debugger acts the same for both the optimized and not-optimized version of the program, program optimizations are said to be hidden from the user. On the other hand, a truthful behavior debugger “displays how optimizations have changed the program portion under consideration or it admits that it cannot give a correct response” [59].

While most general purpose languages distribute a console-based debugger as part of their development kit, it is preferred to have the debugger integrated in the IDE. A single GPL is often supported by multiple IDEs, such languages have a specification of their debugging protocol. For instance the Java Platform Debugger Architecture [40] describes the implementation of the Java debugger. The IDE uses the specification to request useful debug information from the debugger and present it to the user in a graphical environment. Figure 2.2 shows a screenshot of the Java debugger in Eclipse. The top left part of the image shows the call stack of the suspended program, the top right part shows the variables that are
2. BACKGROUND

in the scope of the selected stack frame, the bottom right part shows the defined breakpoints
and the bottom left part shows the source code with a highlighted line that corresponds to
the location at which the execution is suspended.

2.2 Domain-Specific Languages

Common code blocks or common design patterns used by a domain are often abstracted into
frameworks or libraries so the programmer can focus on the business-logic of the applica-
tion. However, this still does not prevent the programmer from writing boilerplate code that
is not related to the domain. Removing all non-domain code and functions would really shift
the focus of the programmer to the domain logic. This is were DSLs come into the picture,
they are specifically tailored to a particular application domain in terms of expressiveness
and ease of use, leading to gains in productivity and reduced maintenance costs [37].

According to Herndon et al. [24] the need for a DSL will usually arise

“If a conceptual framework is rich enough and program tasks within the
framework are common enough, a language supporting the primitive concepts
of the framework is called for. (...) Many tasks can be easily described by
agreeing upon an appropriate vocabulary and conceptual framework. These
frameworks may allow a description of a few lines long to replace many thou-
sand lines of code in other languages.”

The concept of having a language tailored to a specific domain is not new. According to
Mernik et al. [37] in early research papers domain specific languages were also known as
application oriented languages [42], special purpose languages [54], task specific languages
[48], specialized languages [4], application languages [35] and little languages [3]. DSLs
have been around for over 20 years now as the term little languages was introduced by
Bentley in 1986 [3].

Well-known DSLs (although the developers that are using the language are probably
unaware that they use a DSL) are SQL, HTML, regular expressions, CSS, LaTeX, YACC
and VHDL.

2.2.1 Software Development: DSLs vs GPLs

This section discusses the differences between creating a software application using a DSL
and creating a software application using a conventional GPL. Van Deursen et al. [48] iden-
tified the following benefits when using a DSL:

- Domain specific code results in easier understanding of the domain semantics. Which
  in term would be easier to understand by domain experts.
- DSL programs are concise, self-documenting to a large extent, and can be reused for
different purposes [32].
- DSLs can enhance productivity, reliability, maintainability [47] [27], and portability
  [24].
2.3. DSL Engineering and Language Workbenches

- DSLs embody domain knowledge, and thus enable the conservation and reuse of this knowledge.
- DSL can incorporate domain semantic error checking (validation) and optimization [2] [7] [35].
- DSLs improve testability following approaches such as [43].

The following disadvantages can be identified when a DSL is used:

- The initial startup costs are high because a DSL needs to be designed and implemented [48], but overtime DSL costs will be significantly lower [23].
- The costs of education for DSL users are higher because of specialized or unfamiliar tools.
- The limited availability of DSLs [31].
- The difficulty of finding the proper scope for a DSL.
- The difficulty of balancing between domain-specificity and general-purpose programming language constructs.
- The potential loss of efficiency when compared with hand-coded software.
- Tool support that aids DSL program development is missing, but research in the area of language workbenches may invalidate this argument as will be discussed in Section 2.3.

Deursen et al. (2000) [48] mentions several other papers that list other comparisons: In [10], [12] and [27] DSLs are compared to other software generation techniques. A comparison based on the perspective of software maintenance is made in [47], whereas in [31] a comparison is made to other software reuse techniques.

2.3 DSL Engineering and Language Workbenches

According to Mernik et al. [37] developing a DSL is not an easy undertaking, it requires both domain and language development expertise and the development techniques are more varied because of the wide range of implementation approaches. In the early days of DSL development each component was implemented from scratch or using its own tool. Fortunately software can help us.

DSL engineering software assists in the development of new DSLs. Kats et al. [26] identify five important components used in DSL development which all have their own tools.

- A parser for the syntax.
- Semantic analysis to validate the DSL program.
2. **Background**

- Transformations that manipulate DSL programs and convert a high-level, technology-independent DSL specification to a lower-level program.
- A code generator that emits executable code.
- Integration of the language into an IDE.

Language workbenches are a new breed of DSL engineering tools [15] 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 [53], MontiCore [30], Xtext [13], and Spoofax [26].

### 2.3.1 DSL Development Support

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. For example, a LWB can generate a parser from a language definition. That same language definition can be used to write semantic analysis constraints, specify transformations for DSL programs and define syntax colouring for DSL program source files.

### 2.3.2 DSL IDE Integration

According to Wile [56] DSLs are most effective when supported by specialized tooling including, but not limited to, an IDE with editors tailored for the language, a debugger, a test engine and a profiler. IDE support for DSLs is vital for the acceptance of the DSL. Using a high-level language we can define editor services, such as syntax highlighting, reference resolving, code folding, code completion, error reporting, that are also present in well known IDEs. For example, defining syntax highlighting is just as simple as choosing a color and specifying the language element it should be applied to.

Editing of a DSL program can be done using two techniques. The first technique is to edit the DSL in its pure textual form. Before the editor services can be applied to the editor the DSL program has to be parsed to an abstract representation that can be used by all editor services. Error recovery [11, 25] enables the editor services to operate on abstract representation of the source file while it contains syntactical errors.

The second technique is syntax-directed editing, also called projectional editing which is used in projectional language workbenches [15]. In this style of editing the DSL program is persisted in an abstract representation which is not human-readable and hard to edit by hand. The editor shows a (textual) projection of the DSL program which only allows editing in valid positions in the abstract representation. The abstract representation will always be
2.4 Stratego

Stratego \cite{5} is a transformation language for transforming program definitions using rewrite rules and traversal strategies. Stratego is actively used as a software analysis and generation tool in projects such as Spoofax \cite{26}, WebDSL \cite{52} (a web programming DSL) and Mobl \cite{21} (a DSL for mobile web application development).

A Stratego program consists of a set of term-rewrite rules and strategies which operate on abstract syntax trees (ASTs) called Annotated Terms (ATerms). A rewrite rule transforms an ATerm to another ATerm by means of pattern matching while a strategy defines the order in which rules are applied. Stratego rules and strategies always operate on the current term and return a term when they succeed. When a rule cannot be applied or the strategy is not applicable to the current term it is said to fail.

A basic Stratego transformation rule looks like this:

$$r : \ t_1 \rightarrow t_2 \text{ where } s$$

Where $r$ is the name of the rule, $t_1$ and $t_2$ are first-order terms, and $s$ is a strategy expression. A rule can be applied to a term $t$ when the left-hand side pattern $t_1$ matches the term and the condition $s$ succeeds. The term $t$ will then be replaced with an instantiation of the right-hand side pattern $t_2$. If the pattern $t_1$ does not match or the strategy $s$ does not succeed the call to rule $r$ will fail. Multiple rules with the same name can be defined, when such a rule is called only one of the defined rules, that will not result in a fail, will be applied.

Complex transformations can be achieved by combining multiple rule-calls and basic strategy-operators into a new strategy. Examples of basic Stratego-operators are:

- The identity transformation $\text{id}$, which returns the current term.
- The $\text{fail}$ operator to fail the current strategy.
- $s_1$ is a call to a strategy or rule.
- Sequential composition $s_1 ; s_2$, strategy $s_2$ is executed after strategy $s_1$ is executed but only if $s_1$ did not fail.
- Deterministic choice $s_1 \leftarrow s_2$, strategy $s_2$ is executed only when strategy $s_1$ has failed.
- Local variable assignment $\text{foo} := s_1$, the outcome of $s_1$ is saved in variable $\text{foo}$, but note that once a variable is assigned its value cannot be changed.

Also, strategies can be used to define the traversal in which rules are applied to an AST. For example, the $\text{map}$ strategy applies to strategy in its argument to each element in a list. The $\text{all}$ strategy applies the strategy in its argument to all the subterms of the current term.
The bottomup strategy first recursively applies \( s \) to the the subterms of the current term and then applies \( s \) to the result. The topdown strategy first applies \( s \) to the current term and then recursively applies it to all its subterms. The all strategy is a built-in traversal while map, bottomup, and topdown are implemented in Stratego itself, Figure 2.3 shows the implementation of these traversals.

\[
\text{map}(s) : \mathbb{[]} \rightarrow \mathbb{[]}
\]
\[
\text{map}(s) : [x \mid xs] \rightarrow [\langle s \rangle x \mid \langle \text{map}(s) \rangle xs]
\]

\[
\text{bottomup}(s) = \text{all}(\text{bottomup}(s)); s
\]

\[
\text{topdown}(s) = s; \text{all}(\text{topdown}(s))
\]

Figure 2.3: Implementation of the map, bottomup and topdown traversal strategies.

Using dynamic rewrite rules [6] new rules can be instantiated during run-time capturing context-sensitive information. The following fragment shows a definition of a dynamic rule.

\[
r : t_1 \rightarrow t_2
\]
\[
\text{where rules}(dr : t_3 \rightarrow t_4)
\]

When rule \( r \) is executed the dynamic rule \( dr \) is defined. In most cases dynamic rules are used as a lookup table. For example, consider an ATerm that contains multiple class declarations and variable declarations in statement blocks that reference those classes. In the first traversal all available class definitions are collected in a dynamic rule, then in a second traversal this type information is used to determine if a variable declaration references an existing class.

The program transformation abilities of Stratego are driven by the use of the SDF [51] syntax definition language. If an SDF syntax definition is available\(^1\) for a programming language, programs written in that language can be parsed as ATerms allowing them to be used as input for Stratego programs. Note that in SDF, production rules take the form of

\[
p^* \rightarrow \text{sort} \{\text{cons}(c)\}
\]

indicating a pattern \( p^* \) on the left-hand side and the syntactic category \( \text{sort} \) and abstract syntax constructor \( c \) on the right. Patterns can contains keywords as well as references to other rules. For example, the following SDF fragment shows the production rule for an if-else construct.

\[
"\text{if}\ E\ "\ \text{then}\ S + \ "\text{else}\ S + \ "\text{end}\ \rightarrow S \{\text{cons("IfElse")}\}
\]

\( E \) and \( S \) represent sorts that correspond to an expression and a statement, respectively. The plus-symbol dictates that the \( S \) sort has to be repeated at one time.

\(^1\)A set of available SDF grammars can be found at [http://www.syntax-definition.org/Sdf/SdfGrammars](http://www.syntax-definition.org/Sdf/SdfGrammars)
Chapter 3

Case Study: Implementing a Debugger From The Ground Up

In this chapter we describe a case study that gives us a better understanding of how a debugger is implemented. First we describe why we used a case study, then we discuss the debugger design and finally we discuss the implementation details of some components.

3.1 The Problem

Because we do not have any experience with designing and implementing a debugger for a programming language we decided to create a debugger for the Stratego language with functionality as listed in Figure [1.1]. This case study gives us a better understanding of how a debugger actually works and the debugger will serve as a reference implementation when we construct a debugger using a language-independent debugger framework. The acquired knowledge will help us determine what effect the properties of a DSL will have on the debugger architecture and to determine which components of a debugger are language-specific or language-generic.

While designing the Stratego debugger we keep in mind that the architecture of the debugger will be the basis of a language-independent debugger framework. Therefore when making design decisions we will prefer a solution that does not depend on language-specific details and does not require changes to the existing language implementation, for instance modifying the compiler.

3.2 Stratego Debugger Design

A Stratego debugger that is useful during development and maintenance should support stepping through the transformations and inspection of local defined variables. Not only does it provide insight in the traversal of complex strategies, it can also be used to determine why rules are failing. Therefore frameworks such as Spoofax that contain components implemented in Stratego will be easier to maintain.
3. CASE STUDY: IMPLEMENTING A DEBUGGER FROM THE GROUND UP

As presented in Section 2.4, Stratego is a transformation language used to transform program definitions using rewrite rules and traversal strategies. Stratego supports compiled as well as interpreted programs and both are available with a Java or C back-end. Throughout this thesis, we will only focus on the compiled and the interpreted implementation approach with a Java back-end as it will simplify Eclipse integration.

A general debugger architecture overview was already given in Figure 2.1 in Section 2.1, now we need to implement each component with Stratego specific details and solve the debugger design problems listed in Section 1.2. First, we need to determine the debug information that is represented in the execution state and execution context (DP 1). Second, we need to design a communication scheme (via debug events) between the debugger and the debuggee that allows us to retrieve the debug information (DP 2). Third, we need to implement a debugger (back-end) that support settings breakpoints and controlling the execution (DP 3). Finally, we need to integrate the debugger in the Eclipse IDE, which will serve as graphical user interface, the debugger front-end.

3.2.1 Runtime Metadata

As discussed in Section 2.1, a debugger can be used to step through the execution of a program. Using breakpoints, the execution can be suspended at a specific line of code. When in suspended state, it is possible to view the program state. In the Stratego case, the program state can be modelled using a stack trace where each stack frame corresponds to a rule or strategy call. For each stack frame, we want the following runtime metadata to provide insight about the execution state and context to the user. Each stack frame consists of

- a filename that points to the file in which the rule or strategy is defined,
- a line number that points to the active statement in the stack frame,
- the name of the rule or strategy, and
- a collection of local variables that are accessible in the scope of the stack frame.

Using this runtime metadata, the user can navigate to the different source code locations and inspect the relevant execution context.

Currently, the Stratego Java runtime does not keep track of all the runtime metadata mentioned above. Most Stratego metadata is lost during the code generation phase, only rule and strategy names are recognisable in the generated Java code, the source code locations and variable names are lost. Before we can implement a debugger, we have to come up with a design on how to make the relevant metadata available to our Stratego debugger (DP 1).

Before we list the design alternatives, we note that we prefer a solution that allows the reuse of debugger components, in other words, we prefer a language-independent design. Furthermore, we limit our language-independent debugger approach to only support DSLs that use a target language with runtime metadata retrieval capabilities.

We distinguish between two high-level orthogonal design alternatives, a \textit{push} and a \textit{poll} approach: We either extract the runtime metadata that is available in the target language
program and convert it to DSL-specific runtime metadata, or we model the execution of a DSL program with events in which each event represents a state change. Applying this to Stratego gives the following alternatives:

- Use the Java debugger to suspend the Stratego program at specific locations and extract the Java program state, then use an algorithm to convert the Java program state to a Stratego program state.
- Model the execution of a Stratego program using events which are sent to the debugger, each event causes a state change in the debugger.

In the first solution the debugger uses polling to request the debug information, while in the second solution the changes in program state are pushed to the debugger. The polling approach, also used by Wu et al. [58] requires extending the code generator to also include DSL specific debug information in the generated code. When that information is available in the generated code, it can be extracted by the debugger and converted to a Stratego program state. We have to keep in mind that our debugger architecture should be capable of handling different DSL implementations without too much DSL-specific modifications. Thus, we should use an approach that limits the number of modifications to the existing language components. Therefore, we have chosen for the event-based solution.

3.2.2 Debug Events

Debug events which are fired during program runtime reflect the changes of the program state. By capturing these events the debugger is able to reconstruct the program state of the debuggee in the debugger without retrieving the complete state from the debuggee. When the program execution is suspended by the debugger, the user can inspect the program state.

We can represent the Stratego runtime behavior with four events:

1. enter: An enter event is fired before the body of a rule or strategy is executed and creates a new stack.

2. exit: An exit event is fired after a rule or strategy call returns to its caller and removes the top stack frame.

3. step: A step event corresponds to a single statement. It is used to define the location of the current statement within a stack frame.

4. var: A var event is fired when a new variable is declared in the current stack frame.

All events contain the following debug information: filename, line number, strategy or rule name, and the value of the current Stratego term. The var event also contains the name of the introduced variable.
3. CASE STUDY: IMPLEMENTING A DEBUGGER FROM THE GROUND UP

![Diagram](a) The Stratego compiler is modified to support the generation of debug events at the Java-level.

![Diagram](b) Debug events are inserted in the Stratego program via instrumentation at the Stratego-level before the compiler is called.

Figure 3.1: Two approaches to include debug events in a Stratego program.

### 3.2.2.1 Debuggee-Debugger Communication

For the debugger to receive the debug events we must design a communication scheme between the debuggee and the debugger (DP 2). We decided to use the Java Platform Debugger Architecture (JPDA) [40] to take care of the debugger and debuggee communication instead of writing a custom communication scheme that would require additional implementation effort. The Java Debug Interface (JDI) API [39] which is part of the JPDA is a high-level Java API providing information useful for debuggers and similar systems that need access to the running state of a (usually remote) virtual machine. The Stratego debugger uses JDI to connect to a running Java compiled Stratego program and when the connection is established it sets breakpoints at locations that reflect a debug events. When such a breakpoint is hit the Stratego debugger is notified and using reflection it is possible to extract the Stratego metadata. This approach resembles the polling approach from Section 3.2.1 but instead of extracting the entire Java program state, we only extract Stratego-specific metadata that is contained within the event.

This technique requires that the Java code contains source code fragments which are identifiable as debug events and contain Stratego-specific metadata. By implementing the debug events as a call to a function in an external debug library with Stratego-specific metadata contained in the arguments, the debug events can easily be identified. Setting a breakpoint in the body of such a function allows the debugger to extract the metadata from the function arguments using reflection. The remainder of this section shows how these debug events are inserted in the source code.

### 3.2.2.2 Instrumentation

We can identify two implementation approaches for supporting debug events in Stratego programs. The first thought would be to adapt the code generator and let it add appropriate debug events in the generated code. Figure 3.1a shows how this fits into the Stratego toolchain. A modified Stratego compiler (marked with a asterisk) takes a Stratego program...
3.2. Stratego Debugger Design

(a) Stratego program fragment.

```
match : Program(_, def*) -> f*
  with
    f* := <find-functions> def*

(b) Stratego program fragment instrumented with debug events. The debug events are underlined.

Figure 3.2: A fragment of a Stratego program and its debug instrumented counterpart.

as input and produces a Java program with debug events as output. This approach would require major changes to the code generator which is undesirable, due to the maturity and complexity of the Stratego compiler. Also, when keeping in mind that we want to support other DSL implementation approaches, we are better off using a technique that does not require extensive modifications to existing components.

The second approach would be to insert an instrumentation step in the toolchain where we can embed debug events in the DSL program itself. In Stratego the debug events are implemented as a call to a strategy that is part of a special debug library and the arguments of the strategy call contain the relevant metadata. This approach isolates DSL-specific implementation issues in the debug library and does not conflict with the separate debug library technique discussed in the previous paragraph. Figure 3.1(b) shows how this approach fits in the toolchain.

Figure 3.2 shows how a simple Stratego rule `match` and its debug instrumented counterpart would look like. A total of 7 debug events are inserted in the original code fragment. The debug events contain the name of the filename `fn.str`, the name of the rule `match` and the source code location is encoded in the `LI` term.
3. CASE STUDY: IMPLEMENTING A DEBUGGER FROM THE GROUND UP

3.2.3 Program Execution Control

Now that we have covered how the program state is modeled, controlling program execution by means of breakpoints can be discussed. In its most simple form a breakpoint is a request for the user to stop the execution at a specified line number in a file. More complex breakpoints such as a conditional breakpoints will only suspend execution when an expression (defined in the breakpoint) evaluates to true, but for this case study we only consider the line breakpoints. As each debug event updates the program state, the debugger has to check if a breakpoint was hit after each received debug event.

The debugger performs the following steps when a debug event is received:

1. The debugger updates the program state using the debug information in the debug event.
2. The debugger compares the filename and line number of the top level stack frame to each defined breakpoint.
3. If no match is found the execution of the debuggee can continue normally, but when a match is found the program execution of the debuggee is suspended.
4. The debugger front-end, in this the Eclipse Debug Perspective, is notified that a breakpoint was hit and jumps to the corresponding location in the source code.

When the DSL program is suspended the user can inspect the program state and can take one of the following actions: terminate, resume, step over, step into, or step out. A terminate command will terminate the execution of the debuggee. A resume command resumes the execution of the debuggee. 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.

3.3 Implementation Details

In the previous section we have talked about the various design alternatives. This led to a debugger design with debug events that will be pushed to the debugger. The debug events will be added to a Stratego program using instrumentation. In this subsection we will cover the instrumentation component, the event pushing mechanism and the Eclipse integration in more detail.

3.3.1 Instrumentation by Preprocessing

The debug events will be inserted in the Stratego program using a debug instrumentation tool. This instrumentation tool can be inserted as a preprocessor step before the Stratego
3.3. Implementation Details

![Stratego toolchain with instrumentation step](image)

Figure 3.3: The Stratego toolchain with instrumentation step.

<table>
<thead>
<tr>
<th>Before instrumentation</th>
<th>After instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{s1; s2}</td>
<td>\texttt{s-step; s1; s-step; s2}</td>
</tr>
<tr>
<td>\texttt{map(s1; s2); s3}</td>
<td>\texttt{s-step; map(s-step; s1; s-step; s2); s-step; s3}</td>
</tr>
<tr>
<td>\texttt{foo := match-bar}</td>
<td>\texttt{s-step; foo := match-bar; s-var}</td>
</tr>
</tbody>
</table>

Figure 3.4: Basic Stratego fragments before and after instrumentation.

The compiler is executed. The normal processing pipeline of Stratego programs starts with a desugaring step that normalizes Stratego 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. Figure 3.3 shows the Stratego toolchain with the debug instrumentation step. First the Stratego program is parsed and represented as an ATerm, then the debug instrumentation tool adds the debug event calls to the Stratego program and returns an instrumented ATerm. In the next step the ATerm is desugared to core-Stratego, the final ATerm is then used for code generation or interpretation.

The debug instrumentation tool is implemented in Stratego and contains several transformations that augment the Stratego code with debug event calls. For each debug event a transformation rule is defined that defines how to extract the relevant metadata and how to inject a debug event call in the source code. Figure 3.4 shows basic Stratego fragments before and after instrumentation. Note that in the figure the debug information is omitted to keep the examples readable.

The enter and exit calls for rules and strategies (\texttt{s-enter} and \texttt{s-exit} for strategy definitions and \texttt{r-enter} and \texttt{r-exit} for rule definitions) are added before and after the body of the definition. The exit events cannot simply be added as the last statement in the body because the debugger also has to handle fails.

Consider the following Stratego fragment with an explicit fail:

\begin{verbatim}
foo = s1; fail
\end{verbatim}

This fragment could be transformed to the following fragment:

\begin{verbatim}
foo = s-enter; s1; s-step; fail; s-exit
\end{verbatim}

When \texttt{fail} occurs in the body, the execution is directly returned to the caller (similar to an exception in Java), thus the \texttt{s-exit} is never fired. This will bring the program state in the debugger to an invalid state because an event is not received.

We can solve this by wrapping the body in a try-catch-finally block, called the guarded left choice operator in Stratego, which has the following syntax \texttt{s1 < s2 + s3}. First strat-
3. Case Study: Implementing a Debugger from the Ground Up

egy s1 is executed, when s1 fails s2 is executed, otherwise s3 is executed. The debug instrumented version now looks like this:

```java
foo =
    s-enter;
    (s-step; s1; s-step; fail)
< (s-exit; fail) + (s-exit)
```

### 3.3.2 Debug Event Sending

As discussed in the previous section the debug events are implemented as calls to strategies which are defined in the Stratego debug library. The strategies are implemented as native Java methods in dedicated Java classes\(^1\) that extract the debug information from the strategy call arguments and convert it to a String representation. Figure 3.5 shows how the Stratego debugger receives the debug events contained within a Stratego program: During the execution of strategy foo in module bar strategy s-enter is called in the debug-library module (1). The debug-library then calls the external strategy definition (2) which is resolved by a call to the native Java method (3). The debug-information\(^2\) is then converted to a String representation. The breakpoint placed at nothing() is hit and the Stratego JVM is suspended. Finally, the Stratego debugger is notified that the breakpoint was hit and extracts the debug information from the JVM using reflection.

---

2. The debug-information is hidden in the figure for readability.
3.3.3 Eclipse Integration

In order to integrate with the Eclipse debug perspective user interface our Stratego debugger reuses Eclipse specific data structures to model the program state and conforms to interfaces describing the navigate actions when suspended. We get the following views: A breakpoint view that shows the user-defined breakpoints, a variables view that shows the defined variables in the selected stack frame and the debug view that shows an outline of the stack frames. When selecting a stack frame the Stratego editor automatically jumps to the active line for that stack frame and highlights it. Figure 3.6 shows a screenshot of the debugger in action.
Chapter 4

Debuggers for DSLs

In the previous chapter we have described how we implemented a debugger for the Stratego language. We explored several designs but focused on a language-independent approach to support reusability across multiple DSLs. In this chapter we first discuss the properties of a DSL and how they affect the design of a debugger in general, then we will reflect on the Stratego debugger design and determine to what extent the components can be reused in a generalized debugger approach.

4.1 DSL properties and the Impact on Debugger Design

We can conclude from the previous chapter that the design of a debugger depends on the semantics of a language and its implementation approach. The runtime program metadata in the debugger reflects the semantics of a language and the implementation approach determines how that metadata can be extracted from the host language. First we classify the semantics of a language using the executability and programming paradigm of a language and then we review each DSL implementation approach.

4.1.1 Executability

A key design aspect relevant to debugging is the execution model of DSLs. Mernik et al. specify four kinds of executability [37]:

- DSL with well-defined execution semantics. A debugger for this kind of DSL is straightforward to implement.
- Input language of an application generator [9, 44].
- DSL not primarily meant to be executable but nevertheless useful for application generation.
- DSLs not meant to be executable [56].
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 implement 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.

### 4.1.2 Programming Paradigm

Related to the executability of a DSL is its programming paradigm. Imperative and declarative programming are two high-level contrasting paradigms. Imperative programming is a style of programming in which the programmer has to explicitly specify the computations 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 straightforward as the debugger should just follow the control flow. For declarative programs the computations and control flow are implicit. Therefore, the decision to emit debug events that model the internal state of the computations and control flow may depend on the knowledge-level DSL users are assumed to have about the DSL implementation.

### 4.1.3 DSL Implementation Approach

The implementation approach of a DSL has a large impact on the DSL debugging capabilities. The effort required for a debugger implementation is influenced by the availability of debugging capabilities on the platform of the DSL implementation, e.g. suspending the execution when demanded by the user and the ability to extract the execution state and inspect it in the IDE.

DSLs can be classified as external or internal DSLs. External DSLs have their own syntax, whereas internal DSLs rely on the syntax of the host language [16]. For external DSLs, we can also distinguish between DSLs relying on code generators, preprocessors, and interpretation. In this thesis we focus on external DSLs, as they have their own specialized syntax and 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.

#### 4.1.3.1 Compiled DSLs

Compiled DSLs are fully designed for and dedicated to a 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 extending the code generator with the
4.1. DSL properties and the Impact on Debugger Design

generation of debugging metadata in the output code. When not directly supported by the
runtime system of the host language, the generator should also generate additional code to
fire events. This also requires the implementation of a debugger front-end that processes
this information, shows it in the IDE, and allows IDE users to control the execution flow.

4.1.3.2 Domain-Specific Language Extensions

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 macro-like extensions, by adding a preprocessing step that
transforms the guest notation back to the host language, or by implementing a proper ex-
tension of the host language compiler.

Since language extensions are assimilated to their host language, one approach to de-
bugging language extensions is to debug the program at the host language level. This can
lead to problems 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, extensions can be debugged effectively.

4.1.3.3 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 struc-
ture, this makes it easy to retrieve metadata at the DSL abstraction level. However, adding
debug actions such as setting breakpoints and adding stepping support requires extending
the interpreter with an execution control component.

4.1.3.4 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 appli-
cation frameworks by their use of programming techniques such as fluent interfaces, opera-
tor 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 lan-
guage. Providing specialized, domain-specific tool support and statically checking internal
DSLs is more difficult since these DSLs are really using general-purpose language con-
structs 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, it does not show the execution flow and data structures in a domain-specific fashion. This would require analysis of the source code to determine the “DSL parts” and integration of specialized domain-specific debugging facilities with a general-purpose debugger, which is not trivial. 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.

4.1.4 Conclusion

In theory, creating a debugger for a DSL does not differ much from creating a debugger for a traditional general purpose language. Just as with DSL programs, 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 should 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.

4.2 Reflection on the Debugger Design

In this subsection we reflect on our previous debugger design and determine how this will fit into a more generalized approach. We will evaluate if our assumptions regarding the use of debug events, the instrumentation tool in the toolchain and the debugger runtime infrastructure are still valid for a DSL-independent debugger framework.

4.2.1 Debug Events

Although we have identified four common debug events (enter, exit, step and var), we should not limit our debugger design to these events. It may very well be possible that some DSL will require other debug events. For example, in Java it is also possible to listen to an exception event, this event will fire when an exception is thrown. We should consider supporting a generic debug event implementation, that is customizable per DSL debugger instance.
4.2. Reflection on the Debugger Design

4.2.2 Instrumentation in the Toolchain

We have limited our research to DSLs that are implemented with the compiler or the interpreter approach. These implementation approaches distinguish themselves by their explicit hand-written toolchain. Again, we cannot make to much assumptions about the extendability of a toolchain. By implementing the debug events as function calls in the DSL program itself we can instrument a DSL program before the normal toolchain is invoked.

However, this approach requires that the functions that implement a debug event are easily identifiable in the target language. The best way to ensure this is to define those functions in a separate debug library so the generated source code stays the same for every DSL program.

The instrumentation tool for Stratego used a set of predefined rules to add debug events to certain syntax constructs. Such a rule basically answers three questions:

1. At which syntax construct should a debug event be added?
2. How does it extract the relevant debug information?
3. How should the debug event be inserted?

The rule depends on the ATerm structure of a DSL program which in turn depends on its grammar definition. It is impossible to set up a fixed set of rules that hold for each DSL, therefore it makes sense to construct a high-level language that operates on a language definition to answer the three questions above. The instrumentation of DSL programs is then driven by a specification written in that language.

4.2.3 Debugger Runtime Infrastructure

The debugger runtime receives the debug events and updates the program state accordingly. Again, we can hardcode how the debugger will alter the state for the common debug events, but if we support custom debug events the debugger has to be extendable with new debug event handlers.

In the previous paragraph we have mentioned the use of debug libraries that will make the debug event identifiable in the target language. This approach will also allow some degree of reuse between DSLs that have the same target language. For example, two DSLs that use Java as their target language could both import the same debug library.

In our examples we have only covered DSLs that use Java as their target language. This makes it easy to integrate them into the Eclipse IDE because we can use the Java debugger. But how will we be able to support DSLs that do not use Java but another language such as Python or JavaScript as their target language? Fortunately, some general purpose languages have a debugger with a Java interface. For example, Python and JavaScript debugging support is available in Eclipse via the PyDev\(^1\) and ChromeDevTools\(^2\) plugins. It is not guaranteed that each target language debugger API returns the same set of debug events.

---

\(^1\) [http://pydev.org](http://pydev.org)
\(^2\) [https://code.google.com/p/chromedevtools](https://code.google.com/p/chromedevtools)
information in the same format. An explicit conversion is needed that converts the debug information to a generic format.

4.2.4 The Next Step

In the next chapter we propose a generic debugger generation framework that uses a high-level language to define a debug instrumentation component. The debug events will be handled by a language-independent debugger runtime infrastructure with reusable components. The debugger framework does not target all possible DSLs, instead we have chosen to focus on external DSLs which are mainly imperative.
Chapter 5

Language-Independent Debugger Implementation Framework

We propose a language-independent debugger implementation framework that abstracts over the issues raised in Section 4.1 regarding the executability, the programming paradigm and the implementation approach of a DSL. The design of the frames takes the suggestions into account that were proposed in Section 4.2 to adapt the Stratego-specific debugger design from Chapter 3 to a language-independent approach. The framework consists of three components shown in Figure 5.1:

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.

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 debug information. Augmented DSL programs interoperate with the generic debugger infrastructure, which provides the glue to connect an instrumented DSL program to the IDE-integrated debugger that uses the debug events to update its representation of the program state of the debuggee.

Section 5.1 describes the event-based debugging model and the debugger specification language (1) that describes the relation between the DSL program and debug events. Section 5.2 discusses the implementation of the debug information extraction and event generation transformations used by the instrumentation tool (2). And Section 5.3 presents the language-independent debugger runtime infrastructure (3). The infrastructure consists of the implementation of the host-language dependent event-sending mechanism as well as the language-independent debugger.
5. LANGUAGE-INDEPENDENT DEBUGGER IMPLEMENTATION FRAMEWORK

Figure 5.1: Architectural overview of the language-independent debugger implementation framework.

5.1 Declarative Debugger Specification

The declarative debugger specification defines the DSL debugging model. Each declaration defines how a DSL debug event matches to a specific DSL syntax construct. The specification also defines how debug information is extracted from the DSL syntax construct and how a debug event is inserted in the DSL program. The metadata extraction and debug event insertion is implemented using Stratego and will be discussed in Section 5.2.

5.1.1 Debug Event Classes

We use a set of four language-independent debug event classes to capture the runtime program behavior, following Auguston [1] and our conclusion from Section 4.2. 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 existing code generator component. Still, writing the specification requires syntactic and semantic knowledge about the DSL because the language-independent debug events must be linked to syntax constructs in the DSL program that have a semantic meaning corresponding to the meaning of the debug event.
5.1.2 Syntax Event Linking (SEL)

While the debug events we use are the same across various DSLs, the 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 (SEL) language describes this relation.

The SEL specification language describes a) where to inject a debug event, b) transformation strategies that define how to generate debug events in DSL syntax and c) how to extract relevant debug information such as line number, filename and current method name from the DSL program. An 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), that correspond to one of the four debug events (eventClass). The injection is done using transformations that are specified in the Stratego transformation language, 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 an example and a detailed explanation on the basic SEL syntax. We discuss the generation and extraction transformations in Section 5.2.

5.1.3 Syntax Pattern Matching

To find specific AST nodes in a parsed program a syntax construct pattern 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 syntax definition. 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 constructors generated by the Statement sort and _.FunctionDecl matches against all FunctionDecl constructors regardless of the sort.

5.1.4 An Example SEL Specification

In Figure 5.2 a small DSL program with functions and statements is shown on the left, together with its abstract syntax representation on the right. Note that we show abstract syntax trees using a textual representation, using prefix constructor terms for tree nodes and indicating list tree nodes with square brackets. In the abstract syntax the tree nodes that are
5. Language-Independent Debugger Implementation Framework

Example program

```plaintext
module example
function main() {
  bar := 5 + 6;
  baz := baz(bar);
  result := baz + bar;
  print(result);
}
function baz(value) {
  bar := value + 22;
  return bar;
}
```

Program in abstract syntax

```plaintext
Module{
  "example"
  [ FunctionDecl{
      "main"
    , []
    , [ VarAssign("bar", Add(Integer("5"), Integer("6")))]
    , [ VarAssign("baz", FuncCall("baz", [VarUse("bar")]))]
    , [ VarAssign("result", AddVarUse("baz", VarUse("bar")))]
    , [ FuncCall("print", [VarUse("result")])]}
}
FunctionDecl{
  "baz"
  [Argdef("value")]
  [VarAssign("bar", AddVarUse("value", Integer("22")))]
  [Return(VarUse("bar"))]
}
```

Figure 5.2: An example DSL program in concrete and abstract syntax.

SDF grammar example

```plaintext
ID "(" {Expr ","}* ")" ";"-> Statement {cons("FuncCall")}
ID ":=" Expr ";"-> Statement {cons("VarAssign")}
"return" Expr ";"-> Statement {cons("Return")}

"function" ID "(" (ArgDef ",")* ")" ";"-> "{ Statement* }" -> Def {cons("FunctionDecl")}
```

SEL example

```plaintext
instrumentation
  event enter at FunctionDecl.
  creates gen-func-enter
  from extract-function-debug-info
  event step at Statement.
  creates gen-stat-step
  from extract-statement-debug-info
```

Figure 5.3: 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).

important for the debugger are surrounded by a rectangle, function declarations are depicted as a rectangle with a dashed line and statements as a rectangle with a solid line.

Figure 5.3 shows SDF syntax definition rules and a SEL specification that map the functions and statements of Figure 5.3 to events. The dotted line shows how an enter event is matched against the FunctionDecl constructor using a syntax construct pattern with an explicit constructor. The solid line shows how a step 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.
5.2 DSL Program Instrumentation

A debugger requires that debug information about the DSL program is available at runtime. In this section we present a reusable, language-parametric debug instrumentation tool, which uses the SEL specification to incorporate the necessary debug information.

5.2.1 Toolchain

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 debug 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 5.4 shows the toolchain 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. The parser produces an abstract syntax tree, which is used as the input for the debug instrumentation tool. The debug instrumentation tool augments the abstract syntax tree 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.

![Figure 5.4: The toolchain for instrumentation and execution of DSL programs.](image)

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.

5.2.2 Instrumentation by Preprocessing

The instrumentation tool traverses the abstract representation of the DSL program in a bottom-up fashion. If the signature of a syntax construct matches a pattern in the SEL specification then the extraction transformation is applied to this syntax construct to extract debug information. The extractor returns a tuple containing debug information required for the debug event, such as the line number or filename. The generator transformation then encodes the debug information in a debug statement and inserts it in the DSL program.
5. LANGUAGE-INDEPENDENT DEBUGGER IMPLEMENTATION FRAMEWORK

```
return bar();
```

Figure 5.5: Exit event insertion using return expression extraction.

```
def int foo() {
    ...
    if (...) {
        return ...
    } else if (...) {
        return ...
    } else {
        return ...
    }
}
```

Figure 5.6: Exit event insertion using a try-finally block.

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 does not ensure the correct ordering of events. For example, when the return statement contains a method call, the events fired by that method should of course occur before the exit event. The implementation of the generator transformation that defines how events are inserted has to take care of this problem.

Figure 5.5 shows how a return statement can be transformed into a fragment in which the expression of the return statement is evaluated before the exit event is fired. DSLs that support try-finally blocks\(^1\) can overcome the multiple-exit-points issue with less instrumentation effort. When the exit event is placed in the finally-block it is always executed after the body of the try-block is executed. Figure 5.6 shows how a method with multiple return statements is transformed.

---

\(^1\) Try-finally blocks, or try-catch-finally blocks are common in languages that supports exceptions. When an exception is thrown in a try-block it is caught in the catch-block. The finally-block is always executed after the try-block or catch-block completes, even when they contain a return statement.
5.2.3 Example: Extractor and Generator Implementation

In this section we will cover the extractor and generator transformations of Figure 5.3 in more detail. Both the extractor and generator transformations are implemented in Stratego and operate on the matched syntax construct. The extractor transformation returns a tuple containing the debug information and the generator transformation returns a replacement of the matched syntax construct that contains the debug event.

For example, consider the following SEL definition:

```sel
event step at Statement._
    creates gen-stat-step
    from extract-statement-debug-info
```

The pattern in this definition matches every ATerm that is a `Statement` sort. In Figure 5.2 these ATerms are `VarAssign`, `FuncCall` and `Return`. The debug information required for a step event is the filename and the location information of the statement. The extractor transformation `extract-statement-debug-info` needs to return this information. Although the location info is not shown in the example, it is stored as an annotation of the ATerm and can be retrieved using the strategy `origin-location`. The result is the tuple `(a, b, c, d)` where `a` is the start line number, `b` the offset on that line, `c` the end line number and `d` the end offset. The name of the current file can be retrieved via a call to the dynamic rule `DSL-PROGRAM-FILENAME`, which is set when a DSL file is parsed. The extractor transformation for the step event, `extract-statement-debug-info`, is implemented as follows:

```stratego
extract-statement-debug-info:
    s -> debug-information
    where
        (a, b, c, d) := <origin-location> s
        ; filename := <DSL-PROGRAM-FILENAME>
        ; debug-information := (filename, a, b, c, d)
```

After the debug-information is extracted, the step event is inserted. The generator transformation `gen-stat-step` is implemented as follows:

```stratego
gen-stat-step:
    s -> instrumented-s
    where
        debug-information := <GET-CURRENT-DEBUG-INFORMATION>
        // convert the debug-info to a DSL specific data structure
        ; args* := <convert-to-arguments> debug-information
        ; event-call := !FuncCall("step-event", args*)
        ; ![event-call, s]
```

Note that `VarAssign` could also fire a var event, but we only consider the step event in this example.

The `origin-location` strategy is part of the origin tracking library in Spoofax.
Instrumenting code at the DSL-level gives us the advantage that a DSL program instrumented with debug events can run on any back-end 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.

5.3 Debugger Runtime Infrastructure

In this section we discuss our debugger runtime 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 5.7.

By splitting 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 debug library has to be implemented (1). If a new platform is targeted, e.g. JavaScript, then the platform-level debug library (2) and platform-level event handler (3) have to be implemented. The IDE-level debug library component (4) is neither language nor platform specific and is fixed for each debugger implementation unless integration with a new IDE is required. Figure 5.8 shows to which extent components can be reused for different DSLs. WebDSL and Stratego share the Java debug library and the Java event handler because they both use Java as their target language. Mobl on the other hand targets JavaScript and therefore requires its own debug library and event handler.
5.3. Debugger Runtime Infrastructure

5.3.1 DSL-Level Debug Library

The debug events added by the instrumentation tool described in Section 5.2.2 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 gap between DSL code and platform-level code can be bridged when the DSL supports defining functions in native code. This allows the DSL-level debug library to be implemented in the target language, from which it is trivial to call the platform-level debug library. Besides the event forwarding the DSL-level debug library also has to convert DSL-specific data structures that contain the DSL-specific runtime metadata to a data structure that is compatible with the target language.

5.3.2 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. The implementation of the marshalling is discussed in the next subsection.
5. LANGUAGE-INDEPENDENT DEBUGGER IMPLEMENTATION FRAMEWORK

5.3.3 Platform-Level Event Handler

The platform-level event handler receives the debug events and is responsible for controlling the execution of the DSL program. The received debug events from the platform-level debug library are passed on to the IDE-level debug library to update the program state. The DSL program is controlled by suspending and resuming the underlying runtime system of the target language which is determined by 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 the Java Debug Interface (JDI) API [39] to control breakpoints and reflect over the execution state of a running Java application. The JDI API can be used by the platform-level event handler to control the runtime of the running DSL program. The difference between this approach and the approach discussed in Section 3.3.2 and Figure 3.5 is that the event sending part is implemented in a language-independent manner.

Marshalling the event from the platform-level debug library to the platform-level event handler is implemented as follows for the Java platform: Before the DSL program starts the Java event handler sets a breakpoint in the Java debug library of the DSL. The breakpoint is placed so that the debug event metadata is in scope and can be accessed via reflection. When that breakpoint is hit the Java event handler extracts the debug event metadata and sends it to the IDE-level debug library which updates the program state. The IDE-level debug library determines if the DSL program should stay suspend because a DSL breakpoint was hit, otherwise the execution resumes.

Note that the implementation platform does not require a Java-component that exposes the platform-level debugger API. For example, the event marshalling can also be achieved by communicating with an external debugger process via sockets or remote procedure calls. The IDE-level debug library is unaware of the communication scheme between the debuggee and the platform-level event handler. It only needs to respond to debug events.

5.3.4 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 uses language-independent data structures that conform to the Eclipse debug model to implement the debugger. By reusing the Eclipse debug model the user interface automatically contains our debug information.

The debug view contains the stack of the running program, the variables view shows the variables that are in scope of the selected stack frame and the breakpoints view shows the list of DSL breakpoints.

Each of the four debug event classes is processed and updates the program state. The step event is used to change the 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.

5.3.5 DSL Program Execution Flow

Figure 5.9 shows the control flow between the runtime components when a debug event statement is executed. The IDE execution context is contained within the Eclipse JVM and the DSL program execution context is contained within its own process.

1. (a) The DSL program executes a debug event sending statement and (b) forwards it to the platform-level debug library.

2. A breakpoint is hit in the host language, DSL program 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. The program state is then compared to the DSL breakpoints set by the user to determine if a breakpoint is hit.

At this point two execution paths are possible:

5. (a) 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).
   (b) If no DSL breakpoint is hit, the DSL program execution is resumed.

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 actions are implemented as discussed in Section 3.2.3

5.3.6 Eclipse Integration

To achieve a functional DSL debugger the components in the framework need to be joined together in an Eclipse plugin. This requires a certain amount of boilerplate code as well as a DSL-specific extension that defines how to launch the debugger for a given DSL program.
As indicated before, we implemented our framework as part of the Spoofax language workbench. Spoofax uses several high-level languages that operate on a language definition to define editor services for the DSL in question. A debugger that can be defined using the same paradigm would be a welcome addition to the set of DSL support tools that can be defined with Spoofax.

To validate the implementation of our language-independent debugger framework 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. Both languages have Eclipse editor support which is implemented in Spoofax, this gives us the advantage that we can reuse the existing language definition for these languages. For each case study we will cover the SEL specification and the runtime debug components that are needed to build a fully working debugger.

### 6.1 Stratego

As discussed in Section 2.4 Stratego is a transformation 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.

Before we can implement a debugger we have to determine which debug event maps to which syntactic construct of the language definition. After examining the Stratego grammar and several Stratego programs and their ATerm representation we were able to deduce which syntactic constructs correspond to a debug event. The `Strategy` sort, which is the equivalent of a statement, captures the syntactic constructs that map to a step event. The `enter` and `exit` event map to syntactic constructs that are identified by the `RuleDef` or `StrategyDef` sort. The `var` event maps to syntactic constructs that occur in the `Strategy`, `RuleDef` or `StrategyDef` sort.

Figure 6.1: Syntax productions and events in Stratego.

Figure 6.1 shows an overview of the number of grammar rules and their events. The table tells us that there are 58 grammar rules that resolve to the Strategy sort, thus we have a maximum of 58 different Strategy constructors. The var event is matched to multiple sorts, because variables can be defined within strategy expressions and the signature of rule and strategy definitions. The var event does not have to be matched against all constructors in the Strategy sort, because not every constructor can introduce variables.

Each debug event type requires a specific set of DSL program metadata to update the program state with meaningful information. To retrieve the location information, which is mandatory for every event, Stratego programs are parsed to an abstract syntax tree that includes source code location annotations. This location information has to be extracted by the extraction transformations. Furthermore, 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.

6.1.1 Instrumentation using SEL

Using the SEL language we can define the debug instrumentation preprocessor that we will use in the Stratego toolchain. First, we have to specify which Stratego syntax constructs correspond to which debug events. Second, we have to specify how we can extract the debug information from the abstract syntax tree using the extractor transformation. And finally, we have to specify how the debug events are implemented in the Stratego code using the generator transformation. Also, take note that while we are creating a debugger for Stratego both the extractor and generator transformations are also implemented in Stratego.

6.1.1.1 Construct Pattern

Figure 6.2 shows a fragment of the SEL specification for the Stratego language. The StrategyDef sort is a syntax construct that corresponds to a strategy definition, when such a definition is executed an enter event should be fired and an exit event should fire 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 strategy signatures. Therefore, the var event has to be matched against the Strategy sort and the StrategyDef sort.
6.1. Stratego

```
event enter at StrategyDef._
    creates gen-strategy-enter
    from extract-strategy-info

event exit at StrategyDef._
    creates gen-strategy-exit
    from extract-strategy-info

event step at Strategy._
    creates gen-step
    from extract-step-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
```

Figure 6.2: A part of the SEL specification for Stratego.

6.1.1.2 Extractor and Generator

The `bottomup` strategy definition will be used as an example in this subsection. It is part of the Stratego Library and is implemented as follows:

```
bottomup(s) = all(bottomup(s)); s
```

which corresponds to the following ATerm representation:

```
SDef(  
"bottomup",  
[DefaultVarDec("s")]
, Seq(  
   All(Call(SVar("bottomup"), [CallNoArgs(SVar("s"))])),  
   CallNoArgs(SVar("s"))
 )
)
```

The `SDef` constructor is defined as follows in the Stratego grammar:

```
Id "(" {Typedid ","}* ")" "=" Strategy  
-> StrategyDef {cons("SDef")}
```

The pattern `StrategyDef._` in the SEL specification in Figure 6.2 therefore matches against the entire ATerm fragment and is used as input for the `extract-strategy-info` strategy shown in Figure 6.3. The `filename` is retrieved via the `DSL-PROGRAM-FILENAME` strategy at line 5. The `strategy name` is extracted at line 6 and the `location information` is extracted at line 7. The debug-information is then returned as a tuple containing the metadata.

After the debug information is extracted the debug event has to be inserted. Figure 6.4 shows the implementation for the `enter` event. The `gen-strategy-enter` strategy rewrites a strategy definition ATerm to include an `enter` event. First the `enter` event strategy call is
6. Case Studies

```
// f is the term that matched the SEL syntax pattern

extract-strategy-info:
  f -> debug-information
where
  filename := <GET-DSL-PROGRAM-FIILENAME>
  f-name := <get-strategy-name> f
  (a, b, c, d) := <origin-location> f
  debug-information := (filename, a, b, c, d, f-name)

get-strategy-name:
  SDefNoArgs(s-name, s) -> s-name
get-strategy-name:
  SDef(s-name, s-args, s) -> s-name
get-strategy-name:
  SDefT(s-name, s-args, t-args, s) -> s-name

Figure 6.3: Extractor transformation for an enter and enter event.
```

```
// current term is a SDefNoArgs, SDef or SDefT

gen-strategy-enter:
  strategy-def -> instrumented-strategy-def
where
  debug-information := <GET-CURRENT-DEBUG-INFORMATION>
  enter-event := <create-s-enter-call> debug-information
  instrumented-strategy-def :=
    <insert-enter-event(|enter-event)> strategy-def

insert-enter-event(|enter-event):
  SDefNoArgs(s-name, s)
    -> SDefNoArgs(s-name, Seq(enter-event, s))
insert-enter-event(|enter-event):
  SDef(s-name, s-args, s)
    -> SDef(s-name, s-args, Seq(enter-event, s))
insert-enter-event(|enter-event):
  SDefT(s-name, s-args, t-args, s)
    -> SDefT(s-name, s-args, t-args, Seq(enter-event, s))

create-s-enter-call =
  ?[filename, a, b, c, d, f-name]
  ; program-filename := $["{filename}"
  ; location := $["{a}, {b}, {c}, {d}" ]
  ; functionName := $["{f-name}" ]
  ; args* := <map(create-str-arg)> [filename, name, location]
  ; !CallT(SVar("s-enter"), [ ], args*)

Figure 6.4: Generation transformation for the enter event.
```
6.1. Stratego

```plaintext
// current term is a SDefNoArgs, SDef or SDefT
gen-strategy-exit:
strategy-def -> instrumented-strategy-def
where
  debug-information := <GET-CURRENT-DEBUG-INFORMATION>
  ; exit-event := <create-s-exit-call> debug-information
  ; instrumented-strategy-def :=
    <insert-exit-event(|exit-event)> strategy-def
insert-exit-event(|exit-event):
  SDefNoArgs(s-name, s)
    -> SDefNoArgs(s-name, <wrap-s-exit(|exit-event)> s)
insert-exit-event(|exit-event):
  SDef(s-name, s-args, s)
    -> SDef(s-name, s-args, <wrap-s-exit(|exit-event)> s)
insert-exit-event(|exit-event):
  SDefT(s-name, s-args, t-args, s)
    -> SDefT(s-name, s-args, t-args,
               <wrap-s-exit(|exit-event)> s)
// The output will be: s-body < s-exit + (s-exit; fail)
wrap-s-exit(|exit-event):
  s-body -> body-with-exit
  with
    f := <create-fail>
    ; s-exit-fail := Seq(exit-event, f)
    ; body-with-exit := GuardedLChoice(s-body, exit-event,
                                         s-exit-fail)
create-s-exit-call =
  ?(filename, a, b, c, d, f-name)
  ; program-filename := $["[filename]"
  ; location := $["[a],[b],[c],[d]"
  ; functionName := $["[f-name]"
  ; args* := <map(create-str-arg)> [filename, name, location]
  ; !CallT(SVar("s-exit"), [], args*)
```

Figure 6.5: Generation transformation for the exit event.

generated in the create-s-enter-call strategy then the call is inserted in the strategy body in the insert-enter-event strategy.

Figure 6.5 shows the implementation for the exit event. The injection of the exit event is not as trivial as the enter event as discussed in Section 3.3.1. The body of the strategy definition has to be wrapped in the Stratego version of a try-finally-block to catch possible failures in the body that would cause a premature exit. The gen-strategy-exit strategy calls create-s-exit-call to create the exit event call, then the call is inserted in the strategy body by the insert-exit-event strategy. insert-exit-event creates a GuardedLChoice ATerm, the ATerm representation of the guarded left choice operator, which contains three subterms. The first subterm (s-body) is the body of the original strategy definition. When the execution of this statement does not fail, the second subterm is executed which fires an exit event. However, when the statement does fail at some point during execution, the third subterm is executed which fires an exit event followed by a fail.
6. Case Studies

module debug-library

strategies

   enter(debug-info) = ?current-term
   // call the native enter library
   ; java-enter(debug-info, current-term)

Figure 6.6: Stratego debug library fragment.

class java_enter_0_5 extends org.strategoxt.lang.Strategy {
   // method arguments are debug-information
   public IStrategoTerm invoke(Context c, IStrategoTerm current, debug_information_arguments) {
      // Encode debug-info in a single String
      // to limit the inter-VM data transfer
      String debugInfo = extractDebugInfo(debug_information_arguments);
      PlatformLevelDebugLibrary.enter(debugInfo)
      return current;
   }
}

Figure 6.7: Native implementation of Stratego debug library.

6.1.2 Debugger Runtime

The debug event calls are defined in a Stratego library which serves as an API between the Stratego code and the Java platform-level debug library. Every debug event call strategy is implemented in a native Java class which delegates the event information to the Java platform-level debug library.

Every event has a corresponding Java class with an invoke method, the body of this method is executed when the strategy is called. The arguments of the invoke method contain the required debug information for the event. The invoke method converts the debug information from the arguments to a string. The string is then passed to the corresponding method in the platform-level debug library.

Figure 6.6 shows the implementation of the enter event in the Stratego debug library. The enter strategy calls the java-enter strategy which is implemented in a native Java class as shown in Figure 6.7. When java-enter is called it will execute the code in the invoke method of the java_enter_0_5 class. The invoke method will first convert the debug information from the arguments to a string representation using the extractDebugInfo method, then the enter event method in the platform-level debug library is called. Figure 6.8 shows the implementation of this method. The method only contains a call to the nothing method, which does exactly what it says it does: namely nothing. This call to an empty method is needed to have a valid position in the source code at which the debugger can place a breakpoint as discussed in Section 5.3.3.

Now when a debug instrumented Stratego program is started via the debugger, the debugger will place breakpoints in the java classes at the nothing(); statement. During the
6.2. WebDSL

WebDSL is a DSL for web applications with a rich data model [17] and contains multiple sub-languages to handle various concerns such as data validation [19], user interface definitions [52], and access control [18]. WebDSL is used as a subject in this 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 mainly focus on defining a debugger for the declarative part of WebDSL.

6.2.1 WebDSL Debug Events

WebDSL uses a declarative language for page definitions to build the user interface of a web page. 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 supporting context-sensitive customizable user interface components.

During the evaluation of a page definition, the web page is outputted incrementally as HTML code to a stream that will be the response of a web page request. While it would be useful to inspect the generated HTML at any point during the response creation we would rather want to view the user interface components at the DSL-level.

The HTML Document Object Model (DOM) structure of a web page and the WebDSL view of a web page are both defined as a tree structure. We can model the creation of that tree in terms of events that correspond to the template and page calls in WebDSL. An **enter** event is fired when a template is called, an **exit** event is fired when a template call exits a
6. CASE STUDIES

Figure 6.9: A WebDSL fragment with an event trace and the generated HTML.

step event is fired for each statement in the template body and a var event is fired when a variable is introduced. Not only do the debug events serve as actions that change the runtime state, a trace of debug events also shows how a web page is build.

Figure 6.9 shows a WebDSL fragment for the page definition foo on the left side, its event trace in the center and the generated web page on the right side. The definitions foo, viewEvent, commonHeader, mainBody and footer have corresponding enter and exit events in the event trace. The arrows show how a event corresponds to a source file location and a HTML element.

6.2.2 Instrumentation using SEL

The SEL definition for WebDSL is larger than that of Stratego because we need to cover the imperative part of the language, also called Action Code, and the declarative user interface templating language. First we discuss the syntax construct patterns that match the debug events, then we discuss the issues with the extractor and generator transformations.

6.2.2.1 Construct Pattern

Figure 6.10 shows a fragment of the SEL specification for the WebDSL language. The TemplateDefinition sort is the syntax construct to which all page and template definitions
6.2. WebDSL

**instrumentation**

<table>
<thead>
<tr>
<th>Event class</th>
<th>Sort Category</th>
<th>Production Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>step</td>
<td>Statement</td>
<td>26</td>
</tr>
<tr>
<td>step</td>
<td>TemplateElement</td>
<td>61</td>
</tr>
<tr>
<td>enter/exit</td>
<td>Function</td>
<td>1</td>
</tr>
<tr>
<td>enter/exit</td>
<td>TemplateDefinition</td>
<td>4</td>
</tr>
</tbody>
</table>

belong. Whenever such a definition is evaluated an enter event should fire and an exit should fire at the end of the definition. The TemplateElement sort is associated with a step event. For the imperative parts the Statement sort corresponds to a step event and a Function sort has an enter and exit event. Figure 6.11 shows an overview of the number of grammar rules and their events.

### 6.2.2.2 Extractor and Generator

The implementation of the extractor transformations for the WebDSL program debug instrumentation is straightforward, just as with the Stratego case study. The relevant debug information, such as the template name or function name, is contained within the AST element that is matched by the syntax pattern.

By definition (see Section 5.2) the generator transformation takes a source code fragment as input and outputs a debug instrumented version of that fragment. Unlike Stratego, WebDSL does not have a statement-composition operator or a try-finally-block language construct. The statement-composition operator is needed for generation of step events.
Consider the following statement:

```plaintext
var f := foo()
```

Without a statement-composition operator we could try to output an instrumented statement as a statement block:

```plaintext
{  
    step_event()  
    var f := foo()  
}
```

However, this will not give us the desired result because the statement block introduces a new scope and hides the variable \( f \) from the outer scope.

Extending the language with a statement-composition operator is a viable option but the extra operator has no advantages in the language. Therefore, we decided to add an optional post-instrumentation step in the debug instrumentation library. Figure 6.12 shows the implementation of the `gen-step` strategy that returns a `DebugStep ATerm` that wraps the debug statement and the corresponding WebDSL statement in a single ATerm. Figure 6.13 shows the implementation of a post-instrumentation strategy that unwraps the `DebugStep ATerm` into two separate terms in a list. For example, the list

\[[Stat1, Stat2, DebugStep(Stat3, Stat4), Stat5]\]

is rewritten as

\[[Stat1, Stat2, Stat3, Stat4, Stat5]\]

The `replace-debug-aterms` strategy is exhaustively applied on the entire ATerm representation of a WebDSL source file until all `DebugStep ATerms` are replaced.
To make the implementation of the generator transformation for the exit event of a function less complex we decided to extend the WebDSL language with a try-finally-block. The body of the function is put in the try-block and the exit event is put in the finally-block to ensure that it will always fire a function exit event. The try-finally block language addition required 2 extra rules in SDF, 4 new pretty printing rules and 30 lines of Stratego code in the Java servlet code generation component. We view that the minor modification and the fact that WebDSL already supports try-catch blocks justifies the compiler extension.

### 6.2.3 Debugger Runtime

The WebDSL back-end is implemented in Java using Java Servlet technology. This means we can reuse both the platform-specific event handler and the platform-specific debug library we used for the Stratego debugger. 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.

The methods in the platform-specific debug library can be exposed to a WebDSL application by wrapping them in a native Java class and declaring the class signature in WebDSL code as shown in Figure 6.14. The static methods in the native class can be called directly within functions. However, function calls cannot be placed directly inside a template definition, unless they are used to initialize a variable. Therefore, the API contains template definitions, as shown in Figure 6.15 wrapped around the static methods which can be called anywhere inside a template definition. Figure 6.16 shows that the implementation of the step event in the native Java class nativejava.WebDSLDebugEvents is just a thin wrapper around the PlatformLevelDebugLibrary. The other events are implemented in a similar way.

Figure 6.17 shows the debug instrumented version of the `viewEvent` template definition from Figure 6.9. The debug events are underlined and the arguments containing the debug information have been omitted for brevity.
module webdsldebuglib/debugevents

define debug_template_enter(filename : String, 
    location : String, 
    templateName : String) {
    var s := DebugEvents 
        .templateEnter(filename, location, templateName);
}

define debug_template_exit(filename : String, 
    location : String, 
    templateName : String) {
    var s := DebugEvents 
        .templateExit(filename, location, templateName);
}

define debug_template_step(filename : String, 
    location : String) {
    var s := DebugEvents 
        .statementStep(filename, location)
}

define debug_template_var(filename : String, 
    location : String, 
    varname : String) {
    var s := DebugEvents 
        .var(filename, location, varname)
}

package nativejava;
public class WebDSLDebugEvents {
    public static String statementStep(String filename, String location) {
        String debugInfo = filename + "\t" + location;
        PlatformLevelDebugLibrary.step(debugInfo);
        return "";
    }
}

define viewEvent(e:Event) {
    debug_template_enter(...) 
    debug_template_var(...) 
    debug_template_step(...) 
    output(e.Name) 
    debug_template_step(...) 
    viewParticipants(e:Event) 
    debug_template_exit(...) 
}

Figure 6.15: WebDSL runtime debug library for templates.

Figure 6.16: A fragment of the Java back-end debug runtime library for WebDSL.

Figure 6.17: Debug instrumented WebDSL fragment.
6.2.4 Eclipse Integration

Figure [6.18] shows a screenshot of the WebDSL debugger integrated in Eclipse connected to a running YellowGrass application. The application is suspend at line 85 of `tag.app` in the `showTag` template.

Figure 6.18: A screenshot of the WebDSL debugger debugging a YellowGrass application.
Chapter 7

Evaluation Criteria Results

In this chapter we reflect over our work with respect to the evaluation criteria defined in Figure 1.2.

7.1 Usability of the Generated Debuggers

In this subsection we discuss the evaluation criteria that are important for the users of a DSL debugger.

**EV1: How complete are the generated debuggers compared to a handwritten debugger?**

Using a generic framework we were able to implement a Stratego and WebDSL debugger which fulfills the requirements from Figure 1.1. Advanced features such as conditional breakpoints and JVM-like hot code replacement\(^1\) found in debuggers for GPLs are not supported because they depend on the DSL implementation. For example, to support conditional breakpoints the debugger needs to evaluate a condition expressed at the DSL-level in the context of the current stack frame. This requires a parser that can parse DSL expressions and a runtime that supports the evaluation of such expressions within a dynamic context.

When we compare the handwritten Stratego debugger with its generated counterpart we note that the handwritten debugger is also capable of showing the names of the available dynamic rules. Extracting such language-specific context-sensitive information has to be implemented manually. For this we need to extend the debugger runtime library with a user-definable data extraction service that is able to retrieve additional debug related information from the program. In the Stratego case the Stratego debugger runtime would expose an API that returns the names of the active dynamic rules. The debugger would call the API during its program state updating step.

\(^1\)Also known as “Edit and Continue” in the Visual Studio IDE.
EV2: How easy is it for the DSL application programmer to activate the debugger?

The Eclipse integration makes it easy to launch a debugger for a given DSL program. The debugger can either attach to an already running program (assuming the program is build with debug information) or it can start a program and connect to it after it has been instrumented and compiled. The WebDSL debugger can connect to a running application when the Apache Tomcat instance is configured to allow remote debugging. The Stratego debugger can be launched in different ways depending on the target program: a) A stand-alone Stratego program can be started via Eclipse by right clicking on the main module and selecting “Run As → Stratego Program”. b) It can connect to a already running debug instrumented program. c) When the Stratego code is part of a Spoofax editor service, a new Eclipse instance has to be launched because the executed Stratego code needs to be in a different JVM than the debugger. The latter invocation causes additional memory and CPU usage overhead, but, in general, the compile and runtime overhead of the target program is of a much greater concern as will be discussed in the next subsections.

EV3: What is the debug instrumentation duration overhead?

The debug instrumentation per file is split in three phases:

1. The parsing phase: The textual source file is parsed to a term representation.
2. The instrumentation phase: The debug events are added to the term.
3. The writing phase: The instrumented term is written back to disk.

The implementation of the debug instrumentation tool suggests that the duration of the instrumentation phase is linear with respect to the size of the input source file, but we do not know how the duration of the different phases relate to each other. Therefore, we will instrument several Stratego and WebDSL applications and measure the duration of each phase.

Stratego

For our first measurement we target the StrategoXT library\(^2\). The entire StrategoXT library contains about 1000 Stratego files, from which only 500 files contribute to the final distributable jar, the other 500 files are scripts used in test cases. We ignore the test cases because they are not part of the build process.

First, we established a baseline that only measures the duration of parsing and writing non-instrumented Stratego files to get an indication of the parse and write duration.Parsing the 500 files took 8.7 seconds and writing them back to disk took 9.2 seconds. Next, we ran the instrumentation tool again but this time with debug instrumentation enabled. The

\(^2\) The source is located at [https://github.com/metaborg/strategoxt/tree/java-bootstrap](https://github.com/metaborg/strategoxt/tree/java-bootstrap). We target the java-bootstrap branch because this build uses Apache Ant instead of GNU Make which makes it easier to integrate the instrumentation tool into the build process.
7.1. Usability of the Generated Debuggers

parsing time was the same, the instrumentation phase took 9.6 seconds and writing those instrumented files to disk took 37.3 seconds. For completeness we also measured the parsing time of the instrumented files which took 92.2 seconds. For debug instrumented files the parsing time increased with 960% and the writing time increased with 305%.

Although the writing phase takes approximately 67% of the total instrumentation duration, the increased parsing time for the instrumented files negatively affects the duration of the Stratego compiler because it has to parse the instrumented source files. Parsing could be sped up by writing the instrumented files in the pre-parsed rtree-format instead of the normal pretty-printed Stratego syntax. Another solution would be to integrate the instrumentation tool in the Stratego compiler, this eliminates the extra writing step in the instrumentation tool and the parsing step in the Stratego compiler.

Figure 7.1 shows a scatter plot of the total duration (parsing, instrumenting and writing) for each source file in StrategoXT library versus the term-size of the parsed source file together with a trend line. The graph suggests a positive linear correlation between the term-size of a source file and its instrumentation duration.

Figure 7.2 shows the instrumentation overhead of several MetaBorg Stratego-oriented projects\(^3\). The relative duration of the parsing (22% of total time), debug instrumentation (18% of total time) and writing (60% of total time) phase does not deviate very much for the different projects.

\(^3\)The source of the projects can be found at https://github.com/metaborg

Figure 7.1: StrategoXT debug instrumentation duration per file plotted against the term-size of the source file.

![StrategoXT instrumentation duration per source file term-size](image-url)
### 7. Evaluation Criteria Results

<table>
<thead>
<tr>
<th>Project</th>
<th>Parsing</th>
<th>Instrumentation</th>
<th>Writing</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>StrategoXT</td>
<td>8.70s</td>
<td>9.60s</td>
<td>37.30s</td>
<td>55.60s</td>
</tr>
<tr>
<td>EntityLang</td>
<td>0.98s</td>
<td>0.52s</td>
<td>1.79s</td>
<td>3.29s</td>
</tr>
<tr>
<td>Box</td>
<td>1.15s</td>
<td>0.57s</td>
<td>2.41s</td>
<td>4.13s</td>
</tr>
<tr>
<td>Lwc2013</td>
<td>1.34s</td>
<td>1.00s</td>
<td>3.62s</td>
<td>5.96s</td>
</tr>
<tr>
<td>MetaC</td>
<td>2.34s</td>
<td>1.83s</td>
<td>5.62s</td>
<td>9.79s</td>
</tr>
<tr>
<td>Rtg</td>
<td>1.06s</td>
<td>0.65s</td>
<td>2.40s</td>
<td>4.11s</td>
</tr>
<tr>
<td>Runtime-libraries</td>
<td>1.80s</td>
<td>1.94s</td>
<td>5.76s</td>
<td>9.50s</td>
</tr>
<tr>
<td>Sdf</td>
<td>2.63s</td>
<td>1.82s</td>
<td>5.93s</td>
<td>10.38s</td>
</tr>
<tr>
<td>Spoofax-csharp</td>
<td>2.63s</td>
<td>1.82s</td>
<td>5.93s</td>
<td>10.38s</td>
</tr>
<tr>
<td>Spoofax-enfun</td>
<td>2.05s</td>
<td>1.58s</td>
<td>5.10s</td>
<td>8.73s</td>
</tr>
<tr>
<td>Spt</td>
<td>2.15s</td>
<td>1.93s</td>
<td>6.44s</td>
<td>10.52s</td>
</tr>
<tr>
<td>Spx</td>
<td>5.23s</td>
<td>6.76s</td>
<td>17.05s</td>
<td>28.04s</td>
</tr>
<tr>
<td>Stratego</td>
<td>2.58s</td>
<td>2.74s</td>
<td>8.92s</td>
<td>14.24s</td>
</tr>
<tr>
<td><strong>Average percentage of total time</strong></td>
<td><strong>22%</strong></td>
<td><strong>18%</strong></td>
<td><strong>60%</strong></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.2: Stratego debug instrumentation duration per phase and percentage of total duration of various projects.

**WebDSL**

The debug instrumentation overhead for WebDSL applications is comparable to that of Stratego. Figure 7.3 shows the total duration for each source file in the YellowGrass project versus the term-size of the parsed source file. To get an acceptable scale two outliers are removed from the graph. Coincidentally these two points correspond to the first two files that are instrumented. Most likely this is related to the cold start up of the JVM, the HotSpot JIT optimizations haven’t kicked in yet. Nonetheless this graph strengthens our assumption of a linear correlation between the term-size and the debug instrumentation overhead.

Figure 7.4 shows the instrumentation overhead for three WebDSL projects. The relative duration of the parsing phase is 29%, the debug instrumentation phase is 27% and the writing phase is 44%. The relative duration of these phases is roughly similar to the phases during Stratego debug instrumentation.

**EV4: What is the compile-time duration overhead of instrumented code versus non-instrumented code?**

The build steps for a Stratego or WebDSL application are similar. First some DSL-specific preprocessing is done, e.g. copying or generating files. Next, the DSL compiler is invoked to generate Java source code. Then the Java code is compiled and finally distributable artifacts are generated. For both Stratego and WebDSL the Java generation and Java compilation depends on the size of the instrumented code and will therefore introduce an overhead.

---

4The source of these projects can be found at [https://github.com/webdsl/yellowgrass](https://github.com/webdsl/yellowgrass), [https://github.com/webdsl/reposearch](https://github.com/webdsl/reposearch) and [https://github.com/eelcovisser/blog](https://github.com/eelcovisser/blog).
7.1. Usability of the Generated Debuggers

Figure 7.3: YellowGrass debug instrumentation duration per file plotted against the term-size of the source file.

<table>
<thead>
<tr>
<th>Project</th>
<th>Parsing</th>
<th>Instrumentation</th>
<th>Writing</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>YellowGrass</td>
<td>2.86s</td>
<td>3.01s</td>
<td>5.07s</td>
<td>10.94s</td>
</tr>
<tr>
<td>Reposearch</td>
<td>2.99s</td>
<td>2.24s</td>
<td>4.02s</td>
<td>9.25s</td>
</tr>
<tr>
<td>Blog</td>
<td>2.52s</td>
<td>2.50s</td>
<td>3.88s</td>
<td>8.90s</td>
</tr>
</tbody>
</table>

Figure 7.4: WebDSL debug instrumentation duration per phase and percentage of total duration of various projects.

**Stratego**

The StrategoXT java-bootstrapped Ant build consists of four major steps:

- **build-syntax**: Generates the parse-tables from SDF.
- **generate-java**: Generates Java source files from Stratego source files.
- **compile-java**: Compiles the Java source.
- **make-jar**: Makes the distributable strategoxt.jar.

We measured the duration of each build step by performing a build on a fresh source code checkout to eliminate caching. Figure 7.5 shows the duration of a normal build on non-instrumented files and a build with instrumented files, for clarity we have also included...
7. Evaluation Criteria Results

<table>
<thead>
<tr>
<th>Build Step</th>
<th>Normal Code</th>
<th>Instrumented Code</th>
<th>Percentage Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>instrument</td>
<td>54.8s</td>
<td>233.4s</td>
<td>414.0s</td>
</tr>
<tr>
<td>build-syntax</td>
<td>233.4s</td>
<td>233.4s</td>
<td>0%</td>
</tr>
<tr>
<td>generate-java</td>
<td>145.2s</td>
<td>642.0s</td>
<td>342%</td>
</tr>
<tr>
<td>compile-java</td>
<td>23.8s</td>
<td>33.4s</td>
<td>40%</td>
</tr>
<tr>
<td>make-jar</td>
<td>11.6s</td>
<td>13.8s</td>
<td>19%</td>
</tr>
<tr>
<td>Total</td>
<td>414.0s</td>
<td>977.4s</td>
<td>136%</td>
</tr>
</tbody>
</table>

Figure 7.5: StrategoXT build duration for normal and instrumented code.

![StrategoXT build steps duration](image)

Figure 7.6: Comparison of StrategoXT build steps duration for normal and instrumented source code.

The duration of the instrumentation. Figure 7.6 shows a stacked bar plot of the same data to illustrate the relative duration. The duration of the generate-java step increased significantly with 342% (from 145.2 seconds to 642.0 seconds) while the compile-java step increased with 40% (from 23.8 seconds to 33.4 seconds) and make-jar with 19% (from 11.6 seconds to 13.8 seconds). The build-syntax step did not suffer from a performance loss because it doesn’t depend on any instrumented Stratego files.

Figure 7.7 shows a comparison of the compilation duration for normal non-instrumented code and instrumented code for several Stratego projects. For each project we have calculated the total term-size and the percentage increment of the compilation duration. The results show that the total term-size of a project does not give an indication of the compile time overhead for instrumented files.

62
7.1. Usability of the Generated Debuggers

<table>
<thead>
<tr>
<th>Project</th>
<th>Total term-size</th>
<th>Compile duration normal code</th>
<th>Compile duration instrumented code</th>
<th>Percentage increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>EntityLanguage</td>
<td>2264</td>
<td>16.48s</td>
<td>28.35s</td>
<td>72%</td>
</tr>
<tr>
<td>Box</td>
<td>2339</td>
<td>5.39s</td>
<td>9.92s</td>
<td>84%</td>
</tr>
<tr>
<td>Spt</td>
<td>10518</td>
<td>23.43s</td>
<td>134.00s</td>
<td>472%</td>
</tr>
<tr>
<td>Stratego</td>
<td>37417</td>
<td>54.04s</td>
<td>180.44s</td>
<td>234%</td>
</tr>
<tr>
<td>Metac</td>
<td>64904</td>
<td>33.45s</td>
<td>16.57s</td>
<td>308%</td>
</tr>
<tr>
<td>StrategoXT</td>
<td>166719</td>
<td>145.20s</td>
<td>642.00s</td>
<td>342%</td>
</tr>
</tbody>
</table>

Figure 7.7: Project compilation duration on normal code and instrumented code.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Normal Duration</th>
<th>Instrumented Duration</th>
<th>% Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1: parsing</td>
<td>2.24s</td>
<td>3.74s</td>
<td>67%</td>
</tr>
<tr>
<td>Stage 2: importing</td>
<td>8.47s</td>
<td>13.09s</td>
<td>54%</td>
</tr>
<tr>
<td>Stage 3: typechecking</td>
<td>6.40s</td>
<td>7.90s</td>
<td>23%</td>
</tr>
<tr>
<td>Stage 4: model-to-model</td>
<td>33.57s</td>
<td>40.54s</td>
<td>21%</td>
</tr>
<tr>
<td>Stage 5: access control</td>
<td>12.15s</td>
<td>13.06s</td>
<td>7%</td>
</tr>
<tr>
<td>Stage 6: performing</td>
<td>2.94s</td>
<td>4.14s</td>
<td>41%</td>
</tr>
<tr>
<td>Stage 7: renaming</td>
<td>0.93s</td>
<td>1.45s</td>
<td>56%</td>
</tr>
<tr>
<td>Stage 8: generation</td>
<td>0.04s</td>
<td>0.07s</td>
<td>83%</td>
</tr>
<tr>
<td>Stage 9: code generation</td>
<td>145.06s</td>
<td>686.70s</td>
<td>373%</td>
</tr>
<tr>
<td>Stage 10: writing files</td>
<td>0.89s</td>
<td>1.33s</td>
<td>49%</td>
</tr>
<tr>
<td>Total Duration</td>
<td>248s</td>
<td>812s</td>
<td>228%</td>
</tr>
</tbody>
</table>

Figure 7.8: YellowGrass build duration on normal code and instrumented code.

Analysis of the Stratego compiler logs show that the Stratego compiler front-end phase takes about 30%-45% of the total compilation time for non-instrumented files and 50%-65% when using instrumented files. During the front-end phase multiple traversals over the entire program are made to desugar the program to a minimal core language. Adjusting the front-end phase to ignore the debug event calls during the traversal might be enough to improve the compile-time duration. Another approach would be to make the debug instrumentation part of the front-end phase, but care has to be taken to ensure that the debug information corresponds to the non-desugared program.

WebDSL

Figure 7.8 shows the compilation duration of YellowGrass on normal (non-instrumented) code and instrumented code per WebDSL compiler stage, together with the Java compilation duration and the total build duration. The code generation stage has the largest share in the compile-time duration for normal (58% of total duration) and instrumented code (85% of...
7. Evaluation Criteria Results

<table>
<thead>
<tr>
<th>Type</th>
<th>Total Java File Size</th>
<th>Class File Count</th>
<th>Class File Size</th>
<th>Jar Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>25.2 MB</td>
<td>11723</td>
<td>18.5 MB</td>
<td>11.1 MB</td>
</tr>
<tr>
<td>instrumented</td>
<td>61.0 MB</td>
<td>12392</td>
<td>29.9 MB</td>
<td>15.2 MB</td>
</tr>
</tbody>
</table>

Figure 7.9: Measurements on generated StrategoXT artifacts.

During this stage multiple passes over the program are made to analyse the control flow within templates and functions. The addition of the debug event calls negatively influence the traversal duration. The same suggestions for the Stratego compiler front-end are applicable to this: The traversal is adjusted to ignore debug event calls or the debug instrumentation is performed after the analysis.

Fortunately, the WebDSL compiler has a caching mechanism so subsequent builds with small modifications to the source code do not require a full re-compile. For example, after changing the title of a WebDSL-page the total build duration on instrumented files drops from 812 seconds to 99 seconds, with only 8.42 seconds spent in the code generation stage.

**EV5: What is the file-size overhead of the generated binaries when debug instrumentation is applied (in bytes)?**

Stratego and WebDSL both generate Java binaries. For Stratego the binaries are either standalone executable jars or they can be included in other applications as a library. The WebDSL compiler generates Java web applications which are deployed to a Tomcat instance using a war file (Web application ARchive)\(^5\).

**Stratego**

The StrategoXT build generates Java from Stratego, which is then compiled and packaged into a jar. Figure 7.9 show the characteristics of the artifacts generated by the StrategoXT build. A remarkable difference is that the instrumented build of StrategoXT has generated approximately 5.7% more class files than the normal build. The increased number of classes is due to the lifting of strategy arguments to a new strategy. Figure 7.10 shows how a strategy argument will be lifted to a new strategy by the Stratego compiler.

While the total size of the generated Java files is 61.0 MB (almost 3 times as big as for a normal build), the final jar is only 15.2 MB (1.5 times larger than the normal jar) due to compression techniques used in the jar. In theory the increased number of classes and the bigger size of the classes will increase the class loading time, but this overhead isn’t significant when compared to the runtime overhead.

\(^5\)A war file is a jar file with a specific directory layout to define a web application following the Java Servlet specification.
7.1. Usability of the Generated Debuggers

module lift-s
imports
libstrategolib
strategies
foo(s) = !"Stuff"; s
main = foo(debug ; debug ; debug)
lifted = debug ; debug ; debug

Figure 7.10: Stratego lifted call arguments example. The left fragment shows the original source code, the right fragment shows the source code after lifting.

<table>
<thead>
<tr>
<th>Type</th>
<th>Total Java File Size</th>
<th>Class File Count</th>
<th>Class File Size</th>
<th>War Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>22.9 MB</td>
<td>3371</td>
<td>19.5 MB</td>
<td>33.4 MB</td>
</tr>
<tr>
<td>instrumented</td>
<td>44.9 MB</td>
<td>3390</td>
<td>42.8 MB</td>
<td>36.0 MB</td>
</tr>
</tbody>
</table>

Figure 7.11: Measurements on generated YellowGrass artifacts.

WebDSL

Figure 7.11 shows that the size of the generated war slightly increases with instrumented code. Besides the Java classes generated from WebDSL code the war file also contains several Java libraries required for WebDSL, approximately 50% of the size of the war file can be contributed to those libraries.

EV6: What is the run-time duration overhead of debug instrumented programs?

The runtime overhead is determined by the event firing and event handling overhead. The event firing overhead is introduced by the extra debug statements inserted during debug instrumentation. The event handling overhead is the time needed by the debugger to extract the debug information, update the program state and check if a breakpoint is hit. Stratego and WebDSL use the same Java debug runtime library for firing events and the same event handling logic, therefore we expect that the overhead is comparable.
module tiny

imports
    libstrategolib

strategies
    main = foo; ![3, 9]; map(inc)

foo = debug

Figure 7.12: Source code of tiny-module used in Figure 7.13.

<table>
<thead>
<tr>
<th>Event Firing</th>
<th>Compiler Type</th>
<th>Duration</th>
<th>Total Overhead</th>
<th>Avg. Overhead Per Event</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vanilla w/o debugger</td>
<td>2.56s</td>
<td>79.90s</td>
<td>12.0µs</td>
</tr>
<tr>
<td></td>
<td>DI w/o debugger</td>
<td>82.46s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Event Handling | DI w/o debugger        | 82.46s   |                |                        |
|                | DI w/ debugger         | 7717.02s | 7634.56s       | 1.14ms                 |

Figure 7.13: Stratego Compiler runtime event overhead for the compilation of the tiny-module. During the compilation of the module approximately $6.67 \times 10^6$ events are fired.

Stratego

The Stratego Compiler is written in Stratego and is part of the StrategoXT build, therefore we can use it to measure the runtime penalty of running debug instrumented Stratego programs. To measure the runtime penalty for the Stratego Compiler we use four different compiler configurations:

- A vanilla Stratego Compiler without an attached debugger (Vanilla w/o debugger).
- A vanilla Stratego Compiler with an attached debugger (Vanilla w/ debugger).
- A debug instrumented Stratego Compiler without an attached debugger (DI w/o debugger).
- A debug instrumented Stratego Compiler with an attached debugger (DI w/ debugger).

We explicitly distinguish between executing the compiler with and without an attached debugger to determine the overhead of the communication between the debugger and the compiler.

To get an initial estimate of the overhead we first compiled the tiny-module listed in Figure 7.12. During the compilation of the module approximately $6.67 \times 10^6$ events are fired.
### 7.1. Usability of the Generated Debuggers

<table>
<thead>
<tr>
<th>Compilation Step</th>
<th>Vanilla w/o debugger</th>
<th>Vanilla w/ debugger</th>
<th>DI w/o debugger</th>
<th>DI w/ debugger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front-end</td>
<td>13.15s</td>
<td>16.62s</td>
<td>3246.09s</td>
<td></td>
</tr>
<tr>
<td>Optimization</td>
<td>0.52s</td>
<td>0.91s</td>
<td>288.27s</td>
<td></td>
</tr>
<tr>
<td>Backend preprocessing</td>
<td>2.94s</td>
<td>3.92s</td>
<td>666.62s</td>
<td></td>
</tr>
<tr>
<td>Backend transformation</td>
<td>2.18s</td>
<td>3.43s</td>
<td>252.75s</td>
<td></td>
</tr>
<tr>
<td>Backend optimization</td>
<td>2.21s</td>
<td>2.54s</td>
<td>333.65s</td>
<td></td>
</tr>
<tr>
<td>Backend postprocessing</td>
<td>0.16s</td>
<td>0.26s</td>
<td>1.11s</td>
<td></td>
</tr>
<tr>
<td>Pretty printing</td>
<td>4.33s</td>
<td>4.27s</td>
<td>1824.25s</td>
<td></td>
</tr>
<tr>
<td><strong>Total Compilation</strong></td>
<td><strong>25.66s</strong></td>
<td><strong>32.09s</strong></td>
<td><strong>6619.25s</strong></td>
<td><strong>±5.5 days</strong></td>
</tr>
</tbody>
</table>

Figure 7.14: The Stratego compile time measurements for the EntityLang-project broken down by compiler step. During the compilation of the module approximately $4.2 \times 10^6$ events are fired. The **Total Compilation** time for *DI w/ debugger* is estimated using the measurements from the *tiny*-module.

Fired. Figure [7.13](#) shows that the *event firing* overhead introduced by adding the debug events is approximately 0.0120 milliseconds per event. The *event handling* overhead introduced by attaching a debugger is approximately 1.14 milliseconds per event.

Next, we compiled a basic Spoofax project⁶ using the four configurations. Figure [7.14](#) shows the Stratego compile time duration broken down by the compiler phases as they are shown in the logs. Unfortunately we were not able to measure the compilation duration for the *DI w/ debugger* configuration due to technical difficulties⁷. Using the event handling overhead calculated for the *tiny*-module in the previous paragraph we can estimate the total duration for the *DI w/ debugger* configuration at minimally 478,781 seconds which corresponds to 5.5 days.

For the EntityLang-project the average event firing overhead per event is 0.0160 milliseconds and has the same order of magnitude as the *tiny*-module. The event firing duration of the compilation of the *tiny*-module and the EntityLanguage-project is respectively 31 and 205 times as large as the normal compilation duration. The total run-time overhead of the invocation of the Stratego Compiler for the *tiny*-module is 3013 times as large as the normal compilation duration.

Generally, Stratego programs are very event intensive due to the fact that traversal strategies are commonly used and walk over an entire term. For example, the number of events fired during a bottom-up strategy call is equal to six times the term-size of the traversed term. Therefore, the overhead of long running Stratego programs or program that operate on large terms quickly becomes unacceptable.

---


⁷During these long-running tests we eventually got a `org.eclipse.jdi.TimeoutException` fired by the JDI event receiving code which caused the program under the debugger to stop responding.
7. Evaluation Criteria Results

<table>
<thead>
<tr>
<th>Request URL</th>
<th>Vanilla</th>
<th>DI</th>
<th>Debugged</th>
<th>Event Count</th>
<th>Event Firing</th>
<th>Event Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>54ms</td>
<td>87ms</td>
<td>2487ms</td>
<td>1951</td>
<td>16.9µs</td>
<td>1.23ms</td>
</tr>
<tr>
<td>/project/SpaceDancing</td>
<td>50ms</td>
<td>83ms</td>
<td>2568ms</td>
<td>2018</td>
<td>12.6µs</td>
<td>1.23ms</td>
</tr>
<tr>
<td>/issue/SpaceDancing/9</td>
<td>41ms</td>
<td>50ms</td>
<td>969ms</td>
<td>740</td>
<td>12.2µs</td>
<td>1.24ms</td>
</tr>
<tr>
<td>/project/PurpleButter</td>
<td>65ms</td>
<td>110ms</td>
<td>3812ms</td>
<td>3041</td>
<td>14.8µs</td>
<td>1.22ms</td>
</tr>
<tr>
<td>/user/rtlindeman</td>
<td>70ms</td>
<td>108ms</td>
<td>4153ms</td>
<td>3297</td>
<td>11.5µs</td>
<td>1.23ms</td>
</tr>
<tr>
<td>/issue/FooBar/5</td>
<td>40ms</td>
<td>57ms</td>
<td>1268ms</td>
<td>998</td>
<td>17.0µs</td>
<td>1.21ms</td>
</tr>
<tr>
<td>/createIssue/FooBar</td>
<td>35ms</td>
<td>44ms</td>
<td>562ms</td>
<td>391</td>
<td>23.0µs</td>
<td>1.32ms</td>
</tr>
</tbody>
</table>

Figure 7.15: YellowGrass runtime event overhead per page request.

WebDSL

Figure 7.15 shows the duration of several YellowGrass page requests for a non-instrumented (Vanilla), an instrumented without debugger (DI) and an instrumented with a debugger (Debugged) YellowGrass instance. For each page request we have measured the number of events (Event Count), the average event firing overhead (Event Firing) and the average event handling overhead (Event Handling). The request duration measurements were made using Apache JMeter 2.8 and each request was repeated 200 times to minimize random measurement errors.

The table shows that the average event firing duration and event handling duration is in line with the results from the Stratego use case presented in Figure 7.13. Due to the less event intensive execution model of a WebDSL application the page requests for an instrumented application with a debugger connected are between 10 and 60 times slower than a non-instrumented application, which is within an acceptable range.

**EV7: How much effort and time does it take to switch from normal to debugging mode and back?**

For both Stratego and WebDSL programs switching from normal to debugging mode can be achieved by changing a property in the ant build scripts of the program. However, before the program can be launched under a debugger it has to be rebuild entirely. First the instrumentation tool has to traverse the entire source tree and then the compiler is invoked on the instrumented source files. As discussed in the previous paragraphs this generates a large overhead compared to a normal build. Also, switching back to normal mode requires a full rebuild.

The overhead could be reduced by caching the output of the instrumentation tool or by applying a partial instrumentation that only instruments the changed parts of a source file. Furthermore, a compiler with support for incremental compilation, such as the WebDSL compiler, lowers the overhead significantly as shown in Section 7.1.

7.1. Usability of the Generated Debuggers

7.1.1. Usability of the Generated Debuggers

36 ...  
37 bar = s  
38 s = ![3, 10]  
39 ; map(  
40   debug  
41 ; foo-instr  
42   )  
43 )  
44 foo-instr = debug  
45 ...  

Figure 7.16: Call stack structure for higher order non-instrumented strategies. The program is suspended at the statement on line 45 (depicted by the green rectangle).

EV8: What is the accuracy of origin tracking in the presence of program transformations?

To ensure that the origin of the debug events corresponds to a location in the source code as it is presented to the user in an editor the debug instrumentation is applied to the source files before they are desugared. This allows the debugger to highlight the current statement in the editor that corresponds to the last fired in the suspended program. The editor services in the IDE that depend on the origin information, such as type checking and semantic analysis, are not affected by the debug instrumentation because the instrumentation transformation is only applied during program compilation.

EV9: Is it possible to debug programs that link to separately compiled libraries (which may not be instrumented)?

The debugger can attach to instrumented programs that link to separately compiled libraries that are not instrumented. Consequently, when a function defined in the library is called from the instrumented program it will not generate any debug events. The function call is executed as an atomic operation.

When the non-instrumented library function call takes another instrumented function as argument, i.e. a higher order function such as the map strategy in Stratego, the stack frames in the debugger will not correspond to the actual stack frames. Figure 7.16 shows the stack frames of a Stratego program (suspended at line 45) with a non-instrumented map strategy. The arrows show the relationship between the source code location and the stack frame. The call stack misses a stack frame for the map strategy call between s and foo-instr.
7. Evaluation Criteria Results

<table>
<thead>
<tr>
<th></th>
<th>Java LOC(^9)</th>
<th>Stratego LOC(^{10})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handwritten Stratego Debugger</td>
<td>9240</td>
<td>1186</td>
</tr>
<tr>
<td>Language Independent Debugger</td>
<td>6107</td>
<td>1160</td>
</tr>
<tr>
<td>Stratego Debugger Component</td>
<td>2719</td>
<td>644</td>
</tr>
<tr>
<td>WebDSL Debugger Component</td>
<td>1158</td>
<td>343</td>
</tr>
</tbody>
</table>

Figure 7.17: LOC of the debugger components.

7.2 Usability of the Debugger Construction Framework

In this subsection we discuss the evaluation criteria that are important for a DSL designer that wants to implement a DSL debugger.

**EV10**: How much is the programming effort (in LOC) reduced comparing a hand-written debugger versus a generated debugger?

Figure 7.17 shows the lines of code for the handwritten Stratego debugger and the language-independent framework and its implemented debuggers. For the Stratego Debugger we achieved a reduction of 71% for the Java code and 46% for the Stratego code. Still most of the Stratego Debugger Java code is boiler plate code required to properly implement a debugger in the Eclipse IDE. Class definitions that are mandatory in the plugin specification just have to extend a base class implemented in the Spoofax debugger framework. Approximately 300 lines of Java boiler plate code can be generated from generic templates to reduce the implementation effort even more.

**EV11**: What is the ratio of language-independent code in the framework to language-specific code (in LOC) for a DSL debugger?

The ratio of language-independent code in the framework to language-specific code gives an indication of the relative amount of work required to implement a debugger. A higher ratio means less DSL-specific debugger code, which is better. For the Stratego debugger the ratio for Java code is 2.25 and for Stratego code it is 1.80. For the WebDSL debugger the ratio for Java code is 5.27 and for Stratego code it is 3.38. The lower Java code ratio for the Stratego debugger is due to extra functionality in the debugger, namely support for instrumenting, building and launching Stratego modules as stand-alone programs in addition to connecting to a running instrumented program. The WebDSL debugger on the other hand only supports connecting to a running WebDSL application. The lower Stratego code ratio for the Stratego debugger is due to the an extra post-instrumentation transformation and a more complex exit-event generation transformation for rule definitions.

\(^9\)The Java LOC are measured in physical Source Lines of Code using SLOCCount [55].

\(^{10}\)The Stratego LOC is calculated by counting all non-empty lines in a file after layout normalization and comment removal using pp-stratego.
7.3. Language-Parametricity of the Framework

EV12: How easy is it to include the debug instrumentation phase in an existing toolchain?

The debug instrumentation tool is distributed as a jar file and can be invoked from the command line as well as included as an Ant target in an Ant build script. WebDSL projects as well as Spoofax project that contain Stratego code are generally build using Ant scripts. Modifying those scripts to include the debug instrumentation phase did not require much effort due to the explicit target dependencies ant scripts.

For example, the Ant build scripts in a Spoofax project have a `stratego.jvm.helper` target that calls the Stratego to Java compiler. When we modify the signature of the target from

```xml
<target name="stratego.jvm.helper"
   unless="strc-java.availabe"
   if="build.stratego.enabled">
</target>
```

to

```xml
<target name="stratego.jvm.helper"
   unless="strc-java.availabe"
   if="build.stratego.enabled"
   depends="debug-instrument">

   <!-- Call debug instrumentation tool -->
</target>
```

then the `debug-instrument` target defined as follows:

```xml
<target name="debug-instrument"
   if="debug.instrument.enabled">
   <!-- Call debug instrumentation tool -->
</target>
```

then the `debug-instrument` target will only be called when `debug.instrument.enabled` is set.

Historically Stratego programs were compiled using GNU Make but we were unable to successfully include the debug instrumentation phase in an existing GNU Make script. Only recently the StrategoXT library build process was converted from GNU Make to Apache Ant. Modifying the Ant build script required significantly less effort than modifying the GNU Make script.

7.3 Language-Parametricity of the Framework

In this subsection we identify the language-independent and language-parametric components in the architecture of the framework.

EV13: How many components are language-independent?

The debugger framework contains four language-independent components, these components have a fixed implementation for each DSL debugger.
Event types  Even though the event types cannot be considered a component, they are the foundation of the debugger framework. The case studies show that the enter, exit, step and var events are sufficient to model the stack frames and variables in languages with imperative and declarative constructs. The event types are fixed in the current implementation of the framework. Consequently, the program state updating logic can also remain fixed.

Program State Controller  The program state controller is responsible for updating the execution context and execution state after a debug event is received. No DSL-specific logic is required because each event type has a required set of language-independent debug information, such as function name (required for enter and exit events) or location information (required for all events). The language-independent representation of the program state contained within the program state controller drives the content of the Eclipse Debugger user interface component that shows the call stack of a suspended program.

Breakpoint Hit Handler  After an event is processed by the program state controller the breakpoint hit handler comes into action. It compares the current program state with all the enabled breakpoints to determine if the execution should be suspended. The debugger framework only supports line breakpoints, therefore the breakpoint representation is the same for each debugger which allows a language-independent breakpoint handling algorithm.

Step Controller  Whenever a user issues a step action, the step controller has to determine at which event the program should be suspended. As discussed in Section 3.2.3 this only requires inspecting the event-type of the received events and the program state.

EV14: How many components are language-parametric?

The language-parametric components provide hooks to implement DSL-specific behavior. Program launching and program interactions such as suspend, resume and terminate are language-parametric because they depend on the DSL implementation platform. The launch delegate component is responsible for preparing a program before it can be launched under debugger, by calling the appropriate build scripts and starting the program with the correct environment variables.

The VM container component acts as a facade for the suspend, resume and terminate actions and the debug event information extraction to hide the DSL-implementation specific behavior. DSLs that use the same implementation platform, can extend a base component that is tailored for a specific implementation platform. For example, both Stratego and WebDSL use Java as their implementation platform therefore both debuggers have to communicate with the JVM. The low-level details of the JVM communication can be hidden in an extensible VM container component that targets the JVM to enable reusable blocks at the DSL implementation platform level.
7.3. Language-Parametricity of the Framework

**EV15: To what extent is the approach applicable to other languages?**

In Section 4.1 we have discussed the properties of a DSL and how they impact the design of a debugger. We can now reflect on these properties using the Stratego and WebDSL case study.

**Executability** 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. For example, HTML web pages are not “executable”, but when the HTML is generated from a templating language it becomes possible to debug the generation of the HTML as demonstrated with the WebDSL debugger.

**Programming Paradigm** 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 while our debugger approach explicitly depends on the execution model to fire meaningful events. Nevertheless, the debugger can be used to show how the relationship between declarations is constructed improving program understanding. The WebDSL case study has shown that it is possible to debug the declarative template language using the enter, exit and steps events because of the tree-like layout of the user interface components. Additionally, the availability of a try-finally block construct in the language reduces the instrumentation complexity for the multiple-exit problem. It may be worthwhile to extend the language with such a construct.

**DSL Implementation Approach** We only evaluated our approach for interpreted and compiled DSLs. Debuggers for DSLs implemented as language extensions require special attention, as we only need to 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 DSL debugger with a debugger for the host language to seamlessly transition between GPL code and DSL code.

While this thesis focuses on debugging textual DSL programs, the same debugger generation approach can also be applied to domain-specific modeling languages (DSML). According to previous work DSML debuggers should debug both the synthesized artifacts and the model transformations [34]. First, our approach can be used when the synthesized artifacts are code and the model can be expressed as a textual DSL. For visual DSLs the instrumentation preprocessor and debugger runtime have to be modified to interact with the visual model. Secondly, the Stratego use case shows how our approach can be used to create a debugger for a transformation language. Furthermore, the instrumentation preprocessor can be used to link constructs at different levels of abstraction during a model transformation.
Chapter 8

Discussion and Related Work

This thesis 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 [14, 46, 22, 57]. In this chapter we reflect over our work and highlight the differences to related work.

8.1 Discussion

In this section we discuss the validity of our work. We have performed two case studies and made several measurements in this thesis to answer the evaluation criteria and to support various claims. In this section we will therefore discuss the threats to validity of our work.

Several measurements made for the evaluation criteria lack a formalized measuring procedure which makes the results subjective. For EV1 we have only compared the functionality of a handwritten Stratego debugger with a debugger generated using our framework. The definition of “the completeness of the debugger functionality” is ambiguous because it depends on the existence of an existing reference implementation. Although we have specified a fixed set of debugger requirements these requirements might not reflect the expectations of existing DSL program developers (which may even vary per DSL) making our DSL debugger requirements subjective. Furthermore EV2, EV7 and EV12 also lack an objective measurement procedure. The validity can be improved by performing a study with multiple participants and multiple debugger implementations for various DSLs to eliminate the personal bias of the author and to identify differences between implementations, respectively.

In order to get reliable quantitative measurement results we repeated all duration-related measurements 5 times and calculated the average to eliminate random errors introduced by OS activity. Additionally, each Java program was started in a new JVM to prevent biased measurements, since repeating the program invocation in an existing JVM could significantly decrease the duration for each repeated invocation because a previous run might have triggered JIT optimizations. The variation of measurements was within 5% of the average which we found acceptable.

Even though Stratego and WebDSL are different DSLs, they share the same DSL imple-
mentation platform, namely Java. This shows that our layered architecture allows reuse at the DSL implementation platform level, but these case studies do not support our claim that our approach also holds for DSLs with a different implementation platform than Java. For this we need to implement a debugger that uses a different implementation platform than Java, for instance Mobl uses JavaScript as its implementation platform. Another case study to consider is implementing a debugger for a DSL that supports multiple implementation platforms to determine if and how this affects the reusability of debugger components.

8.2 Related Work

We can split the related work in two groups. First we will discuss tools for building debuggers that support generation of some parts of a debugger implementation. Second we will discuss libraries that can be used as part of a hand-written debugger implementation.

8.2.1 Tools for Building Debuggers

The Meta-Environment [28] 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 [46] 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. [57] 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 using an aspect oriented approach. 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 [41], they weave debug instrumentation into these semantic actions. In contrast, our approach relies on a separate debug instrumentation tool operating directly on the DSL program source code which does not depend 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) [53] 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.
8.2. Related Work

8.2.2 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) \[8\] 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.
Chapter 9

Conclusions and Future Work

9.1 Conclusion

In the introduction we explained the relationship between the evaluation criteria groups and the research question. Therefore, we will first summarize the results of each evaluation criteria group before we answer our main research question.

Usability of the Generated Debuggers (RQ part a) Debuggers are an important tool in modern software engineering practice. Full tool support with debugging facilities ensures optimal productivity of developers with DSLs. The results show that the generated debuggers can compete with handwritten debuggers in terms of functionality, but come up short performance-wise. Our approach causes a compile-time and run-time overhead for instrumented programs. While the compile-times for instrumented programs are 2-6 times as large, the overhead decreases significantly when the compiler supports separate compilation techniques. The run-time overhead per event is approximately 1 millisecond. For languages with a high event density, such as Stratego, the run-time duration for instrumented programs becomes 3 orders of magnitude larger. For languages with a lower event density, such as WebDSL, the run-time duration for instrumented programs only becomes 1-2 orders of magnitude larger.

Usability of the Debugger Construction Framework (RQ part b) The debugger framework presented in this thesis 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. The DSL developer has to define debug information extraction and debug event generation transformations for the instrumentation tool, extend the DSL runtime with a debug event sending library to communicate with the debugger and configure language-parametric components in the debugger for a proper integration with Eclipse.

Additionally, we want to emphasize that although this work is implemented as part of the Spoofax workbench, the concept is neither Spoofax, Eclipse, Java or Stratego-specific. The debug instrumentation tool can be implemented in any another transformation language.
9. **Conclusions and Future Work**

and the debugger can be integrated into another IDE. The platform-level debug library, platform-level event handler can be re-used as well as the part of the Spoofax debug runtime that processes the events and updates the language-independent program state.

**Language-Parametricity of the Framework (RQ part c)** For each component in the framework a trade-off was made between a language-independent design to reduce the implementation effort and a language-parametric design to support a larger variety of DSLs. The degree of parametricity in the framework was sufficient to implement the Stratego and WebDSL debugger, still a substantial amount of duplicate boilerplate code was needed for the Eclipse integration.

*Can we generate a DSL debugger that a) adds a run-time and compile-time overhead which is comparable to a conventional debugger with the same set of functions, b) using an approach that reduces the programming effort and c) is applicable to other languages?*

We conclude that we have only partially met the conditions stated in the research question. Specifically, the run-time overhead (part a) is significant and the applicability to other languages (part c) is not confirmed. Nonetheless, we have presented a language-independent debugger framework that contains a declarative debugger specification, a reusable language-parametric debug instrumentation tool and debugger runtime implementation architecture.

9.2 **Future Work**

In this section we list some directions for further research and improvements for our approach.

First and foremost, to improve the validity of our research more debuggers have to be built with our framework. Especially, building debuggers for DSLs that have an other platform implementation language than Java is a must.

Secondly, we have already hinted at the possibility of supporting DSL-specific events and program state information. The debug instrumentation tool and the SEL language are not limited to a fixed set of debug events. On the contrary the program state controller is language-independent and has to be refactored to a language-parametric component to inject custom event handlers that update the program state with DSL-specific information.

The third extension is in the field of data visualization. Currently data structures are represented as tree structures while other more sophisticated visualizations, such as graphs, could be of more benefit to the user. A similar extension was also addressed in the WebDSL case study: Debugging the control flow in a template call shows how the user interface controls are created but it does not show a direct link with the web page as it is shown in the browser. We envision a communication protocol between the debugger and the web page that with the help of origin tracking can show the relation between an HTML fragment in the browser and a WebDSL fragment in the IDE.

Furthermore, additional research is required to lower the run-time overhead of a debug instrumented program. The Stratego case study showed that the communication between
9.2. Future Work

a JVM and the debugger is of major significance. Alternative implementations that reduce the amount of communication might have a negative effect on the reusability. One specific approach comes to mind: When the program state controller and breakpoint hit controller are moved to the debuggee runtime, the debugger-debuggee communication is significantly reduced to the execution control actions (suspend, resume, terminate and step) and breakpoint hit notifications. However, this does force us to implement custom components for each DSL implementation platform.

Also, we view that our event-oriented framework is not limited to a debugger application. With a minor adjustment in the platform-level debug library it is possible to record the time stamp of the moment of firing. By calculating the difference between the enter and exit time for a stack frame we get DSL-specific profiling information. With this information we are not far from a DSL-specific profiler. Although not as precise as a native profiler, the stack frame duration can give an indication of bottlenecks at the DSL level. Further research has to show if and how such a profiler can be implemented on top of our approach.

Finally, it would be interesting to see if other debugger features such as conditional breakpoints defined in the syntax of the DSL, hot code replacement and immediate evaluation of DSL code fragments in the suspend program context can be implemented in a language-independent fashion.


http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdi/index.html

[40] Oracle. Java platform debugger architecture.
http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/


