Contributions to the Generation of Semantic Editors

Emma Söderberg

Licentiate Thesis, 2010

Department of Computer Science
Lund University

ISSN 1652-4691
Licentiate Thesis 12, 2010
LU-CS-DISS:2010-12

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: Emma.Soderberg@cs.lth.se
WWW: `http://cs.lth.se/Emma_Soderberg`

Typeset using LaTeX
Printed in Sweden by Tryckeriet i E-huset, Lund, 2010

ABSTRACT

The state-of-the-art in development tools today provides users with a large number of both syntactic and semantic services, such as syntax highlighting, name completions and refactorings. Preferably, every language should have sophisticated tool support, but unfortunately, the manual task of tool development is both time-consuming and error-prone, making it too costly for small language communities. This situation can be improved by the use of tool generators and frameworks supporting reuse. Tool generators can provide swift generation of language-specific components from high-level specifications, while frameworks can provide reusable language-generic components. In this thesis we address a number of problems related to the *generation of semantic services* used in editors. We present a *prototype tool generator* based on reference attribute grammars (RAGs). RAGs extend attribute grammars with references that turn syntax trees into graphs, and we find them both powerful and expressive, suitable for the generation of semantic services. As an example of how RAGs can be used for advanced semantic services, we show how they can modularly express the semantics of *flow analysis*, using Java as a large case study. Flow analysis is useful in several semantic services, including dead code detection, and as a contributing service in a refactoring tool. We also present contributions to improving the performance of RAG-based systems. The approach is to use profiling to automatically compute a selective *caching configuration* of attributes, performing better than full caching. We have chosen to focus on text-based editors, which leaves us with the predicament of translating text to syntax tree through parsing. The text in an interactive editor often contains parsing errors, making it difficult to maintain a corresponding syntax tree. To mitigate this problem, we present a novel approach to structural error recovery of text – *bridge parsing*, used as a pre-processor which makes it independent of the actual parser in use. In addition, we present how bridge parsing can be integrated with *scanner-less generalized LR parsing* – a fruitful combination outperforming the Eclipse JDT with regards to structural error recovery.

PREFACE

This thesis is for the Licentiate degree which is a Swedish degree between the MSc and PhD. It consists of an introductory part, three peer-reviewed papers and two technical reports. The papers included in this thesis are[1]:

I. **Extensible Intraprocedural Flow Analysis at the Abstract Syntax Tree Level**[2] Emma Söderberg[3], Görel Hedin, Torbjörn Ekman, Eva Magnusson. Submitted to the journal of *Science of Computer Programming*, Elsevier 2010

II. **Practical Scope Recovery using Bridge Parsing** Emma Nilsson-Nyman[4], Torbjörn Ekman, Görel Hedin. In the proceedings of the *1st Conference on Software Language Engineering* (SLE'08), Toulouse, France, September 2008 *Lecture Notes of Computer Science* 5452:95–113 © 2009 Springer Berlin / Heidelberg

III. **Natural and Flexible Error Recovery for Generated Parsers** Maartje de Jonge, Emma Nilsson-Nyman[5], Lennart C.L. Kats, Eelco Visser. In the proceedings of the *2nd Conference on Software Language Engineering* (SLE'09), Denver, Colorado, USA, October 2009 *Lecture Notes of Computer Science* 5969:204–223 © 2010 Springer Berlin / Heidelberg

[1]The author of this thesis has recently changed her last name from Nilsson-Nyman to Söderberg

[2]An extended version of the following paper: **Declarative Intraprocedural Flow Analysis of Java Source Code** Emma Nilsson-Nyman, Torbjörn Ekman, Görel Hedin, Eva Magnusson. Proceedings of the *8th Workshop on Language Descriptions, Tools and Applications* (LDTA'08), Budapest, Hungary, April 2008. Electronic Notes of Theoretical Computer Science 238(5):155–171 © 2009 Elsevier B.V.

[3]The main author and responsible for the implementation and the evaluation.

[4]The main author and responsible for the algorithm, the implementation and the evaluation.

[5]Contributed with the integration of bridge parsing with SGLR, the implementation of the bridge parser generator and parts of the evaluation of the approach.

Technical reports included in this thesis are:

I. **Automated Selective Caching for Reference Attribute Grammars** Emma Söderberg[6] and Görel Hedin. *Technical report, LU-CS-TR:2010-245, ISSN 1404-1200, Report 94, 2010*, Department of Computer Science, Lund University

II. **A Semantic Editing Model in Support of Reference Attribute Grammars** Emma Söderberg. *Technical report LU-CS-TR:2010-246, ISSN 1404-1200, Report 95, 2010*, Department of Computer Science, Lund University

The author has also contributed to the following peer-reviewed papers, not included in this thesis:

- **Providing Rapid Feedback in Generated Modular Language Environments: Adding Error Recovery to Scannerless Generalized-LR Parsing** Lennart C.L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, Eelco Visser. Proceedings of *24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications* (OOPSLA'09), 445–464 Orlando, Florida, USA, October 2009 © 2009 ACM

- **A Plan for Building Renaming Support for Modelica** Görel Hedin, Emma Nilsson-Nyman, Johan Åkesson. Electronic proceedings of *3rd Workshop on Refactoring Tools* (WRT'09) Orlando, Florida, USA, October 2009 Available online at `http://refactoring.info/WRT09` [May 2010] (Hosted by University of Illinois, Department of Computer Science)

- **Ad-hoc Composition of Pervasive Services in the PalCom Architecture** David Svensson Fors, Boris Magnusson, Sven Gestegård Robertz, Görel Hedin, Emma Nilsson-Nyman. Proceedings of *International Conference on Pervasive Services* (ICPS'09), 83–92 London, United Kingdom, July 2009 © 2009 ACM

[6]The main author and responsible for the implementation, the evaluation and the design of the approach. However, the design of the approach evolved much during discussions with the co-author.

Acknowledgements

The work in this thesis is partly funded by Vetenskapsrådet (The Swedish Research Council), VINNOVA (The Swedish Governmental Agency for Innovation Systems), and EPSRC (The Engineering and Physical Sciences Research Council of the UK). I am also grateful for grants from The Royal Physiographic Society in Lund, and The Google Anita Borg Memorial Scholarship.

The work presented in this thesis has been carried out within the *Software Development and Environments Group* (SDE) at the Department of Computer Science at Lund University, within the *Programming Tools Group* (PTG) at the Computing Laboratory, University of Oxford, and in collaboration with the *Software Engineering Research Group* (SERG) at TU Delft.

My first and foremost thanks goes to my supervisor Dr. Görel Hedin to whom I am sincerely grateful for advice, support and patience during my exploration of the fields of reference attribute grammars and semantic editors. She originally inspired me to start this journey, and her knowledge and experience has been invaluable during the work on this thesis. I am also grateful to my assistant supervisor Prof. Boris Magnusson for advice and discussions.

I am very grateful to Dr. Torbjörn Ekman for advice and for giving me support and early encouragement. Particularly, for inviting me to work with his group (PTG) at the Computing Laboratory at the University of Oxford in the summer of 2007, and to Dr. Oege de Moor (the head of the group) for his invitation for yet another stint the summer of 2008. A big thanks to the remaining PTG members for providing a great place to work and for making my visits most enjoyable. Special thanks to Max Schäfer for using JedGen and our flow analysis modules, and to Dr. Oege de Moor for using the flow analysis modules in his teaching. Thanks to Magdalen College for providing a splendid place to stay during these visits.

I am also grateful to Dr. Eelco Visser for showing an interest in our work on scope recovery and for inviting me to visit his group (SERG) at TU Delft in the winter of 2009. A big thanks to my co-authors at TU Delft, Lennart Kats and Maartje de Jonge for fruitful discussions and nice work. Thanks to the remaining SERG members whom I have had the pleasure to meet at various occasions.

Thanks to Dr. Eva Magnusson for joint work on control-flow and for sharing her knowledge on collection attributes. Thanks to Dr. Johan Åkesson and the people at Modelon for joint work on the JModelica IDE and for using JedGen. Thanks to Erik Mossberg, Jesper Mattsson and Philip Nilsson for using early versions of JedGen in their master thesis work. Thanks to past and present co-authors and colleagues in the SDE group: Sven Gestegård Robertz, David Svensson Fors and Thomas Forsström for joint work. Thanks to Dr. Robert Fuhrer at IBM TJ Watson Research Center, Hawthorne, NY for hosting my visit in October of 2009.

A big thanks to all colleagues and students at the Department of Computer Science in Lund for providing a nice and relaxed place to work. Thanks to Dr. Klas Nilsson for initially employing me as a research assistant at the department of Computer Science during the summer of 2005, and to Per Holm for employing me as a teaching assistant in the fall of 2003. Thanks to Dr. Flavius Gruian for advice on writing, to Dr. Roger Henriksson for help and advice, and to Dr. Jonas Skeppstedt for interesting discussions about compilers. Thanks to Lhinn Nilsson-Nyman, Lena Ohlsson, Tomas Richter and the technical support crew, Anne-Marie Westerberg, and Jonas Wisbrandt for help with practical things.

Last but not least, I would like to thank my family and friends for all their support. A special thanks to everyone participating in various crazy ventures – the floor ball team, the badminton crew, the running group, the climbing maniacs and the happy golfers. Thanks to my parents (Christina and Lennart) and siblings (Caja, Lhinn and Carl) for all their love and support. A special thanks to my "inseparable twin" Lhinn who never grows tired of me dragging her into things. Thanks to my new in-laws (Majvor and Jan) and their children (Katarina and Fredrik) with families for making me feel most welcome. Last, but far from the least, I would like to thank my husband Christian, who fills my life with sunshine.

Emma Söderberg
June 2010

CONTENTS

I Introduction **1**
1 Methods, Languages and Tools 1
2 Towards the Generation of Semantic Editors 4
3 Long-term Goals and Challenges 8
4 Contributions 10
5 Future Work 12

Included Papers **25**

I Extensible Intraprocedural Flow Analysis at the Abstract Syntax Tree Level **27**
1 Introduction 28
2 Control-flow Analysis 29
3 Dataflow Analysis 39
4 Dead Code Analysis 45
5 Language Extensions 49
6 Evaluation 50
7 Related Work 54
8 Conclusions 56

II Practical Scope Recovery using Bridge Parsing **59**
1 Introduction 60
2 Background 61
3 Bridge Parsing 64
4 Bridge Parsing for Java 70
5 Evaluation 74
6 Conclusions 78

III Natural and Flexible Error Recovery for Generated Parsers 83
1 Introduction . . . 84
2 Error Recovery . . . 86
3 Coarse-grained Error Recovery . . . 87
4 Fine-grained Error Recovery . . . 93
5 Bridge Parsing . . . 94
6 Evaluation . . . 99
7 Related Work . . . 103
8 Conclusion . . . 104

Included Technical Reports 109

I Automated Selective Caching for Reference Attribute Grammars 111
1 Introduction . . . 111
2 Reference Attribute Grammars . . . 112
3 Attribute Instance Graphs . . . 114
4 Computing a Cache Configuration . . . 117
5 Evaluation . . . 118
6 Related Work . . . 124
7 Discussion and Future Work . . . 125
8 Acknowledgements . . . 126

II A Semantic Editing Model in Support of Reference Attribute Grammars 129
1 Introduction . . . 130
2 A Historical Note on the Development . . . 131
3 Background . . . 132
4 The Semantic Editing Model . . . 135
5 The Service Flora . . . 141
6 Current Status and Future Work . . . 145
7 Acknowledgements . . . 146

CHAPTER I

INTRODUCTION

Software is inherently difficult to develop. According to Brooks [Bro87], the difficulty lies in the essential complexity of the task. Not surprisingly, large amounts of research have been directed on facilitating software development, as summarized by Boehm [Boe06]. This thesis focuses on how to facilitate the development of *semantic editors*, that is, editors with knowledge of the meaning of a programming language.

1 Methods, Languages and Tools

Software development research can roughly be divided into three areas – *methods*, *languages* and *tools*. These areas are inter-connected, for instance, methods can be enforced through the use of tools and new languages benefit from tool support. The topic of this thesis belongs in the third area – tools.

1.1 Methods

Software projects are notorious for being hard to manage and often take more time than expected, according to Brooks in [Bro95]. Besides Brooks' observations, we should not neglect that software development is a human activity – humans do the work and humans make mistakes [Wei88]. Important parts of software development is about people management – people run projects, design solutions, implement programs, test applications, and so on.

One recent influential contribution to the methods of software development is *agile development* [BBvB$^+$01, HC01], focusing on individuals and interactions, working software, customer collaboration and responding to change. While agile development focuses more on people and teamwork than on languages and tools, good languages and tools can help support core agile values and principles [BA04], like communication, feedback and quality.

1.2 Languages

Good communication is essential to successful software development, both in written form and in speech. We communicate in many ways: in speech using natural language, body language, etc., and in written form using free text, forms, tables, etc. For communication with computers, we use languages for which we formally can describe structure and meaning so that computers "understand". Originally, these formal languages, or programming languages, were in much regard more developed for computers than for humans [Wei88]. The introduction of higher-level languages, with language constructs closer to human thinking, was a relief for software developers, as summarized by Brooks [Bro87]:

> *"Surely the most powerful stroke for software productivity, reliability and simplicity has been the progressive use of high-level languages for programming."*

Much research has been focused on so called *domain-specific languages* [vDK97, vDKV00] (DSLs), also referred to as *little languages* [Ben86], and application-oriented or special-purpose languages [SH03]. A DSL is a form of higher-level language, not necessarily Turing complete or executable, with language constructs close to a certain problem domain. At the same time, research has continued in the *general-purpose language* domain where interesting new languages like Scala [Ode04] have emerged, supporting more than one programming paradigm; notably *object-oriented programming* and *functional programming*. Another example of research related to the use of languages, whether they are in textual or graphical form, is that of *model-driven development* [Sel03], focusing on the composition of models and code generation. The intention being to replace high-level languages in a manner similar to how high-level languages replaced assembly code with the use of compilers.

While good languages are essential, they need to be integrated with tools in order to be fully taken advantage of.

1.3 Tools

Software development is an error-prone task: bugs are easily introduced and they can be both hard to find and remove. Even the smallest bug can have devastating consequences. For instance, the NASA Mars Climate Orbiter failed to land in 1999 due to a software bug [IBN99], and the U.S. telephone system broke down during 9 hours in 1990 due to another software bug [EDMW90]. However, computers do not logically make mistakes – they follow the instructions we give them even though those instructions may be faulty. Parnas expresses his view on human error as follows [PC85] :

> *"Human errors can only be avoided if one can avoid the use of humans."*

Tools can help us to *not* make mistakes, as well as, *speed-up* development. For instance, integrated development environments (IDEs), can provide developers with *instant feedback* on problems in their program, for example, by showing *compile-time errors*, supporting *refactorings* [Opd92, Fow99], and detecting *code smells* [VEM02]. Short feedback loops are important in recent agile methodologies where developers need to respond efficiently to code changes. Early examples of IDEs for dynamic languages include the Interlisp environment for LISP by Teitelman et al. [TM81] and the SmallTalk environment by Goldberg and Robson [GR83], both serving as inspiration for later tools supporting statically typed languages.

Today, the *state-of-the-art*, concerning tools in Java development, is jointly represented by IntelliJ IDEA [Jet10], NetBeans [Cor10] and the Eclipse Java Development Toolkit (JDT) [Fou10a]. These are all language-based tools for Java with knowledge of the semantics of the Java language. This knowledge is used to provide users with a large number of interactive syntactic and semantic services to facilitate development of Java programs.

1.4 Tools for All

Preferably, every language should have sophisticated tool support. However, state-of-the-art tools are normally hand-crafted and the result of years of development. This is unfortunate for small language communities where this type of development is too expensive, the result being that many language users do their programming in generic text editors. Two important ways of facilitating tool development are *reuse* and *generation*.

Reuse The Eclipse platform [DRW04, Fou10b], provides a plugin-based integrated platform facilitating development of tools and reuse of existing tools. For example, tools for version management and file browsing can be reused by new editing tools. Due to the extensibility of the platform, a number of projects building on the platform have emerged. The most well-known example is the Eclipse Java Development Toolkit (JDT). Another example is the Eclipse Modeling Project [The10a], which focuses on the development of model-based tools. This project includes several subprojects addressing different concerns. For example, the Eclipse Modeling Framework (EMF) by Budinsky et al. [BBM03], provides means for expressing structured data models. Other examples of platforms supporting reuse include the NetBeans IDE [Cor10] and IntelliJ IDEA [Jet10], both extendable through the use of plug-ins.

Generation An early example of tool generation includes the Synthesizer Generator by Reps and Teitelbaum [RT84], which generates syntax-directed editors supporting incremental re-compilation of code during editing. The GANDALF system by Habermann and Notkin [HN86] semi-automatically generates software development environments including syntax-directed editors, and also other tools needed in the software development process, for example version management tools. The Mjølner/ORM system, by Magnusson and Hedin [HM88], also supports the generation of syntax-directed editors but for statically typed object-oriented languages.

In recent years, a number of tool generating systems have emerged which extend the Eclipse Platform. Examples include the IDE Meta-tooling Platform (IMP) by Charles et al. [CFS$^+$09], which has a semi-automatic approach to the development of language-oriented textual editors with features like syntax highlighting, code outlines and name completion. IMP semi-generates these editors using wizards and generation of code skeletons. Developers manually fill in language-specific behavior in these code skeletons. The MontiCore system by Krahn et al. [KRV08], supports generation of textual editors with services like syntax highlighting and structural outlines. MontiCore uses a combined grammar format for concrete and abstract syntax supporting modular language extensions.

2 Towards the Generation of Semantic Editors

Our vision is to *generate semantic editors* with features similar to those found in today's *state-of-the-art* tools. With semantic editors, we mean editors with knowledge of the semantics (and the syntax) of a programming language. This knowledge is used to provide users with services utilizing *context-sensitive semantic information* about a program, like name completion. Compared to traditional text editing and compilation, semantic editors provide integrated compilation, using an *in-memory semantic program model*. The in-memory model is updated continuously to keep it synchronized with user changes in the editor.

The mentioned existing editor generators (IMP and MontiCore), focus on services like syntax highlighting, code folding and structural outlines. These services depend on lexical analysis and an in-memory model in the form of a parse tree, obtained from a parser. In order to support more advanced semantic services, like name completion, these editors can extract semantic information from the parse tree using the visitor pattern [GHJV95].

We want to support the generation of semantic editors with semantic services like name completion and refactorings. These services depend on semantic analyses, like name analysis and type analysis, normally performed on an in-memory model in the form of an *abstract syntax tree* (AST). To remove the need for visitors, we want to utilize an in-memory AST attributed with context-specific semantic information facilitating the generation of advanced services like refactorings.

2.1 Editing Styles

Semantic editors can support different editing styles: *textual*, *syntax-directed* and *graphical*. *Textual editing* provides an efficient editing style for expert programmers. It is widely used and supported in state-of-the-art tools like the Eclipse JDT. Text editors need to deal with incomplete nonparsable text which makes keeping the in-memory model synchronized with user changes challenging. In contrast, *syntax-directed editors* prevent this predicament by not allowing users to make syntactic mistakes. Instead, the editor provides users with a toolbox of parsable language templates which may be inserted, moved or removed in a program. Syntax-directed editors are useful for novice programmers who might be unfamiliar with the syntactic constructs of the language. For example, the Cornell Program Synthesizer by Teitelbaum and Reps [TR81] provides a syntax-directed editor which has been used in teaching of programming. The syntax-directed editing style is also used in, for instance, XML editors which are described using an XML schema or a Document Type Definition (DTD). In *graphical editing*, users modify graphical components in a "drag-and-drop" fashion. This style of editing is good for providing an overview and for visualization of relations. For instance, in UML editors classes and inter-class relations are visualized which provides an overview of the object-oriented design of an application.

Finally, the above mentioned editing styles can all use the same in-memory model, which means they can be combined to suite the specific needs of a certain domain. In this thesis we focus on textual editing due to its wide use in state-of-the art-tools.

2.2 The State of the Art

What features are comprised in the state-of-the-art of semantic editors today? A recent study by Hou and Wang [HW09], gives an answer to this question. The study focuses on the Eclipse JDT, but gives a good overview of features supported, not only by the Eclipse JDT, but also by the IntelliJ IDEA and the NetBeans tool.

Hou and Wang find the following grouping of visible features in the Eclipse JDT based on developer activities: *project setup* (e.g., handling of files, project configuration), *code manipulation* (e.g., reading and writing code), *build and run* (e.g., automatic building, execution of programs), *debugging* (e.g., breakpoint support, program execution state inspection) and *testing* (e.g., incorporated testing support using JUnit [Men10], a testing framework for Java).

Considering the code manipulation feature in more detail, Hou and Wang divide it into two sections: *reading* (e.g., display of program information like errors, code search and navigation) and *writing*. The writing category is further divided into three areas: *editing* (e.g., automatic indentation, generation of code fragments), *refactoring* (behavior-preserving code transformations, e.g., rename, inline method), and *code assist* (inferred code manipulation actions, e.g., quick fix, name completion). Semantic editors support the *code manipulation* features listed above.

Besides the feature grouping presented above, the code manipulation features can be divided into *lexical*, *syntactic* and *semantic*, depending on the type of analysis needed for realizing the feature, or service. For example, features like refactorings and code assist normally require semantic analysis, while syntax highlighting and automatic indentation normally require only lexical analysis and syntactic analysis (parsing), respectively.

2.3 Defining Semantics

There are a number of different formalisms available for describing the semantics of a programming language. Examples include *denotational semantics* by Scott and Strachey [Sco72, Str66], *Attribute grammars*, by Knuth [Knu68], *Natural semantics* by Kahn [Kah87], and *Algebraic semantics* by Bergstra et al. [Ber89]. These formalisms have all been used in the generation of semantic editors:

- *Denotational semantics* describe the meaning of a program using mathematical objects called *denotations*. The formalism is based on domain theory which is concerned with the mapping from one domain to another domain. The formalism is used in practice in, for example, the PSG system by Bahlke and Snelting [BS86].

- *Attribute grammars* (AGs) add semantic rules called attributes as extensions to a context-free grammar. Attribute grammars have been used in practice in many systems. For example, the Synthesizer Generator by Reps and Teitelbaum [RT84], generates syntax-directed editors with incremental evaluation of attributes. The APPLAB system by Bjarnason et al. [BHN99], provides an environment for interactive development of domain-specific languages. The Lrc system by Kuipers and Saraiva [KS98], a function-oriented implementation of *higher-attribute grammars* (HAGs) (an extension to AGs), generates programming environments with incremental evaluation of HAGs.

- *Natural semantics* is an extension of *structural operational semantics* by Plotkin [Plo81], which describes the meaning of a program based on how it is interpreted, that is, as a sequence of computational steps. The formalism is used in practice in the CENTAUR system by Borras et al. [BCD$^+$88], which provides a generic interactive environment customizable to a certain language using the TYPOL system by Kahn et al. [Kah87]. Natural semantics are also utilized in the RML system by Pettersson [Pet94].

- *Algebraic semantics*, or the *Algebraic Specification Formalism* (ASF) by Bergstra et al. [Ber89], describes semantics using algebraic equations. The formalism is used in practice in the ASF+SDF Meta-Environment by Klint [Kli93], which supports development of interactive language-based tools using language descriptions in ASF and the *syntax definition formalism* (SDF) by Heering et al. [HHKR89].

Some recent systems have made use of other means for expressing semantics. For example, the Spoofax/IMP system by Kats et al. [KdJNNV09], provides a language development environment built on IMP using various DSLs to specify editor services. The system uses strategic term rewriting to express language semantics, as supported by the Stratego system by Visser [Vis01]. The xText project by Efftinger and Völker [eff], provides a means for generating textual semantic editors in the EMF project. The system uses the Object Constraint Language (OCL) [The10b] to describe language semantics. The MontiCore system by Krahn et al. [KRV08], uses UML-like associations to, for example, describe declaration-use relations.

We will use the *reference attribute grammar* (RAGs) formalism [Hed00], an extension of attribute grammars, to generate semantic services, like name completion and refactorings. We are confident that RAGs can be used to further push the frontier of sophistication in generated semantic editors. As proofs of concept, RAGs have been shown to handle large compiler implementations. For example, the JastAddJ compiler for Java by Ekman and Hedin [EH07a] and the JModelica compiler for Modelica by Åkesson et al. [ÅEH10], both implemented using the RAG-based JastAdd system by Ekman and Hedin [JT, EH07b]. RAGs have also been used to define refactoring extensions to JastAddJ by Schäfer et al. [SEdM08, SVEdM09, SDS+10].

2.4 Attribute Grammars and Semantic Editors

Following the development of AGs and their use in semantic editors, we find that the formalism has been used in several systems. The first occurrence of the combination of AGs and semantic editors were in the Synthesizer Generator by Reps and Teitelbaum [RT84].

Attribute Evaluation The Synthesizer Generator supports a subclass of AGs called *ordered attribute grammars* (OAGs) introduced by Kastens et al. [Kas80]. OAGs are a subclass of Knuth-style AGs which allow for a statically established evaluation order of attributes. In the Synthesizer Generator the knowledge of evaluation order is used to incrementally evaluate attributes. The evaluation time of attributes affects the response time of an AG-based semantic editor. Incremental evaluation has the goal of allowing a more efficient evaluation of attributes where only attributes affected by a change are re-evaluated. Much research has been focused on trying to speed up incremental evaluation of attributes [JF85, HT86, Pec90].

In contrast to a statically determined attribute evaluation order, some research has been focused on *dynamic evaluation* of attributes. For example, Jourdan presents a demand-driven approach to attribute evaluation where attributes are evaluated when they are needed, allowing the full class of Knuth-style AGs to be evaluated [Jou84].

Extensions to AGs In order to facilitate modularity and increase expressiveness and extensibility, AGs have been extended with new concepts. Examples of extensions include Farrows introduction of *circular attribute grammars* (CAGs) [Far86] which support circular dependencies between attributes, not supported by Knuth-style AGs. CAGs provide means for defining, for example, data flow analysis which requires fix-point iteration. *Reference attribute grammars* (RAGs), as described by Hedin [Hed00] allow attributes to have references to other AST nodes as values. Similar extensions have been presented by Boyland [Boy05] and by Poetzsch-Heffter [PH97]. RAGs allow for modular and concise descriptions of, for example, name analysis where use nodes can point directly to their declaration using references. *Higher-order attribute grammars* (HAGs), described by Vogt et al. [VSK89], allow attributes to have attributed trees as values. HAGs provide a means to handle, for example, multi-pass compilation by step-wise refinement of a parse tree. *Forwarding*, by van Wyk et al. [WMBK02], allow for the creation of attributed subtrees, in a fashion similar to HAGs, with forwarding of attribute calls to the created node. Forwarding can, for example, be used to handle operator overloading. A similar notion called ReRAGs has been presented for RAGs by Ekman and Hedin [EH04]. *Collection attributes*, by Boyland [Boy96], allow attributes to have collections as values and can be used to, for example, define sets of uses for a declaration.

2.5 The JastAdd System

In our work, we use the JastAdd system which supports RAGs, as several other AG extensions like higher-order attributes, collection attributes and circular dependencies. The JastAdd system is implemented using object-oriented techniques and supports aspect-oriented programming in the form of inter-type declarations [KHH$^+$01]. Examples of other systems supporting a similar set of AG extensions include the Kiama library by Sloane et al. [SKV09], an attribute grammar implementation embedded as a library in the Scala language. The ASTER system by Kats et al. [KdJNNV09], supports the description of attribute evaluation by the use of *attribute decorators*, as an extension to the Stratego transformation language by Visser [Vis01]. The Silver system by Wyks et al. [vWBGK07], which has been used to implementation an extendable Java compiler, related to the JastAddJ compiler implemented using JastAdd.

3 Long-term Goals and Challenges

Our long-term goal is to develop general techniques allowing families of advanced high-quality semantic editors to be created easily by generating them from high-level specifications. This long-term goal includes the following subgoals:

G1 **High-quality**: Semantic editors need to be convenient to use with regard to *correctness* (of services), *robustness* (in handling of incomplete programs), and *performance* (with sufficient responsiveness).

G2 **Advanced**: Semantic editors should support services that make use of context-sensitive program information. For example, name completions, refactorings, bug detection and metrics.

G3 **Families**: Languages grow [SJ99] and there is a need for different variants. For example, the Java language has developed from version Java 1.4 to Java 5 (and now to Java 7), and the Modelica language has an extension (Optimica) focused on optimization. The technology needs to support different variants of a language while avoiding double maintenance.

We think that RAGs have the potential to serve as the underlying technology to reach these goals. But we see a number of challenges:

C1 **Reusable architecture** What is a suitable tool architecture? In particular, we want to allow reuse of generic components, support extensibility of the language, and extensibility of the semantic editing features supported.

C2 **Editing integration** How can different editing styles (text, syntax-directed, graphical) be integrated with the RAG-based in-memory model? For example, in textual editing, changes to the code need to be re-parsed and inserted into the model and related attributes need to be updated.

C3 **Performance** The semantic editor should be able to handle large programs and libraries with reasonable performance in terms of time and memory costs. Interesting problems here include both bringing down the attribute evaluation cost as such, and supporting incremental updating of the in-memory model.

To validate the technology it is interesting to apply it to different kinds of languages:

L1 Small experimental **domain-specific languages**. For example, used in course projects or in other small projects. This allows us to evaluate the learning curve of the tool.

L2 **General-purpose languages**, like Java. This allows us to compare our generated semantic editors with those existing in hand-coded IDEs.

L3 **Industrial special-purpose languages**, like Modelica, which need industrial strength tool quality due to real use in industry. This allows us to generate services otherwise too expensive to develop in a language community.

L4 **Bootstrapping** Bootstrapping is a good measure of maturity and provides an organized means for decreasing the dependencies on other tools.

4 Contributions

This thesis provides some contributions to our long-term goal and addresses some of the subgoals, challenges and language kinds identified in the previous section.

4.1 Extensible Flow Analysis

Semantic editors have the potential of providing advanced feedback to users provided that the needed information is contained in their in-memory models. Normally, some analyses like dead code analysis are performed later in the compiler-pipeline, beyond the use of the in-memory model. This makes dead code information unavailable. We show how this information can be added to the abstract syntax tree (AST), and thereby made available to users.

Our implementation supports intraprocedural control-flow and dataflow analysis and uses RAGs augmented with circular attributes and collection attributes. The control-flow implementation uses RAGs to super-impose control-flow graphs on top of the AST and the dataflow module extends this implementation. The implementation is described in concise and composable modules and provide extensible frameworks for further source code analyses and language extensions. We show how a code smell service like dead code analysis can modularly extend the flow analysis modules. The flow analysis modules have also been used by Schäfer et al. to implement refactorings [SEdM08, SVEdM09]. Both modules extend the JastAdd Extensible Java Compiler (JastAddJ), and we have applied the dead code analysis on normal-sized Java programs. The analysis performs well in comparison to other analysis frameworks with imperative implementations. More details of the flow analysis implementation are given in Paper I.

This work addresses the goal of sophisticated semantic editors (**G2**) and the challenge of composing a reusable architecture (**C1**). Sophistication, is addressed, in that we want to construct semantic editors with advanced semantic services like refactorings, and reusablility, is addressed, in that we want to modularly extend existing semantic language modules. The presented application, dead code analysis, is shown to work on Java which shows that we can handle a large general-purpose language (**L2**). In addition, the implementation modularly handles language extension from Java 1.4 to Java 1.5, further showing how we address the growth of languages (**G3**).

4.2 Structural Recovery of Text

In textual editing, the mapping between the textual representation and the semantic model depends heavily on robust parsing. However, certain errors like broken scopes are especially hard to handle and available error recovery techniques often fail.

We present a novel approach called *bridge parsing* for structural recovery of source files useful in textual semantic editing. In semantic editing, a working semantic model is needed even when the code is broken to such an extent that ordinary parser error recovery fails. The approach is shown to improve error recovery when set to work as a preprocessor to three parser generators; Beaver, ANTLR and LPG (earlier known as Jikes). The approach is tested on Java code. More details of the algorithm are given in Paper II.

Further, we show how bridge parsing can be combined with *scannerless generalized LR parsing* (SGLR) by Visser [Vis97] to provide improved error recovery. This approach builds on error recovery for SGLR presented in [KdJNNV09] inspired by bridge parsing. The combination is shown to be fruitful to such an extent that it outperforms the hand-crafted parser in the Eclipse JDT. More details of the combination are given in Paper III.

This work addresses the goal of high-quality semantic editors which are convenient to use (**G1**), the challenge of supporting a good editing integration (**C2**) and support for general-purpose languages (**L2**). It addresses conveniece in that the text editor needs to be robust and able to provide contextual semantic feedback even when the code is hard to parse, a common scenario during textual editing. It also supports a good editing integration by providing a light-weight and flexible error recovery strategy which works with different kinds of text parsers.

4.3 Automated Selective Attribute Caching

Even small improvements, with regard to evaluation time and memory usage, can affect the experienced convenience of using a semantic editor, since this performance can affect the responsiveness of the editor. One effective technique to reduce evaluation time of attributes in the underlying RAG-based AST is to cache attribute values, in order to avoid evaluation of the same attribute more than once. However, a large semantic editor specification may contain hundreds of attributes and it may be hard to manually decide which of these attributes to cache. Not the least, since they normally depend on each other in intricate ways. A simple solution is to cache all attributes (full caching). The opposite solution, to not cache any attribute, is normally not an option because this is very slow. Full caching provides reasonable performance, but a manual configuration can provide significant improvements both regarding evaluation time and memory usage. Unfortunately, a manual configuration requires deep knowledge of how attribute evaluation works and how attributes depend on each other.

We present a study of the attribute caching behavior in the JastAdd system, using the JastAddJ compiler for Java as a large use case. We describe a profiling-based technique for automatically finding a good caching configuration. The technique has been evaluated by compilation of Java programs from the DaCapo benchmark suite [BGH$^+$06]. Based on profiling of a single program in the suite we managed to provide a caching configuration capable of compiling other programs in the benchmark suite with a mean speed-up of 23.5% (in comparison with the execution time of a fully cached compiler). More details are given in Technical Report I.

This work addresses the goal and challenge of semantic editors with good performance (**G1**, **C3**) and support for general purpose languages (**L2**). A good cache configuration of attributes can significantly improve the performance of a RAG-based semantic editor, both regarding computation time and memory use.

4.4 An Outline of a RAG-based Semantic Editing Model

In developing a RAG-based semantic editor generator, the first step is to define how a RAG-based semantic editor and the framework around it should work – the *semantic editing model*. A semantic editing model needs to handle, for instance, synchronization of in-memory models. When a user makes a code modification the corresponding RAG-based model needs to change. Preferably, this update should render as small amount of work as possible since any delay in updating may have an effect on the responsiveness of the semantic editor.

We present an outline of a RAG-based editing model supported by our semantic editor generator – *JedGen* (JastAdd-based Semantic Editor Generator). The meta-tool is in its infancy and current work is focused on defining the semantic editing model. Still, the editing model has been used by Schäfer et al. in exploration of RAG-based refactorings [SVEdM09] for Java and in several master thesis projects at our department [Mat09, Mos09, Nil10], as well as in a graduate course on RAGs. More details of the semantic editing model are given in Technical Report II.

This work addresses all of the stated goals (**G1**, **G2** and **G3**) and, as a consequence, all of the stated challenges (**C1**, **C2** and **C3**). All the goals need to be addressed in order to design a stable foundation to stand on in aiming for these goals. This design process includes the consideration of elements like reusability and performance, stated as challenges. The JedGen prototype is being used to develop semantic editors for Java (**L2**), Modelica (**L3**) and small domain-specific languages in course projects (**L1**).

5 Future Work

This thesis work only takes some steps towards our long term goal of generating semantic editors from high-level specifications. There are several interesting ways to continue the work presented in this thesis:

5.1 Further Development of JedGen

In order for the JedGen tool to reach a state where it can be bootstrapped (L4), further development is needed. Currently, users of JedGen have plugged in manually to the semantic editing model described in Technical Report II. We would like to widen the scope of generation by generating the code needed to integrate with the Eclipse platform. Besides a wider generation scope we want to add support for, yet, unsupported services like refactorings (**G2**) and improve performance (**G1**).

5.2 Validation of JedGen

As a validation of the stated goals and to show that JedGen can support various kinds of languages (L1-L4) we would like to implement proof of concept tools. There are two interesting implementation projects already started. A semantic editor extension to the JastAddJ compiler in support of Java (**L2**), and a semantic editor extension to the JModelica compiler in support of Modelica (**L3**). The Java editor provides us with a chance to compare our editor with state-of-the-art editors like that provided in the Eclipse JDT. Both editors provide us with an opportunity to show how we can support language families (**G3**), through language extensions like Java 1.4 to Java 1.5, and language variants like Optimica for Modelica.

As a first step towards bootstrapping (**L4**), we would like to finish an already started bootstrapping of JastAdd on top of the JastAddJ compiler. The next step would be to define a semantic editor for JastAdd in JedGen, and eventually a semantic editor for JedGen in JedGen. Further, we would like to start the development of a semantic editor extension for a domain-specific language (**L1**). Plans are being made for a semantic editor supporting the QUPER model [RBSO08], a model for requirements engineering.

5.3 Refactorings

An interesting research direction is to work on supporting refactorings in JedGen, building on work on RAG-based refactorings by Schäfer et al. [SEdM08, SVEdM09]. An interesting study would be to use JedGen to define refactorings for Modelica, as extensions to the JModelica compiler (**L3**). An outline of how to describe the rename refactoring for JModelica has been presented by Hedin et al. [HNNÅ09]. This work would address the goal of providing advanced semantic editors (**G2**) and the challenge of constructing a reusable architecture (**C1**), since refactorings are added as extensions to existing language modules.

5.4 Incremental Attribute Evaluation of RAGs

Currently, during updating of subtrees in a RAG-based AST we need to re-evaluate *all* attributes in the AST since they may depend on information in the updated subtree. Preferably, we would like to update *only* those attributes affected by an update. An interesting research direction would be to study how to provide incremental evaluation to RAGs. Possibly this work can build on earlier work on incremental attribute evaluation, like that of Reps [RTD83], Boyland [Boy02] and

Hedin [Hed94]. This work focuses on a more fine-grained incremental updating on the attribute level, while we primarily are interested in a more coarse-grained updating on the subtree level. This work would address the goal of semantic editors with good performance (**G1**) and the challenge of providing semantic editors with good responsiveness (**C3**).

5.5 Visualization and Graphical Editing

Our focus so far has been on textual semantic editing with the goal of generating a semantic editor with similar features as the Eclipse JDT. Some visualization in the form of a content outline is currently provided by JedGen. An interesting direction would be to support graphical editing. A first step in this direction would be to provide more support for visualization of graphs. Our approach would be to define these graphs as super-imposed graphs on the AST using RAGs, in a fashion similar to how control-flow is super-imposed on the AST in the flow analysis modules presented in Paper I. Some work on providing program visualization using RAGs has been done by Magnusson and Hedin [MH00]. The second step would be to allow users to edit the graphical representation. This type of editing presents some challenges in how to define the semantics of change. For example, what happens when a user re-directs an arc, corresponding to an attribute value, to point to a different graphical component? If this is a pointer to a variable declaration from a variable use, this change may indicate that the variable use now has a different declaration. For validation we can provide a graphical editor extension to JModelica to show how we can support a language with both a textual and a graphical notation. This work would address the goal of providing advanced semantic editors (**G2**) and the challenge of supporting a good editing integration (**C2**).

BIBLIOGRAPHY

[ÅEH10] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Development of a Modelica compiler using JastAdd. *Science of Computer Programming*, 75:21–38, January 2010.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional, 2004.

[BBM03] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework.* Pearson Education, 2003.

[BBvB$^+$01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. The agile manifesto, 2001. `http://agilemanifesto.org/` [Accessed May 2010].

[BCD$^+$88] Patrick Borras, Dominique Clément, Th. Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and V. Pascual. CENTAUR: The system. In *Software Development Environments (SDE)*, pages 14–24, 1988.

[Ben86] Jon L. Bentley. Little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[Ber89] Jan A. Bergstra. *Algebraic specification.* ACM, New York, NY, USA, 1989.

[BGH$^+$06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel

Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM.

[BHN99] Elizabeth Bjarnason, Görel Hedin, and Klas Nilsson. Interactive language development for embedded systems. *Nordic Journal of Computing*, 6(1):36–54, 1999.

[Boe06] Barry Boehm. A view of 20th and 21st century software engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 12–29, New York, NY, USA, 2006. ACM.

[Boy96] John Tang Boyland. *Descriptional Composition of Compiler Components*. PhD thesis, University of California at Berkeley, 1996.

[Boy02] John Tang Boyland. Incremental evaluators for remote attribute grammars. In *Proceedings of the Second Workshop on Language Descriptions, Tools and Applications (LDTA 2002)*, volume 65 of *Electronic Notes in Theoretical Computer Science*, pages 9–29. Elsevier B.V., July 2002.

[Boy05] John Tang Boyland. Remote attribute grammars. *Journal of the ACM*, 52(4):627–687, 2005.

[Bro87] Frederick. P. Brooks. No Silver Bullet – Essence and Accicents of Software Engineering". *IEEE Computer*, 20:10 – 19, 1987.

[Bro95] Frederick P. Brooks. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[BS86] Rolf Bahlke and Gregor Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Langanguages and Systems (TOPLAS)*, 8(4):547–576, 1986.

[CFS$^+$09] Philippe Charles, Robert M. Fuhrer, Stanley M. Sutton, Jr., Evelyn Duesterwald, and Jurgen Vinju. Accelerating the creation of customized, language-specific ides in eclipse. *SIGPLAN Notices*, 44(10):191–206, 2009.

[Cor10] Oracle Corporation. NetBeans IDE, 2010. `http://netbeans.org/` [Accessed May 2010].

[DRW04] J. Des Rivières and J. Wiegand. Eclipse: a platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[EDMW90] Philip Elmer-DeWitt, Thomas McCarroll, and Paul A. Witteman. Time: Ghost in the machine, January 1990. `http://www.time.com/time/magazine/article/0,9171,969266,00.html` [Accessed May 2010].

[eff]

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In Martin Odersky, editor, *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 144–169. Springer, 2004.

[EH07a] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM.

[EH07b] Torbjörn Ekman and Görel Hedin. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69(1–3):14–26, December 2007.

[Far86] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 85–98, New York, NY, USA, 1986. ACM.

[Fou10a] The Eclipse Foundation. The Eclipse Java Development Tool (JDT), 2010. `http://www.eclipse.org/jdt/` [Accessed May 2010].

[Fou10b] The Eclipse Foundation. The Eclipse Platform, 2010. `http://www.eclipse.org/` [Accessed May 2010].

[Fow99] Martin Fowler. *Refactorings; Improving the design of existing code*. Addison Wesley Longman, Inc., Reading, Massachusetts, USA, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[HC01] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *IEEE Computer*, 34(9):120–122, 2001.

[Hed94] Görel Hedin. An overview of door attribute grammars. In Peter Fritzson, editor, *CC*, volume 786 of *Lecture Notes in Computer Science*, pages 31–51. Springer, 1994.

[Hed00] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.

[HHKR89] Jan Heering, P. R. H. Hendriks, Paul Klint, and J. Rekers. The syntax definition formalism SDF—reference manual—. *SIGPLAN Notices*, 24(11):43–75, 1989.

[HM88] Görel Hedin and Boris Magnusson. The Mjølner Environment: Direct interaction with abstractions. In Stein Gjessing and Kristen Nygaard, editors, *ECOOP*, volume 322 of *Lecture Notes in Computer Science*, pages 41–54. Springer, 1988.

[HN86] Nico A. Habermann and David Notkin. GANDALF: software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[HNNÅ09] Görel Hedin, Emma Nilsson-Nyman, and Johan Åkesson. A plan for building renaming support for Modelica. In *Electronic proceedings of the 3rd Workshop on Refactoring Tools (WRT'09)*. Hosted by University of Illinois, Department of Computer Science, 2009. Available online at `http://refactoring.info/WRT09` [May 2010].

[HT86] Roger Hoover and Tim Teitelbaum. Efficient incremental evaluation of aggregate values in attribute grammars. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 39–50, New York, NY, USA, 1986. ACM.

[HW09] Daqing Hou and Yuejiao Wang. An empirical analysis of the evolution of user-visible features in an integrated development environment. In *CASCON '09: Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 122–135, New York, NY, USA, 2009. ACM.

[IBN99] National Aeronautics Investigation Board and Space Administration (NASA). Mars climate orbiter mishap investigation board phase i report, 1999. `ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf` [Accessed May 2010].

[Jet10] JetBrains. IntelliJ IDEA, 2010. `http://www.jetbrains.com/idea/` [Accessed May 2010].

[JF85] Gregory F. Johnson and C. N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 141–151, New York, NY, USA, 1985. ACM.

[Jou84] Martin Jourdan. An optimal-time recursive evaluator for attribute grammars. In Manfred Paul and Bernard Robinet, editors, *Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 167–178. Springer, 1984.

[JT] The JastAdd Team. jastadd.org. `http://jastadd.org/` [Access May 2010.

[Kah87] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[Kas80] Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13(3):229–256, March 1980.

[KdJNNV09] Lennart C.L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco Visser. Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 445–464, New York, NY, USA, 2009. ACM.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[Kli93] Paul Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2):176–201, April 1993.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Journal Theory of Computing Systems*, 2(2):127–145, June 1968.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: Modular development of textual domain specific languages. In Richard F. Paige and Bertrand Meyer, editors, *TOOLS (46)*, volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315. Springer, 2008.

[KS98] Matthijs F. Kuiper and João Saraiva. Lrc - a generator for incremental language-oriented tools. In Kai Koskimies, editor, *CC*, volume 1383 of *Lecture Notes in Computer Science*, pages 298–301. Springer, 1998.

[Mat09] Jesper Mattsson. The JModelica IDE: Developing an IDE reusing a JastAdd compiler. Master's thesis, Lund University, Lund, Sweden, August 2009.

[Men10] Object Mentor. Junit.org, 2010. `http://www.junit.org/` [Accessed May 2010].

[MH00] Eva Magnusson and Görel Hedin. Program visualization using reference attributed grammars. *Nord. J. Comput.*, 7(2):67–86, 2000.

[Mos09] Erik Mossberg. Inspector – tool for interactive language development. Master's thesis, Lund University, Lund, Sweden, October 2009.

[Nil10] Philip Nilsson. Semantic editing compiler extensions using JastAdd. Master's thesis, Lund University, Lund, Sweden, June 2010. To be presented.

[Ode04] Martin Odersky. The Scala Experiment - can we provide better language support for component systems? In Wei-Ngan Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, page 364. Springer, 2004.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992. Available at: `ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdykethesis.ps.Z`.

[PC85] David L. Parnas and Paul C. Clements. A rational design process: How and why to fake it. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 2 of *Lecture Notes in Computer Science*, pages 80–100. Springer Berlin / Heidelberg, 1985.

[Pec90] Stephen B. Peckham. *Incremental attribute evaluation and multiple subtree replacements*. PhD thesis, Ithaca, NY, USA, 1990.

[Pet94] Mikael Pettersson. RML –a new language and implementation for natural semantics. 844:117–131, 1994.

[PH97] Arnd Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica*, 34(10):737–772, 1997.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, Aarhus, Denmark, 1981.

[RBSO08] Björn Regnell, Richard Berntsson-Svensson, and Thomas Olsson. Supporting roadmapping of quality requirements. *IEEE Software*, 25(2):42–47, 2008.

[RT84] Thomas Reps and Tim Teitelbaum. The Synthesizer Generator. In Peter B. Henderson, editor, *Proceedings of the ACM SIGSOFT-/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 19 of *SIGSOFT Software Engineering Notes*, pages 42–48, Pittsburgh, Pennsylvania, USA, May 1984. ACM.

[RTD83] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3):449–477, 1983.

[Sco72] Dana Scott. Mathematical concepts in programming language semantics. In *AFIPS '72 (Spring): Proceedings of the May 16-18, 1972, spring joint computer conference*, pages 225–234, New York, NY, USA, 1972. ACM.

[SDS$^+$10] Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. Correct Refactoring of Concurrent Java Code. In Theo D'Hondt, editor, *24th European Conference on Object-Oriented Programming (ECOOP '10)*, 2010.

[SEdM08] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for Java. In Gail E. Harris, editor, *OOPSLA*, pages 277–294. ACM, 2008.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[SH03] Jean E. Sammet and David Hemmendinger. Programming languages. In *Encyclopedia of Computer Science*, pages 1470–1475. John Wiley and Sons Ltd., Chichester, UK, 2003.

[SJ99] Guy L. Steele Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, October 1999.

[SKV09] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. A pure object-oriented embedding of attribute grammars. In T. Ekman and J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier B. V., 2009.

[Str66] Christopher Strachey. Towards a formal semantics. pages 198–216, 1966.

[SVEdM09] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping stones over the refactoring rubicon. In Sophia Drossopoulou, editor, *ECOOP*, volume 5653 of *Lecture Notes in Computer Science*, pages 369–393. Springer, 2009.

[The10a] The Eclipse Foundation. The Eclipse Modeling Project, 2010. `http://www.eclipse.org/modeling/` [Accessed May 2010].

[The10b] The Object Management Group (OMG). The Object Constraints Language (OCL), 2010. `http://www.omg.org/technology/documents/formal/ocl.htm` [Accessed May 2010].

[TM81] Warren Teitelman and Larry Masinter. The Interlisp Programming Environment. *IEEE Computer*, 14(4):25–33, 1981.

[TR81] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981.

[vDK97] Arie van Deursen and Paul Klint. Little languages: Little maintenance. In *Proceedings of the SIGPLAN Workshop on Domain-Specification Language*. ACM, 1997.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[VEM02] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 97, Washington, DC, USA, 2002. IEEE Computer Society.

[Vis97] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.

[Vis01] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *RTA '01: Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, pages 357–362, London, UK, 2001. Springer-Verlag.

[VSK89] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher-order attribute grammars. In *PLDI*, pages 131–145, 1989.

[vWBGK07] Eric van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. In *Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, Electronic Notes in Theoretical Computer Science. Elsevier B. V., 2007.

[Wei88] Gerald M. Weinberg. *The psychology of computer programming*. Van Nostrand Reinhold Co., New York, NY, USA, 1988.

[WMBK02] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in attribute grammars for modular language design. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 128–142, London, UK, 2002. Springer-Verlag.

INCLUDED PAPERS

Paper I

Extensible Intraprocedural Flow Analysis at the Abstract Syntax Tree Level

Abstract

We have implemented intraprocedural control-flow and dataflow analysis of Java source code in a declarative manner, using reference attribute grammars augmented with circular attributes and collection attributes. Our implementation is built on top of the JastAdd Extensible Java Compiler and we have run the analyses on normal-sized Java programs. We show how the analyses can be built using small concise composable modules, and how they provide extensible frameworks for further source code analyses and language extensions. Measurements indicate that our analyses perform well in comparison to other analysis frameworks with imperative implementations.

Emma Söderberg, Görel Hedin, Torbjörn Ekman, Eva Magnusson
Submitted to the journal of *Science of Computer Programming*, Elsevier 2010

This is an extended version of a paper with the title **Declarative Intraprocedural Flow Analysis of Java Source Code** Emma Nilsson-Nyman, Torbjörn Ekman, Görel Hedin, Eva Magnusson. In the proceedings of the *8th Workshop on Language Descriptions, Tools and Applications* (LDTA'08), Budapest, Hungary April 5th, 2008 Electronic Notes of Computer Science 238(5):155–171 ©2009 Elsevier B.V.

1 Introduction

Control-flow and dataflow analysis are key elements in many static analyses, and useful for a variety of purposes, e.g., code optimization, refactoring, enforcing coding conventions, bug detection, and metrics. Often, such analyses are carried out on a normalized intermediate code representation, rather than on the abstract syntax tree (AST). This simplifies the computations by not having to deal with the full source language. However, doing these analyses directly at the AST level can be beneficial, since the high-level abstractions are not compiled away during the translation to intermediate code. This is particularly important for tools that are integrated in interactive development environments, such as refactoring tools and tools supporting bug detection and coding convention violations.

In this paper, we present a new approach for computing intra-procedural control-flow and dataflow at the AST level. Our approach is declarative, making use of attribute grammars. Advantages include compact specification and modular support for language extensions, while giving sufficient performance for practical use.

To make the approach work, we rely on a number of extensions to Knuth's original attribute grammars [14]: *Reference attributes* [12] allow the control-flow edges to be represented as references between nodes in the AST. *Higher-order attributes* [22] are used for reifying entry and exit nodes in the control-flow graph as objects in the AST. *Circular attributes* [10,16] are used for writing down mutually recursive equations for dataflow as attributes, automatically solved through fixed-point iteration. Finally, *collection attributes* [5,15], enable the simple specification of reverse relations, for example, computing the set of predecessors, given the set of successors. These mechanisms are all supported in the JastAdd system [8], which we have used to implement our approach.

As a case study, we have implemented control-flow graphs and dataflow analysis for Java by extending JastAddJ (the JastAdd Extensible Java Compiler) [9]. Control-flow is implemented at the statement level, and includes abruptly completing control-flow like Java's exceptions. For dataflow, we have implemented both liveness analysis and reaching definition analysis. As an example of a tool-oriented analysis, we have implemented a detector of dead assignments to local variables.

The implementation is modular and extensible. Similar to the internal modularization of JastAddJ [9], each module can be viewed as an object-oriented framework, with a client API representing the result of the analysis, and an extension API for the attributes that need to be defined by a language extension module. In many cases, new language features can reuse the existing analyses as they are, but for language constructs affecting control-flow, rules need to be added. We exemplify this by considering the effect on the analyses when extending Java 1.4 to Java 5.

These are the main contributions of this paper:

- We show how to concisely specify control-flow graphs at the AST level, using reference and collection attributes. We show how the approach is applied to Java, including support for exceptions.
- We show how dataflow analyses can be specified on top of the control-flow, using circular attributes, resulting in specifications very similar to textbook definitions.
- We show how the analyses for Java 1.4 can be modularly extended to support Java 5 with a very small number of additional rules.
- We evaluate the performance of the approach by implementing a dead assignment analysis on top of the dataflow analyses, and running it on real world Java applications of up to 130 000 lines of code, showing that the approach is practical for real problems.

The rest of this paper is structured as follows. The implementation of control-flow analysis is described in Section 2, and the dataflow analysis in Section 3. An application doing dead code analysis is given in Section 4, and Section 5 discusses how to extend the analysis when the source language is extended. Section 6 provides a performance evaluation of our method. Finally, Section 7 discusses related work and Section 8 concludes the paper.

2 Control-flow Analysis

In control-flow analysis, the goal is to build a control-flow graph (CFG) where nodes represent blocks of executable code, and successor edges link the blocks in their possible order of execution. The nodes typically correspond to basic blocks, i.e., linear sequences of program instructions with one entry and one exit point [2]. Each node n has a set of immediate successors, $succ(n)$, and a set of immediate predecessors, $pred(n)$, both of which can be empty.

Different levels of granularity can be chosen for the blocks, e.g., statement level or expression level. In this paper, we do the analysis at the statement level, treating each source code statement as a block.

2.1 Control-flow API

In JastAdd, a program is represented as an AST, with nodes that are objects with attributes. To represent the CFG, we superimpose it on the AST, treating statement nodes as nodes in the CFG. We represent the $succ$ and $pred$ sets as attributes of statements. In some cases, we extend the AST with synthetic empty statements, to reify CFG nodes for which there is no natural node in the original AST. In particular, we add explicit entry and exit statements for each method.

```
public Set<Stmt> Stmt.succ();
public Set<Stmt> Stmt.pred();

public Stmt MethodDecl.entry();
public Stmt MethodDecl.exit();
```

Figure 1: The generated Java API for CFGs.

JastAdd builds on Java, and generates an ordinary Java API for the AST and its attributes. Figure 1 shows the generated Java API for CFGs. Here, **`Stmt`** and **`MethodDecl`** are AST classes. JastAdd specs can use this API to specify additional analyses, for example dataflow. The API can also be used by ordinary Java code, for example, an integrated development environment implemented in Java.

2.2 Language Structure

Figure 2 shows an example Java method and its corresponding AST. We will use this as a running example to illustrate how the control-flow graph is superimposed on the AST. To keep the example concise, we have omitted parameters and local declarations in the code.

```
void m() {
  if(c > 2)
    x = c;
  while(c < 10) {
    x += p();
    c++;
  }
}
```

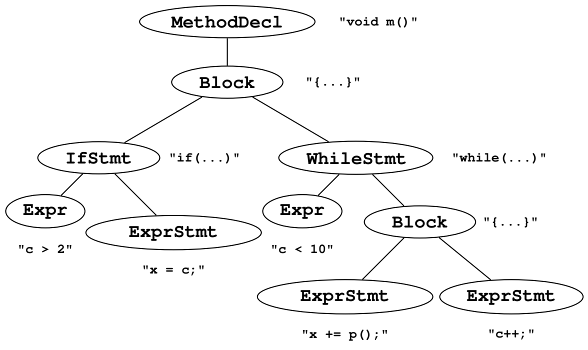


Figure 2: Sample Java method and its abstract syntax tree.

A simplified part of the abstract grammar for Java is shown in Figure 3. It is written in an object-oriented form with abstract classes **`Stmt`** and **`Expr`**, and subclasses for the individual statements and expressions such as **`WhileStmt`** and **`VarAccess`**.

The grammar uses a typical syntax with the Kleene star for list children, angle brackets for tokens, and square brackets for optional children. Children are either

```
MethodDecl ::= ParamDecl* Block;
ParamDecl  ::= <Type:String> <Name:String>;

abstract Stmt;
Block      : Stmt ::= Stmt*;
IfStmt     : Stmt ::= Expr Then:Stmt [Else:Stmt];
WhileStmt  : Stmt ::= Expr Stmt;
ExprStmt   : Stmt ::= Expr;
VarDecl    : Stmt ::= <Type:String> <Name:String> [Init:Expr];
ReturnStmt : Stmt ::= [Expr];
EmptyStmt  : Stmt;

abstract Expr;
AssignExpr   : Expr ::= LValue:Expr RValue:Expr;
VarAccess    : Expr ::= <Name:String>;
MethodAccess : Expr ::= <Name:String> Arg:Expr*;
```

Figure 3: Simplified parts of the Java abstract grammar in Figure 2.

named after their types, such as a `Block` child of a `MethodDecl`, or with given names preceding the typename. For example, the left and right children of an `AssignExpr` are named `LValue` and `RValue`.

Certain constructs in Java can act as both expressions and statements, for example assignments. They are represented as expressions in the grammar, for example `AssignExpr`, and the class `ExprStmt` serves the purpose of adapting such expressions to serve as statements. The full grammar for Java is available at the JastAdd web site [1].

2.3 The control-flow graph

Figure 4 shows how the AST has been attributed with successor edges and synthetic nodes, to form the CFG for the example method. The statement nodes constitute the nodes of the CFG, and reference attributes represent the successor edges. Two synthetic nodes are added to represent the entry and exit of the graph.

Some statements can be viewed as explicitly transferring control whereas other merely let the control flow through them. For example, the `IfStmt` in Figure 4 can transfer control to its `Then` branch, whereas the assignment inside the while loop merely transfers the control to the next statement, as determined by its enclosing block.

Based on this observation, we distinguish between the following three categories of statements.

```
void m() {
  if(c > 2)
    x = c;
  while(c < 10) {
    x += p();
    c++;
  }
}
```

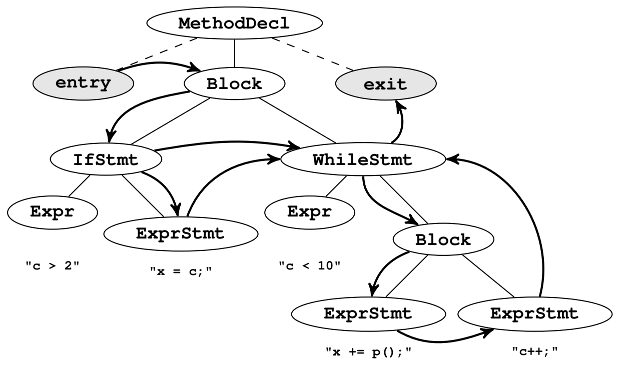


Figure 4: Example method and its CFG. Successors are shown as directed edges. Synthetic nodes are grey and the dashed lines show parent-child relations to these nodes.

Non-directing statement which merely transfers control to the next statement, as given by its context. `ExprStmt` is an example statement in this category.

Internal flow statement which may transfer control to and between its children. Examples of statements in this category are `Block`, `WhileStmt`, and `IfStmt`.

Abruptly completing statement which may transfer control to a non-local location, in effect abruptly completing an enclosing statement. Examples in Java include *break* statements, *throw* statements, and *return* statements.

In the following subsections, we will discuss how the different parts of the CFG are specified, and how these different categories of statements are handled.

2.4 The successors framework

Figure 5 shows a small attribution framework for the successor edges. It specifies the behavior for non-directing statements, and can be specialized to handle internal flow and abruptly completing statements. The framework introduces two attributes: `succ` and `following`. The `succ` attribute is a set of references to statement nodes, and represents the successor edges in the CFG. The `following` attribute of a statement s, is its set of successors as seen from its enclosing statement, i.e., without any knowledge of the internal flow or possible abruptly completing statements inside s. In the framework, `succ` is defined to be equal to `following`, thus capturing the behavior of non-directing statements. Subclasses of `Stmt` can override this definition to cater for internal flow or abrupt completion.

```
// The successor edges in the CFG
syn Set<Stmt> Stmt.succ();

// Statements that follow a statement, as seen from its context
inh Set<Stmt> Stmt.following();

// By default, they are the same.
eq Stmt.succ() = Stmt.following();
```

Figure 5: The attribution framework for successors.

The attribute `succ` is *synthesized*, whereas `following` is *inherited*[1]. The difference is that synthesized attributes must be defined in the node in which they are declared, whereas inherited attributes must be defined in an ancestor node. So, `succ` is defined by an equation in `Stmt`, and can have overriding equations in subclasses of `Stmt`, similar to ordinary virtual methods. The attribute `following` of a statement s, must instead be defined by one of the ancestor nodes of s. So to use this framework, equations must be provided that define the value of `following` for all possible statements.

As an example, consider the `Block` statement whose CFG specification is shown in Figure 6, specializing the successors framework. A `Block` is an example of an internal flow statement. To capture the internal flow, it overrides the definition of its own `succ` attribute, transferring control to its first internal statement, if there is one. Since a block has a list of statement children, it must also define the value of `following` for each of these children. This is done by the equation `Block.getStmt(int i).following = ...` which applies to the `i`:th statement child of a block. For the last child, `following` is simply the same as for the block itself. For other children, `following` contains the next child in the block. The function `singleton` used in this definition returns a set containing a single given reference.

In Figure 7 we illustrate how the framework is specialized for another internal flow statement: `IfStmt`. We assume that the code for the conditional expression is associated with the `IfStmt` node itself. Because `IfStmt` is an internal flow statement, it overrides its `succ` attribute. Control can be transferred to the `Then` part, and, depending on if there is an `Else` part or not, to the `Else` part or to the `following` statements. Note that it is not necessary to define the `following` attribute for the `Then` and `Else` parts, since they should have the same value as `following` for the `IfStmt` itself, so the same equation in some ancestor applies to these parts.

[1]Note that this use of the term *inherited* stems from Knuth [14] and is unrelated to and different from the object-oriented use of the term.

```
eq Block.succ() =
    (getNumStmt() = 0) // no children
    ? following()
    : singleton(getStmt(0));

eq Block.getStmt(int i).following() =
    (i = getNumStmt()-1) // last child
    ? following()
    : singleton(getStmt(i+1));
```

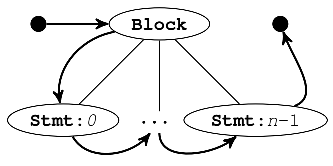


Figure 6: Specializing the successors framework for **`Block`**.

```
eq IfStmt.succ() =
    singleton(getThen()).union(
        hasElse()
            ? singleton(getElse())
            : following());
```

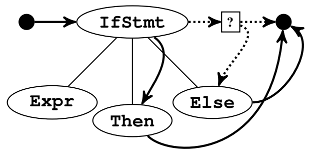


Figure 7: Specializing the successors framework for **`IfStmt`**.

To sum up, non-directing statements like **`ExprStmt`** can reuse the successors framework as is, and do not need to override or define any attributes. For internal-flow statements, the **`succ`** attribute needs to be overridden, and, depending on the behavior, the **`following`** attribute may need to be redefined for the constituent statements. Abruply completing statements also need to override the **`succ`** attribute, as will be discussed in later sections.

2.5 The entry and exit framework

To make sure there will always be well-defined entry and exit nodes, even for empty methods, we add two synthetic empty statements to each method. Nodes can be added declaratively to an AST by means of *higher-order attributes*, also known as *non-terminal attributes* (NTAs) [22]. An NTA is like a non-terminal in that it is a node in the AST. However, instead of being constructed as part of the initial AST, typically built by a parser, it is defined by an equation, just like an attribute. So in this sense, it is both an attribute and an AST node, hence the term higher-order. The right-hand side of an equation for an NTA must denote a *fresh* object, i.e. an object not already part of the AST, typically computed by a *new* expression.

```
syn nta Stmt MethodDecl.entry() = new EmptyStmt();
syn nta Stmt MethodDecl.exit() = new EmptyStmt();

eq MethodDecl.entry().following() =
    singleton(getBlock());
eq MethodDecl.getBlock().following() =
    singleton(exit());
eq MethodDecl.exit().following() =
    empty();
```

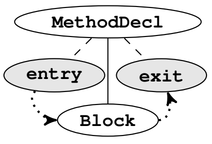


Figure 8: Attribution framework for entry and exit nodes. Dotted directed edges indicate elements in the **`following`** sets.

Figure 8 shows the attribution framework defining the entry and exit nodes. Since the method declaration is the parent of both the entry and exit nodes, as well as of the main block, it furthermore needs to define their **`following`** attributes. Naturally, the entry is followed by the main block, which is followed by the exit node, which in turn has no following statements, as specified in the equations. The function **`empty`**, used when defining **`following`** for the exit node, simply returns the empty set.

2.6 Handling separate conditions in branching statements

For if-statements, we treated the execution of the conditional expression as part of the if-statement itself. For some internal flow statements, like the **`DoStmt`**, the conditional expression needs to be represented separately from the host statement, since the execution does not start with this expression. We solve this by adding a synthetic statement as a placeholder for the expression, again using an NTA.

Figure 9 shows the use of this technique. A new empty statement is created as an NTA, and the **`succ`** attribute of the while-statement is overridden to transfer control to the placeholder. The **`following`** attributes for the NTA and the **`Stmt`** part are also defined, creating a cycle in the control flow.

2.7 Handling abruptly completing statements

The Java statements **`break`**, **`throw`**, **`continue`** and **`return`** transfer control to non-local locations, in effect abruptly completing the execution of an enclosing statement [11]. We will call these statements *abrupt* statements.

Normally, the successor of an abrupt statement is a so called *target* node. For example, the target of a **`return`** statement is the exit node. However, if the abrupt statement is inside the **`try`** block of a Java exception handler with a **`finally`** block,

```
syn nta Stmt DoStmt.cfgExpr() = new EmptyStmt();

eq DoStmt.succ() = singleton(getStmt());

eq DoStmt.getStmt().following() =
    singleton(cfgExpr());

eq DoStmt.cfgExpr().following() = following().union(getStmt());
```

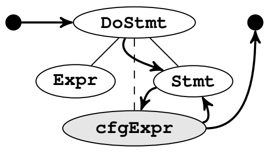


Figure 9: Specializing the successor framework for **`DoStmt`**, introducing a placeholder statement for the conditional expression.

the **`finally`** block will intercept control before transferring control to the target(s). Figure 10 shows an example.

```
try {
  return;
} finally {
  n();
}
```

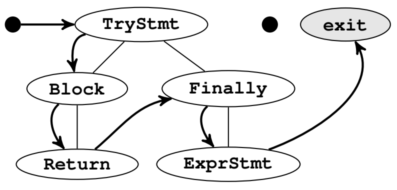


Figure 10: The control flow from a **`return`**, in the presence of a **`finally`** block.

In a similar way, the other abrupt statements also have a target to which control is normally transferred. For **`throw`** it is a matching **`catch`**, or the **`exit`** node. For **`break`** the target is the statement following a matching enclosing loop or labelled statement. For **`continue`** the target is the first part of a matching enclosing loop. Figure 11 shows the control-flow in three simplified examples, without **`finally`** blocks.

As an example of how to handle abrupt statements, we will take a closer look at **`BreakStmt`**. The other abrupt statements are handled in an analogous way. We introduce an inherited attribute **`breakTarget`**, returning a singleton set with the matching target, or the empty set if no target is found (corresponding to a compile-time error). For the **`BreakStmt`**, this attribute will be the true successor, i.e., either the normal target (e.g., a while loop), or a **`finally`** block, in case the normal target is outside of an enclosing **`TryStmt`** with a **`finally`** block.

The attribute **`breakTarget`** is also defined for **`TryStmt`**. Here, the value is the target outside the **`TryStmt`**, i.e., ignoring its internal **`finally`** block, typically the

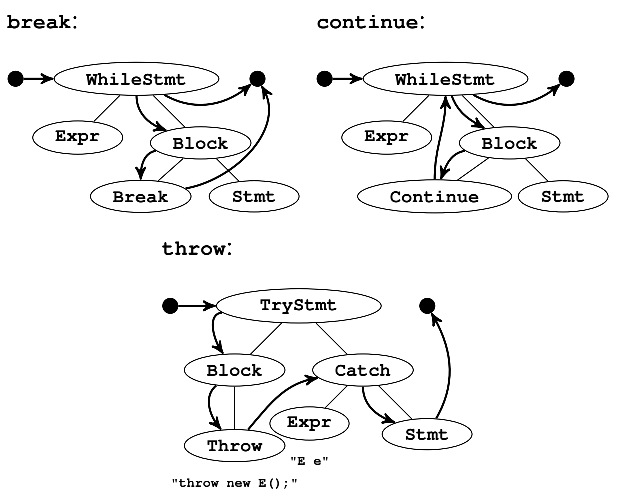


Figure 11: Control flow for some abrupt statements.

normal break target. By accessing this attribute, the **`finally`** block can find its successor, i.e., the normal break target. This solution works also for nested **`TryStmt`**s with **`finally`** blocks, in which case control is transferred from the **`break`**, statement, through all the **`finally`** blocks of enclosing **`try`** statements, and finally to the normal target.

The **`breakTarget`** attribute is parameterized by the **`BreakStmt`** to allow the target for the correct **`BreakStmt`** to be found. This attribution solution, using parameterized inherited attributes, is similar to the JastAdd implementation of Java name analysis, as presented in [7].

The successor of a **`BreakStmt`** is now simply defined as the **`breakTarget`** of itself. Figure 12 shows the specification. There are several equations defining **`breakTarget`**, and if there is more than one in a chain of ancestors, the closest equation applies. Therefore, if a **`BreakStmt`** is enclosed by a **`TryStmt`**, and then by a **`BranchTargetStmt`** (e.g., a while loop), the equation in the **`TryStmt`** will hold. If the **`BreakStmt`** is not enclosed by any of these kinds of statements, the equation defined in **`BodyDecl`** will hold, defining the target to be the empty set. To illustrate how this works, consider Figure 13, showing the values of **`breakTarget`** for an example program.

To handle the other three abrupt statements we define three more attributes – **`continueTarget`**, **`returnTarget`** and **`throwTarget`**, and use them in a similar fashion. With this approach we end up with potentially several abrupt statements transferring control to the **`finally`** block. The potential successors of the **`finally`** block is thus the set of normal targets for all these intercepted abrupt statements.

```
eq BreakStmt.succ() = breakTarget(this);

inh Set BreakStmt.breakTarget(BreakStmt stmt);
inh Set TryStmt.breakTarget(BreakStmt stmt);

// Equations for breakTarget
eq BodyDecl.getChild().breakTarget(BreakStmt stmt) = empty();
eq BranchTargetStmt.getChild().breakTarget(BreakStmt stmt) =
    targetOf(stmt)
    ? following()
    : breakTarget(stmt);
eq TryStmt.getBlock().breakTarget(BreakStmt stmt) =
    hasFinally()
    ? singleton(getFinally())
    : breakTarget(stmt);
```

Figure 12: Specializing the successor framework for `BreakStmt`. The `targetOf` attribute is defined in the fontend.

For this reason, we introduce an attribute `interceptedAbruptStmts` which contains references to these statements. Given this attribute, the `TryStmt` can define the `following` attribute for its `finally` block, as shown in Figure 14. Here, the attribute `targetAt` uses the double dispatch pattern [13] to let each kind of abrupt statement decide how to compute its target[2].

Handling unchecked exceptions

In addition to explicitly thrown exceptions, using the `throw` statement, exceptions can be thrown implicitly by the runtime system at more or less every statement in the program. Examples include null pointer exception, division by zero, out of memory, etc. So in this sense, more or less every statement can have abrupt completion. Instead of adding explicit successor edges for all these possible control paths, we define an inherited attribute `uncheckedExceptionTarget` for `Stmt` nodes, and in that way make all nodes aware of these potential successors. By default, this attribute is a set containing the exit node. But if there are `catch` clauses that match `RuntimeException` or `Error`, these clauses are also added.

This approach is inspired by the factored control-flow graph explained in [6] where unchecked exception branches are summarized at the end of basic blocks to limit the number of branches.

[2]The equation for `following` uses an assignment and a for loop which might be surprising since our approach is declarative. However, because we use Java method body syntax to define attribute values, it is natural to use imperative code here. This is perfectly in agreement with the declarative approach as long as that code has no net side effects, i.e., only local variables are modified.

```
{
    while (..) {
        try {
            break;
        } finally {
            n();
        }
    }
    m();
}
```

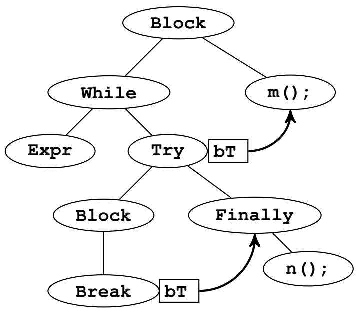


Figure 13: Values of the `breakTarget` attribute (`bT`).

2.8 Predecessors

To complete the implementation of the control-flow API, we now define the set of predecessors. This is simply the inverse of the successors relation, so if there is a successor edge from a to b, there will be a predecessor edge from b to a. Such inverse relations are easily defined using *collection* attributes [5,15]. The attributes we have seen so far have been defined using an equation located in an AST node. A collection, in contrast, is an attribute whose value is defined by the combination of a number of *contributions*, distributed over the AST. This way, we can define the predecessor sets by letting each statement contribute itself to the predecessor sets of its successors. Figure 15 shows the JastAdd specification.

In the computation of the collection, it will be initialized to **`empty()`**, and then all the contributions will be added, using the method `add`. It is assumed that `add` is commutative, i.e., that the order of adding the contributions is irrelevant. A more detailed presentation of collection attributes and their evaluation in JastAdd is available in [15].

3 Dataflow Analysis

We want to analyze dataflow on the control-flow graph defined in the previous section. Two typical examples of dataflow analyses are liveness analysis and reaching definition. We describe our implementation of these analyses using JastAdd in the following two subsections.

3.1 Liveness analysis

A variable is *live* at a certain point in the program, if its assigned value will be used by successors in the control-flow graph. If a variable is assigned a new value

```
eq TryStmt.getFinally().following() {
    Set flw =
      (getFinally().canCompleteNormally())
      ? following()
      : empty();
    for (Stmt abrupt : interceptedAbruptStmts) {
      flw = flw.union(abrupt.targetAt(this));
    }
    return flw;
}

syn Set Stmt.targetAt(TryStmt t) = empty();
eq BreakStmt.targetAt(TryStmt t) = t.breakTarget(this);
eq ContinueStmt.targetAt(TryStmt t) = t.continueTarget(this);
...
```

Figure 14: Specializing the successor framework for `TryStmt`.

```
coll Set Stmt.pred() [empty()] with add;
Stmt contributes this to Stmt.pred() for each succ();
```

Figure 15: Using a collection attribute to define the predecessors.

before an old value has been used, the old assignment to the variable is unnecessary – *dead*, unless the assignment has side-effects.

We express liveness in the same fashion as Appel in [3] using four sets – in, out, def and use. The def set of a statement s contains the variables assigned values in s, and the use set contains the variables whose values are used in s. From these two sets we calculate the in and out sets, i.e., variables live into a statement and variables live out of a statement, using the following equations:

Definition 1 *Let n be a statement node and $succ[n]$ the value of the* `succ` *attribute for the node n:*

$$\begin{aligned} in[n] &= use[n] \cup (out[n] \setminus def[n]) \\ out[n] &= \bigcup_{s \in succ[n]} in[s] \end{aligned}$$

We note that the equations for the in and the out sets are recursive and mutually dependent, i.e. they have a circular dependency on each other. Equations like these are usually solved by iteration until a fixpoint is reached, which is guaranteed if all intermediate values can be organized in a finite height lattice and all operations are monotonic on that lattice. We will explain how circular equations like these can be implemented as circular attributes in JastAdd [16].

The *use* and *def* sets

The main challenge in computing the *use* set for each statement, is to support all kinds of statements and expressions in the source language. A complex language such as Java has more than 20 statements and 50 expressions. Fortunately, it is quite easy to support all these constructs in JastAddJ (the JastAdd Extensible Java Compiler), since each expression that accesses a local variable encapsulates a `VarAccess` node performing the actual binding. Moreover, each `VarAccess` node has two `boolean` attributes, `isDest` and `isSource`, determining whether the access acts as a definition (*l-value*) or use (*r-value*). Some nodes actually act as both. For example, a `VarAccess` that is the child of the post increment operator '`++`', will both read from and write to the variable. JastAddJ also defines an attribute `decl` for `VarAccess` nodes, referring to the appropriate declaration node. Figure 16 summarizes the JastAdd API used.

```
public boolean VarAccess.isDest();
public boolean VarAccess.isSource();
public Decl VarAccess.decl();
```

Figure 16: JastAddJ API used by liveness analysis

In the liveness analysis, we represent *use* and *def* as sets of references to declaration nodes in the AST. We implement them using collection attributes, letting `VarAccess` nodes contribute their declaration to the appropriate collection of their enclosing statement, depending on their role as an *r-value* and/or *l-value*. The variable and parameter declarations are also viewed as assignments, so they contribute themselves to their own *def* set. Note that even if a variable declaration does not have an init clause, it is regarded as an assignment, since when the variable comes into existence, it will be assigned some value by the runtime system, even if it is not defined by the language what that value is. Figure 17 shows the implementation of these attributes. A helper attribute `enclosingCFGStmt` is also included, giving all expressions access to their closest enclosing statement.

These three attributes effectively compute the *use* and *def* sets for all statements in Java. Consider for instance a `MethodAccess` with the structure described in Figure 3. Its arguments may very well contain uses and definitions, since both are expressions in Java. However, we need not provide any additional equations for that language construct since contributions from each `VarAccess` are collected automatically.

```
// closest enclosing statement
inh Stmt Expr.enclosingCFGStmt();
eq Stmt.getChild().enclosingCFGStmt() = this;

// def
coll Set<Decl> Stmt.def() [empty()] with add;
VarAccess contributes decl() when isDest()
    to Stmt.def() for enclosingCFGStmt();
VarDecl contributes this to Stmt.def() for this;
ParamDecl contributes this
    to Stmt.def() for enclosingCFGStmt();

// use
coll Set<Decl> Stmt.use() [empty()] with add;
VarAccess contributes decl() when isSource()
    to Stmt.use() for enclosingCFGStmt();
```

Figure 17: Implementation of def and use for liveness analysis

These abstractions are also important from an extension point of view. If we add a new language construct that modifies a local variable we need only make sure it encapsulates a **`VarAccess`** and provide equations for the inherited attributes **`isDest`** and **`isSource`**, which are needed elsewhere in the frontend anyway, and the use set and def set attributes are still valid.

The in and out sets for liveness

The equations for the in set and out set in Definition 1 are mutually dependent. As mentioned earlier, such equations can be solved by iteration as long as the values form a finite height lattice and all functions are monotonic. This is clearly the case for our equations since the power set of the set of local variables ordered by inclusion forms a finite lattice, with the empty set as bottom, on which union is monotonic. A fixpoint will thus be reached if we start with the bottom value and iteratively apply the equations as assignments until no values change.

JastAdd has explicit support for fixpoint iteration through circular attributes, as described in [16]. If we declare an attribute as circular and provide a bottom value, then the attribute evaluator will perform the fixpoint computation automatically. This allows us to implement the in and out sets using circular attributes, resulting in a specification very close to the textbook definition, as shown in Figure 18.

In our actual implementation, we use an even more concise specification of the out set by defining it as a collection attribute, reversing the direction of the computation by making use of the predecessors instead of the successors. See Figure 19.

```
// in
syn Set<Decl> Stmt.liveness_in() circular [empty()] =
    use().union(liveness_out().compl(def()));

// out
syn Set<Decl> Stmt.liveness_out() circular [empty()] {
    Set<Decl> set = empty();
    for(Stmt s : succ()) {
        set = set.union(s.liveness_in());
    }
    return set;
}
```

Figure 18: Implementation of liveness *in* and *out* sets, using circular attributes.

```
coll Set<Decl> Stmt.liveness_out() circular [empty()] with add;
Stmt contributes liveness_in() to Stmt.liveness_out()
    for each pred();
```

Figure 19: Alternative implementation of the *out* set, using a circular collection.

An alternative to using circular attributes would be to manually implement the fixpoint computation imperatively. Such a solution requires manual book keeping to keep track of change, which significantly increases the size of the implementation and the essence of the algorithm gets tangled with book keeping code. Also, it is necessary to either statically approximate the sets of attributes involved in the cycle to iterate over, or to manually keep track of such dependencies dynamically. This is all taken care of automatically by the attribute evaluation engine in JastAdd when using circular attributes.

3.2 Reaching definitions

In computing *reaching definitions*, we are interested in sets of *definitions* (assignments), rather than in sets of variable declarations. Because definitions may occur in several different syntactic constructs, not just in assignment statements, we define an interface `Definition` to abstract over the relevant AST classes, namely `VarAccess`, `VarDecl`, and `ParamDecl`. Not all variable accesses are definitions, but the `isDest` attribute can be used to decide this.

A definition of a variable is said to *reach* a use of a variable if there is a path in the control-flow graph from the definition to the use. A variable use may be reached by more than one variable definition in which case the actual value of the variable can not be decided statically. For cases where there is only one reaching definition the use might be replacable with a constant, a property typically used in, for example, constant propagation.

We define five sets – defs, gen, kill, in and out, in the same fashion as Appel [3]. The defs set of a variable declaration v contains all definitions of that variable. The gen set of a statement s contains the definitions in s, i.e., corresponding to the new variable values generated by that statement. The kill set of a statement s is the set of definitions killed by definitions made in s. Consider a definition d of a certain variable v. The kill set for a definition d is the defs for v, minus the definition d itself, see Definition 2. The kill set for a statement is simply the union of the kill sets of its gen set.

The in set of a statement s is the set of definitions that reach the beginning of s, and out is the set that reaches the end of s. Given the kill and gen sets, in and out are defined as shown in Definition 3. Note that the equations for in and out are recursive and mutually dependent, hence requiring a fixpoint iteration for evaluation.

Definition 2 *Let d be a definition of a variable v:*

$$d : v \leftarrow \ldots : \mathit{kill}[d] = \mathit{defs}[v] \setminus \{d\}$$

Definition 3 *Let n be a statement node and $pred[n]$ the value of the* **`pred`** *attribute for the node n:*

$$\begin{aligned} \mathit{in}[n] &= \bigcup_{p \in pred[n]} \mathit{out}[p] \\ \mathit{out}[n] &= \mathit{gen}[n] \cup (\mathit{in}[n] \setminus \mathit{kill}[n]) \end{aligned}$$

The defs set

To implement the defs set, we use a collection attribute on **`Variable`**, which is an interface implemented by **`VarDecl`** and **`ParamDecl`**. We then let the definitions contribute themselves to their declaration. The contributing **`VarAccess`** nodes check that they are actually acting as definitions, using the attribute **`isDest`**. Since the analysis is intraprocedural, they also check that they define local variables or parameters, rather than, for example, fields. The implementation is shown in Figure 20.

The gen and kill sets

The gen set for a statement contains all the definitions inside the statement. Again, we use a collection attribute to implement this set. **`VarAccess`** nodes that serve as definitions, contribute themselves to the gen set of their enclosing statement. Variable declarations contribute themselves to their own gen. Parameter declarations,

```
coll Set<Definition> Variable.defs() [empty()] with add;
VarAccess contributes this
    when isDest() && decl().isLocalOrParameter()
    to Variable.defs() for decl();
VarDecl contributes this
    to Variable.defs() for this;
ParDecl contributes this
    to Variable.defs() for this;
```

Figure 20: Implementation of $defs$ using attributes.

finally, are considered as assignments performed when entering the method, and are therefore contributed to the gen of the entry statement, using a helper attribute `entry`. The $kill$ set is implemented using the same strategy, see Figure 21.

The in and out sets for reaching definitions

In Definition 3 the sets in and out are defined as two mutally dependent equations using the $kill$ and gen sets. Again we use circular attributes, obtaining an implementation very similar to the textbook definition of these sets. See Figure 22.

4 Dead Code Analysis

To evaluate the efficiency and scalability of our approach, we have implemented a simple intraprocedural analysis for Java which detects dead code in the shape of dead assignments. We locate assignments whose values are not used later in a method, and variable declarations that are not used at all. This analysis can easily be added as an extension to the dataflow analyses described in Section 3. We try out three versions – a basic analysis, locating unused local variables, one based on liveness analysis, and one combining liveness analysis with reaching definitions analysis. In all versions we say that a statement is live if it has side effects, namely if there is

- .. a write to a non-local variable
- .. a write to an array element
- .. a method or constructor call
- .. a returned value
- .. a thrown value

```
// helper attribute entry
inh Stmt ParamDecl.entry();
eq MethodDecl.getParamDecl(int i).entry() = entry();

// gen
coll Set<Definition> Stmt.gen() [empty()] with add;
VarAccess contributes this when isDest()
    to Stmt.gen() for enclosingCFGStmt();
VarDecl contributes this
    to Stmt.gen() for this;
ParamDecl contributes this
    to Stmt.gen() for entry();

// kill
coll Set<Definition> Stmt.kill() [empty()] with add;
VarAccess contributes decl().defs().compl(this)
    when isDest() && decl().isLocalOrParameter()
    to Stmt.kill() for enclosingCFGStmt();
VarDecl contributes defs().compl(this)
    to Stmt.kill() for this;
ParamDecl contributes defs().compl(this)
    to Stmt.kill() for entry();
```

Figure 21: Implementation of *gen* and *kill* .

To collect all dead assignments of a compilation unit, we add a collection (`coll`) attribute `deadAssignments` to the `CompilationUnit` class. This class represents a file with one or more classes which might contain one or more methods:

```
coll Set<Stmt> CompilationUnit.deadAssignments()
    [empty()] with add;
```

The `CompilationUnit` class is connected to the grammar in Figure 3 as follows:

```
CompilationUnit ::= ClassDecl*;
ClassDecl       ::= MethodDecl*;
MethodDecl      ::= ...
```

Dead assignments contribute themselves to the collection of their enclosing `CompilationUnit` using a `contributes` clause. The reference to the `CompilationUnit` node is propagated to descending statement nodes using an inherited attribute `enclosingCompilationUnit`:

```
ExprStmt contributes this when isDeadAssign()
    to CompilationUnit.deadAssignments()
```

```
// out
syn Set<Definition> Stmt.reaching_out() circular [empty()] =
    gen().union(reaching_in().compl(kill()));

// in
coll Set<Definition> Stmt.reaching_in()
    circular [empty().mutable()] with add;
Stmt contributes reaching_out()
    to Stmt.reaching_in() for each succ();
```

Figure 22: Implementation of the *in* and *out* sets for reaching definitions.

```
    for enclosingCompilationUnit();
VarDecl contributes this when isDeadAssign()
    to CompilationUnit.deadAssignments()
    for enclosingCompilationUnit();
```

Each of these statements – **`ExprStmt`** and **`VarDecl`** contribute to the collection if their **`isDeadAssign`** attribute is true. We define this attribute to be false by default for all statements:

```
syn boolean Stmt.isDeadAssign() = false;
```

4.1 Version I: Dead uses

In our first version, we consider variable declarations, i.e., "`int i =0;`" or only "`int i;`" for cases where we know that `i` is unused or all uses of `i` are in dead assignments. We define that variable declarations are dead when there are no side effects and all uses are dead. We define this by adding an equation for the **`VarDecl`** node:

```
eq VarDecl.isDeadAssign() =
    !mayHaveSideEffects() && allUsesAreDead().isEmpty();
```

The latter attribute **`allUsesAreDead`**, in the above equation, iterates over a set given by another attribute, **`uses`**, defined as follows:

```
coll Set Variable.uses() [empty()] with add;
VarAccess contributes this to Variable.uses() for decl();
```

A use is defined to be dead if its enclosing statement is a dead assignment:

```
syn boolean VarAccess.inDeadAssign();
```

```
eq VarAccess.inDeadAssign() = enclosingCFGStmt().isDeadAssign();
```

4.2 Version II: Liveness analysis

The second version also takes liveness analysis into account.

Definition 4 *If a variable is defined in a statement, but not live immediately after the statement, the statement is considered dead in the sense that the assignment is unnecessary. That is, a statement s is dead when:*

$$kill[s] \neq \emptyset \wedge kill[s] \cap out[s] = \emptyset$$

The result of the liveness analysis can provide us with a very useful attribute **`isDead`**, based on the equation in Definition 4:

```
syn boolean Stmt.isDead() {
    Set res = def().compl(liveness_out());
    return !res.isEmpty() && res.equals(def());
}
```

We continue to define that expression statements, i.e. statements like “`a = 0;`“ or “`a++;`“, are dead assignments when there are no side effects and the liveness analysis attribute **`isDead`** is true:

```
syn boolean Stmt.isDeadAssign() = false;
eq ExprStmt.isDeadAssign()= !mayHaveSideEffects() && isDead();
```

We keep the earlier equation for **`VarDecl`** nodes, hence also collecting unused variables as well as dead assignments.

4.3 Version III: Liveness analysis and reaching definition

In the third version, we add combine liveness analysis with reaching definitions analysis. We implement this by adding a condition to the equation for **`isDeadAssign`** for expression statements, defining that for cases when there are no side effects and the **`isDead`** attribute is true the statement might still be a dead assignment. Considering cases like:

```
a = 0;
b = a; // b is dead
```

where the first assignment is dead as a consequence of the fact that the second statement is dead. We modify the previous definition as follows:

```
eq ExprStmt.isDeadAssign()
    = !mayHaveSideEffects() && (isDead() ||
    (hasAssigns() && allReachedUsesAreDead()));
```

This says that an assignment can be dead if all reached uses are dead. This is where we make use of the reaching definition implementation. The **allReachedUses-AreDead** attribute investigates whether all reached uses are dead using an attribute **reachedUses** defined on **Stmt** which returns a set of reached uses. We let uses, that is **VarAccess** nodes, implement an interface **ReachedUse**:

```
syn boolean Stmt.allReachedUsesAreDead() circular [false];
eq Stmt.allReachedUsesAreDead() {
    for (ReachedUse use : reachedUses())
        if (!use.inDeadAssign())
            return false;
    return true;
}
```

Here we create a circular dependency for the attribute **isDeadAssign**. The **is-DeadAssign** attribute depends on the **allReachedUsesAreDead** attribute which depends on the **isDead** attribute. The **isDead** attribute depends on **isDeadAssign** for **VarAccess**. We handle this by declaring that the **isDeadAssign** attribute is **circular**, that is:

```
syn boolean Stmt.isDeadAssign() circular [false];
```

5 Language Extensions

The previous examples have illustrated how the control-flow specification for individual statements can be written modularly. Similarly, the control-flow implementation for Java 1.4 can be extended modularly to support Java 1.5. The only new language construct that affects the CFG is the new enhanced **for** statement. This is an internal-flow statement with the following abstract syntax:

```
EnhancedFor : BranchTargetStmt ::= VarDecl Expr Stmt;
```

This statement iterates over the elements in the iterable object denoted by **Expr**. In each iteration, a new element is assigned to **VarDecl**, and the **Stmt** is executed. To capture this flow, we let the **EnhancedFor** itself represent the initialization of the iterator and then we create a control-flow placeholder node, using an NTA, to represent the "hasNext" check of the iterator. We provide equations defining the **succ** attribute for **EnhancedFor** and the **following** attributes of its constituents. Figure 23 shows the specification.

Note that since the analyses of liveness, reaching definitions, and dead assignments are defined in terms of the control-flow graph, they will work automatically also for the new **EnhancedFor** construct.

```
syn nta Stmt EnhancedFor.cfgNext() = new EmptyStmt();
eq EnhancedFor.succ() = singleton(cfgNext());

eq EnhancedFor.cfgNext().following() =
    following().union(getVarDecl());
eq EnhancedFor.getVarDecl().following()=
    singleton(getStmt());
eq EnhancedFor.getStmt().following()=
    singleton(cfgNext());
```

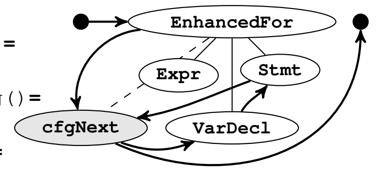


Figure 23: Control flow for **`EnhancedFor`**

Name	Version	Lines of Code	Candidates	# Flows	Avg. |Flow|
ANTLR	2.7.7	37 730	4 253	3 332	14.0
Bloat	1.0	38 581	5 771	5 095	9.0
JCharts	1.0	9 968	1 924	1 469	7.0
FOP	0.95	130 300	18 662	19 632	7.0

Figure 24: Applications used for evaluation. Candidates are the number of local variable declarations and assignments in an application. The last two columns show the number of intraprocedural flows in an application and the average size, with regards to the number of statements in the flow, of these flows.

6 Evaluation

To evaluate our implementations we used a set of Java applications, listed in Figure 24; **ANTLR** is a parser and translator generator, **Bloat** is a byte-code level optimization and analysis tool, **JCharts** is a charting utility tool and **Apache FOP** is a print formatting tool. We calculated the number of dead assignment *candidates*, that is, assignments and declarations of local variables. For each application we also counted the number of lines of code (excluding blank lines and comments) and the number of intraprocedural *flows*, i.e., the number of methods, constructors and similar constructs with local control flow. We also report the average size of these flows (average number of statements in the control-flow graphs). As might be expected, there is a correlation between the number of candidates, flows and lines of code, as shown in Figure 24. We note that one of the applications has substantially longer flows, with an average of 14 statements per flow.

Application	**Version I** *unused variables*	**Version II** *liveness*	**Version III** *liveness+reaching definition*
ANTLR	163	265	265
Bloat	10	20	20
JCharts	5	9	14
FOP	5	20	20

Figure 25: Dead Assignments (#) for each application and analysis version.

6.1 Analysis results

Figure 25 shows the results from our analyses, measuring unused variables (version I), dead assignments based on liveness analysis (version II), and dead assignments based on a combination of liveness and reaching definitions analysis (version III).

We find dead candidates in all of the applications, especially in the one with long flows, as shown in Figure 24. Larger methods make it harder to get an overview of the logic of the method, making it harder to spot unnecessary assignments or declarations.

There seems to be no direct correlation between the number of dead candidates and the number of lines of code. This indicates that this metric depends more on code quality or maturity rather than code size.

We note that versions II and III differ for only one of the applications, and there with only 5 occurrences of dead assignments. I.e., the reaching definition analysis does not contribute much in comparison to liveness. These additional occurrences are due to a ripple effect that the reaching analysis can detect, when one or more assignments depend on a dead assignment:

```
a = b; // also dead via dependency to b
b = c; // also dead via dependency to c
c = 0; // dead
```

Another thing to mention is that a number of the found dead assignments are variables that are assigned `null`. There may be cases where such an assignment can have side-effects on memory management. We still include these assignments in the analysis, since we only provide suggestions to users and there are no actual eliminations performed by the analyses.

To validate the results of our analyses we ran Soot, an optimizing framework for bytecode [21], on the same Java applications. Soot performs optimizations on different kinds of intermediate code representations. It takes both Java class files and source files as input. For Java input it uses the frontend from JastAddJ for parsing and semantic checks before translation to intermediate code. Soot can

Application	**Plain**	**Version I** *unused variables*	**Version II** *liveness*	**Version III** *liveness + reaching def.*
ANTLR	1.9 ± 0.1 s 100 ± 2 kb	2.4 ± 0.1 s 111 ± 2 kb	6.1 ± 0.1 s 160 ± 4 kb	12.8 ± 1.4 s 184 ± 5 kb
Bloat	3.1 ± 0.1 s 85 ± 0 kb	4.0 ± 0.7 s 92 ± 1 kb	5.0 ± 0.6 s 142 ± 1 kb	12.4 ± 1.4 s 193 ± 8 kb
JCharts	1.7 ± 0.3 s 67 ± 2 kb	2.7 ± 0.3 s 70 ± 1 kb	3.3 ± 0.3 s 83 ± 1 kb	3.8 ± 0.2 s 116 ± 17 kb
FOP	22.8 ± 0.2 s 370 ± 1 kb	23.1 ± 4.2 s 379 ± 0 kb	29.8 ± 0.4 s 418 ± 1 kb	34.8 ± 0.5 s 428 ± 6 kb

Figure 26: Execution time and memory usage for JastAddJ. All execution times are averages over 20 runs on a pre-heated VM. Both execution time and memory usage are shown with a confidence interval of 95%. The time measures were acquired using the multi-iteration approach described in [4].

perform both intraprocedural and whole program optimizations with a much wider scope than the analyses presented in this paper.

In letting Soot analyse the same Java applications, we expect it to find a lot more dead assignments among the selected candidates than we do. The main reason for this is that Soot performs a number of additional analyses with the goal of producing efficient bytecode. Soot also performs a more fine-grained analysis. Soot can, for example, eliminate a dead assignment with side-effects on the right-hand side:

```
a = m(); // live due to side-effect on the right-hand side
```

In Soot, the assignment can be eliminated while still preserving the right-hand side:

```
/*a = */m(); //  the assignment is "removed" but call remains
```

If we let Soot analyse the applications listed above, it finds many more dead assignments. Forr example, in ANTLR, Soot finds 1 718 dead candidates while our analyses find 265 at most.

6.2 Performance

Another important aspect of these analyses is their running time and memory usage. In order to be useful in an interactive setting they need to finish within an acceptable time frame and the amount of memory used should preferably be low. All

	JastAddJ			**Soot**	
App.	**Plain**	**Ver. II** *liveness*	**Ver. III** *liveness + reaching*	**Plain**	**-O**
ANTLR	8.0 ± 0.3	13.2 ± 0.3	14.0 ± 0.5	33.4 ± 1.2	58.8 ± 2.1
Bloat	9.3 ± 0.4	12.9 ± 0.4	26.7 ± 0.7	39.0 ± 1.7	67.4 ± 4.0
JCharts	5.9 ± 0.3	13.5 ± 0.7	14.7 ± 0.7	27.6 ± 0.9	38.3 ± 1.5
FOP	59.4 ± 12.0	76.7 ± 1.8	80.4 ± 1.8	139.4 ± 12.4	186.0 ± 23.5

Figure 27: Time for JastAddJ and Soot. Each configuration was run from a terminal 10 times and elapsed time for the process was measured using the Unix command `time`. The averages of these time measures, along with the confidence interval for a confidence interval of 95%, are given above in seconds. Soot performs semantic analysis and translations for the plain version, while with the -O argument Soot performs *several* intraprocedural analyses. JastAddJ performs semantic checks by default while version II and III also performs *one* intraprocedural analysis. No output was generated for any of the configurations.

time and memory measurements referred to below were performed on a Lenovo Thinkpad X61 running Ubuntu 9.10 (Karmic Koala).

With interactive compilation in mind, we measure time for pure compilation and for compilation plus analysis version I, II and III using the multi-iteration approach with a pre-heated VM, as presented in [4]. Figure 26 shows the results as average execution time and memory usage for each application with a confidence interval of 95%. The execution time and memory usage for plain compilation is notably shorter and smaller then the other combinations. The numbers tend to increase with the complexity of the analysis which is to be expected. The numbers for Version III are still within an acceptable time frame. Especially since in an interactive setting, compilation is rarely done in batch mode but rather in a more incremental fashion, one compilation unit at a time. Memory usage increases for all combinations, in particular for ANTLR which has larger methods and hence more dataflow sets to store in memory.

For comparison we also measure execution time from the command line, using the Unix command `time`, for Soot and JastAddJ. The average execution times for different configurations are shown in Figure 6.2. Bear in mind that Soot performs several more intraprocedural analyses with the `-O` option while JastAddJ performs one for version I, II and III. However, they do share the same front end. Without the `-O` option, Soot performs the same work as JastAddJ plus translation to internal intermediate code. This translation, plus potential other tasks internal to Soot and unknown to us, makes up the difference in time between the plain version for JastAddJ and Soot. The numbers in the table are merely included to put the performance of JastAddJ into perspective.

Modules			**Number of Rules**				
Name	**Version**	**LOC**	`syn`	`inh`	`eq`	`coll`	`contr.`
Java Frontend	1.4	10 352	471	168	1 453	0	0
	1.5	4 909	166	48	588	0	0
Control Flow	1.4	301	21	24	126	2	3
	1.5	9	1	0	9	0	0
Liveness	1.4	24	2	0	2	3	5
Reaching	1.4	96	10	0	9	6	14
Side Effects	1.4	22	1	0	1	1	7
Helpers	1.4	27	8	5	17	0	0
– Version I	1.4	25	2	1	3	2	3
– Version II	1.4	7	1	0	3	0	0
– Version III	1.4	21	2	0	4	0	0

Figure 28: Size of modules using lines of code (LOC) and number of JastAdd rules separated into different columns for – `syn`, `inh`, `coll`, `eq`, `contributes`.

6.3 Specification size

Finally, we are interested in the actual size of the implementations. By adding higher-level abstractions in the form of attributes our wish is to decrease the code size needed for the actual dataflow analysis implementation. Table 6.3 shows an overview of different modules, including the frontend of JastAddJ. Each module is separated into two rows when there is a modular extension from Java version 1.4 to Java version 1.5. For cases where such an extension is unnecessary due to reused behavior, only numbers for version 1.4 are given. Besides size we also show the number of JastAdd rules divided into different columns depending on rule type. Besides the modules presented in this paper, we show, for completeness, the size of the Side Effects module and the Helpers module needed by the Control Flow, Liveness, Reaching Definition and Dead Assignment modules. The Dead Assignment module is presented as three parts – one for each version. Generally, the numbers in the table tend to decrease further down in the table showing how we gradually raise the level of abstraction, modularly adding higher-level APIs and as a consequence the sizes of the modules decrease.

7 Related Work

Silver is a recent attribute grammar system with many similarities to JastAdd, but which does not support circular attributes. It has also been applied for declarative flow analysis [23], but using a different approach than ours. In Silver, the specification language itself is extended to support the specification of control-flow and dataflow analysis. The actual data-flow analysis is not carried out by the attribute grammar system, but by an external model checking tool. This approach is motivated by the difficulty of declaratively specifying data-flow analysis on the same program representation as, for example, type analysis. No performance figures for this approach are reported. In contrast, we have shown how both control-flow and dataflow can be specified in a concise way directly using the general attribute grammar features of JastAdd, in particular relying on the combination of reference attributes, circular attributes and collection attributes.

In [18], Mughal describes how control-flow can be expressed using attributes in the Cornell Synthesizer Generator (CSG) [19] with the purpose of performing incremental code generation. Two attributes, `entry` (synthesized) and `completion` (inherited), are presented which point out where to start evaluation of a statement and where to start to evaluate the next statement. These share common ground with the `succ` and `following` attributes presented in this paper. However, neither circular attributes nor real reference attributes are supported by CSG, making it hard to progress towards dataflow analysis using the control-flow implementation.

Farrow introduced circular attributes, and used liveness as a motivating example [10]. He builds on traditional attribute grammars without reference attributes, and does therefore not build any explicit control-flow graph. The dataflow analysis is instead defined directly in terms of the underlying syntax, with rules for each kind of statement.

Morgenthaler [17] has developed static analysis techniques for source-to-source tools. To reduce the cost, techniques for efficient demand-driven analyses are proposed as opposed to traditional exhaustive methods. These techniques operate directly on the AST, the most appropriate data structure for a source-to-source tool architecture. No explicit control-flow representation is built. Instead, a so called *virtual control flow* is constructed by demand-driven computations of all possible control successors and predecessors. Functions realizing this scheme for the C language, implemented in C++, required about 1000 lines of code. A major difference between this approach and ours is that in using JastAdd, the demand-driven evaluator is automatically constructed from concise declarative grammar specifications.

Soot, [21], is a framework for optimizing, analyzing, and annotating Java bytecode. The framework provides a set of intraprocedural and whole program optimizations with a much wider scope than the analyses presented in this paper. Soot is based on several kinds of intermediate code representations, including typed three-address code, and provides seamless translations between the different representations. Java source code is first translated into one of these representations in which some high-level structure is lost. The control-flow and data-flow frameworks in Soot are indeed quite powerful with reasonably small APIs. A major difference, as compared to our approach, is that the Soot approach is not declarative and therefore relies on manual scheduling when combining analyses, or adding new analyses as new specializations of the framework.

Schäfer et al. have used a variant of our analyses modules, extended with finer-grained expression-based control flow, in the implementation of experimental refactoring tools for Java. They report performance on par with industrial strength refactoring tools [20].

8 Conclusions

Control-flow and dataflow analysis are usually cumbersome to implement for source level analyses of complex languages such as Java. The main reason being the tedious work to implement analyses that support all language constructs in today's mainstream languages. Moreover, since languages constantly evolve there is a need to update the analyses accordingly.

We have shown how reference attributed grammars, augmented with circular attributes and collection attributes, provide an excellent foundation for declaratively specifying control-flow and dataflow analysis. The specifications are concise and close to text book definitions, yet the generated analyzers are sufficiently efficient for real applications. As example applications we implemented different versions of high-level dead code analysis, suitable for an interactive setting. Performancewise, our implementations finished within an acceptable time frame and did well in comparison to other optimizing frameworks like Soot. The specifications are also extensible in that the analyses can be extended modularly when new features are added to a language.

Our analyses are performed at the statement level which reduces the granularity of the results. More fine-grained results could be achieved by redefining the control-flow on the expression level, as has been done in [20]. This would most likely result in the location of more dead code and is one topic for future work. There are several other interesting ways to continue this work as well. The design ideas and frameworks presented in this paper are general and it would be interesting to see how they extend to more advanced analyses, e.g., object-oriented call graph construction and inter-procedural points-to analysis. We already have promising work in this direction, for example simple whole program devirtualization analysis [15]. Another possible direction for future work is to design and implement declarative frameworks for other traditional backend analyses such as translation to SSA-form. The technique presented in this paper could in principle be applied to an AST representing intermediate code or bytecode. It would also be interesting to apply these techniques to do domain-specific source level analyses, for example, enforcing framework conventions.

Bibliography

[1] JastAdd, 2007. http://jastadd.org.

[2] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

[3] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 2002.

[4] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, 2008.

[5] John Tang Boyland. Remote attribute grammars. *J. ACM*, 52(4):627–687, 2005.

[6] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. *SIGSOFT Softw. Eng. Notes*, 24(5):21–31, 1999.

[7] Torbjörn Ekman and Görel Hedin. Modular Name Analysis for Java Using JastAdd. In *Generative and Transformational Techniques in Software Engineering*, volume 4143/2006, pages 422–436. Springer Berlin / Heidelberg, 2006.

[8] Torbjörn Ekman and Görel Hedin. The jastadd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

[9] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 1–18, 2007.

[10] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of CC'86*, pages 85–98. ACM Press, 1986.

[11] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[12] Görel Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.

[13] D. H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Proceedings of OOPSLA'86*, pages 347–349, 1986.

[14] Donald E. Knuth. Semantics of context-free languages. *Journal Theory of Computing Systems*, 2(2):127–145, June 1968.

[15] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. Extending attribute grammars with collection attributes - evaluation and applications. In *Proceedings of Seventh IEEE Working Conference on Source Code Analysis and Manipulation*, September 2007.

[16] Eva Magnusson and Görel Hedin. Circular reference attributed grammars - their evaluation and applications. *Science of Computer Programming*, 68(1):21–37, August 2007.

[17] J D Morgenthaler. *Static Analysis for a Software transformation Tool.* Ph.D. thesis, University of San Diego, California, 1997.

[18] Khalid A. Mughal. Generation of incremental indirect threaded code for language-based programming environments. In *Compiler Compilers and High Speed Compilation*, volume 371/1989 of *Lecture Notes in Computer Science*, pages 230–242. Springer Berlin / Heidelberg, 1988.

[19] Thomas Reps and Tim Teitelbaum. The synthesizer generator. *SIGPLAN Not.*, 19(5):42–48, 1984.

[20] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and Extensible Renaming for Java. In Gregor Kiczales, editor, *23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008).* ACM Press, 2008.

[21] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[22] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings PLDI'89*, pages 131–145. ACM Press, 1989.

[23] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute Grammar-based Language Extensions for Java. In *Proceedings of ECOOP'07*, LNCS. Springer, 2007.

PAPER II

PRACTICAL SCOPE RECOVERY USING BRIDGE PARSING

Abstract

Interactive development environments (IDEs) increase programmer productivity, but unfortunately also the burden on language implementors since sophisticated tool support is expected even for small domain-specific languages. Our goal is to alleviate that burden, by generating IDEs from high-level language specifications using the JastAdd meta-compiler system. This puts increased tension on scope recovery in parsers, since at least a partial AST is required by the system to perform static analysis, such as name completion and context sensitive search.

In this paper we present a novel recovery algorithm called bridge parsing, which provides a light-weight recovery mechanism that complements existing parsing recovery techniques. An initial phase recovers nesting structure in source files making them easier to process by existing parsers. This enables batch parser generators with existing grammars to be used in an interactive setting with minor or no modifications.

We have implemented bridge parsing in a generic extensible IDE for JastAdd based compilers. It is independent of parsing technology, which we validate by

Emma Nilsson-Nyman, Torbjörn Ekman, Görel Hedin
In the proceedings of the *1st Conference on Software Language Engineering* (SLE'08), Toulouse, France September 29th-30th, 2008 *Software Language Engineering*, *Lecture Notes of Computer Science* 5452:95–113 © 2009 Springer Berlin / Heidelberg

showing how it improves recovery in a set of typical interactive editing scenarios for three parser generators: ANTLR (LL(variable lookahead) parsers), LPG (LALR(k) parsers), and Beaver (LALR(1) parsers). ANTLR and LPG both contain sophisticated support for error recovery, while Beaver requires manual error productions. Bridge parsing complements these techniques and yields better recovery for all these tools with only minimal changes to existing grammars.

1 Introduction

Interactive development environments (IDE) have become the tool of choice in large-scale software development. This drastically increases the burden on language developers since sophisticated tool support is expected even for small domain-specific languages. The work presented in this paper is part of a larger effort to generate IDEs from high-level language specifications based on attribute grammars in the JastAdd meta-compiler tools. The AST is used as the predominant data structure and all static semantic analyses are implemented as attribute grammars on top of that tree. This approach has been used successfully to implement a production-quality Java compiler [6], extensible refactoring tools [16], source level control-flow and data-flow analyses [14], and various Java extensions [1, 9].

One key insight from our earlier work is that we can use the AST as the only model of the program and then superimpose additional graph structure using attributes on top of that tree, e.g., name bindings, inheritance hierarchies and call graphs. The IDE can then reuse and extend this model to provide services such as name completion, cross-referencing, code outline, and semantic search facilities. However, this allows us to use the same extension mechanisms that have proven successful in building extensible compilers to build extensible IDEs. This puts extreme tension on the error recovery facilities in parsers since incomplete programs are the norm rather than the exception during interactive development, and a recovered AST is necessary for instant feedback to the user. An unfortunate consequence of this challenge is that, despite the wealth of research in automatic parser generators from high-level grammars and sophisticated error recovery mechanisms, most IDEs still rely on hand crafted parsers to provide such services to the user.

In this paper we present an algorithm for scope recovery, preserving as much of the AST structure as possible, that is neither tied to a particular parsing technology nor to a specific parser generator. A light-weight pre-processor takes an incomplete program and recovers scope nesting before the adjusted source file is fed to a traditional parser. Existing error recovery mechanisms are then sufficient to build a partial AST suitable for static semantic analysis. This approach even makes it feasible to use a batch parser rather than an incremental parser since the speed of parsing a complete source unit is usually satisfactory even for interactive editing on today's hardware.

The approach has proven successful when combined with several parser generators in our IDE generator for JastAdd based compilers in the Eclipse IDE framework. We have integrated bridge parsing in an IDE framework for compilers where all services use the AST as the only model to extract program information from. An IDE based on Eclipse is generated from an attribute grammar for static semantic analysis and our largest example is an environment for Java which includes support for name completion, content outline, cross-referencing, and various semantic search facilities.

We demonstrate that the approach is independent of parsing technology and parser generator by combining bridge parsing with three different parser generators: ANTLR which generates an LL(variable lookahead) parser, LPG which generates an LALR(k) parser, and Beaver which generates an LALR(1) parser. LPG and ANTLR are particularly interesting because of their sophisticated error recovery mechanisms. We show that on a set of typical interactive editing scenarios, bridge parsing improves recovery for both these tools and the few cases with degraded performance can be mitigated by minor enhancements in the IDE. The contributions of this paper are:

- A general algorithm for recovering scope information suitable for arbitrary parsing technologies.
- An implementation in an IDE generator for JastAdd based compilers in Eclipse.
- A careful comparison of its effect on error recovery in state of the art parser generators.

The rest of the paper is structured as follows. Section 2 explains the requirements on error recovery and outlines previous work and room for improvement. Bridge parsing is introduced in Section 3 and an example of language sensitive recovery for Java is presented in Section 4 and evaluated in Section 5. We finally discuss future work and conclude in Section 6.

2 Background

We first outline scenarios requiring error recovery in the setting described above, and then survey existing recovery mechanisms and explain why they are not sufficient or have room for improvement. This serves as a background and related work before we introduce bridge parsing in Section 3. A more thorough survey of error recovery techniques is available in [4].

2.1 Error recovery scenarios

The interactive setting this approach is used in puts extreme pressure on error recovery during parsing. Incomplete programs with errors are the norm rather than the exception, and we rely on an AST to be built to perform most kinds of IDE services to the user. It is therefore of paramount importance that common editing scenarios produce an AST rather than a parse failure to allow for services such as name completion while editing. There is also a tension between sophisticated error recovery and the goal to lower the burden on language developers building IDEs. Ideally, she should be able to reuse an existing parser, used in the compiler, in the IDE with minimal changes while providing satisfactory recovery. We define the following desirable properties of error recovery in this context:

- Support for arbitrary parser generators and parsing formalisms.
- Only moderate additions to add recovery to a specific language grammar.
- Effective in that recovery is excellent for common editing scenarios.

The motivation behind the goal of using arbitrary parser generators is that JastAdd defines its own abstract grammar and as long as the parser can build an AST that adheres to that grammar it can be used with JastAdd. We can thus benefit from the wealth of available parsing techniques by selecting the most appropriate for the language at hand. Notice that many proposed parser formalisms are orthogonal to the problem of handling incomplete programs, e.g., GLR-parsing [18], Earley-parsing [5], and Parsing Expression Grammars [7], all deal with ambiguities in grammars rather than errors in programs.

The overall goal of the project is to lower the burden on language developers who want to provide IDE support for their languages. The extra effort to handle incomplete programs should therefore be moderate compared to the overall effort of lexical and syntactic analysis.

The effectiveness criterion is a very pragmatic one. We want the automatic recovery to be as close as possible to what a user would manually do to correct the problem. Here we borrow the taxonomy from Pennello and DeRemer [15] and consider a correction *excellent* if it repairs the text as a human reader would have, otherwise as *good* if the result is a reasonable program and no spurious errors are introduced, and otherwise as *poor* if spurious errors are introduced.

Consider the simple incomplete program below. A class `C` with a method `m()` and a field `x` is currently being edited. There are two closing curly braces missing. An excellent recovery, and the desired result of an automatic recovery, would be to insert a curly brace before and after the field `x`. A simpler recovery, with poor result, would be to insert two curly braces at the end of the program.

```
class C {
  void m() {
    int y;
  int x;
```

Notice that we need to consider indentation to get an excellent result. Changing the indentation of the field `x` to the same as for the variable `y` should, for instance, change its interpretation to a variable and result in a recovery where both closing braces are inserted at the end of the file. Meaning that the simpler recovery alternative above would be sufficient.

2.2 Simple recovery

The simplest form of recovery is to let the parser automatically replace, remove, or insert single tokens to correct an erroneous program. This information can easily be extracted from the grammar and is supported by many parser generators. However, for incomplete programs this is insufficient since series of consecutive tokens are usually missing in the source file. It should be noted that this kind of recovery serves as a nice complement to other recovery strategies by correcting certain spelling errors.

2.3 Phrase recovery

A more advanced form of recovery is phrase level recovery where text that precedes, follows, or surrounds an error token is replaced by a suitable nonterminal symbol [8]. Beacon symbols, e.g., delimiters, and keywords, are used to re-synch the currently parsed production and erroneous tokens are removed from the input stream and replaced by a particular error token. These symbols are language specific and the parsing grammar therefore needs to be extended with error productions unless automatically derived by the parser generator. This form of recovery works very well in practice when beacon symbols are available in the input stream and implemented in many parsing tools. Incomplete programs usually lack some beacons required to re-synch the parser and often result in a parsing failure where the recovery can not proceed.

2.4 Scope recovery

Hierarchical structure is usually represented by nested scopes in programming languages, e.g., blocks and parentheses. It is a common error to forget to close such scopes and during interactive editing many scopes will be incomplete. Burke and Fisher introduced scope recovery to alleviate such problems [2]. Their technique requires the language developer to explicitly provide symbols that are closing scopes. Charles improves on that analysis by automatically deriving such symbols from a grammar by analyzing recursively defined rules [3]. Scope recovery can drastically improve the performance of phrase recovery since symbols to open and close scopes are often used as beacons to re-synch the parser. Scope recovery usually discards indentation information which unfortunately limits the possible result of the analysis to good rather than excellent for the previously outlined example.

2.5 Incremental parsing

An interactive setting makes it possible to take history into account when detecting and recovering from errors. This opens up for assigning blame to the actual edit that introduced an error, rather than to the location in the code where the error was detected. A source unit is not parsed from scratch upon a change but instead only the changes are analyzed. Wagner and Graham present an integrated approach of incremental parsing and a self versioning document model in [20, 21]. It is worth noting that this requires a deep integration of the environment and the generated parser, which makes it less suitable for our setting due to the goal of supporting multiple parser generators.

2.6 Island parsing

Island parsing is not an error recovery mechanism per se but rather a general technique to create robust parsers that are tolerant to syntactic errors, incomplete source code, and various language dialects [12]. It is based on the observation that for source model extraction one is often only interested in a limited subset of all language features. A grammar consists of detailed productions describing language constructs of interests, that are called islands, and liberal productions matching the remaining constructs, that are called water. This allows erroneous programs to be parsed as long as the errors are contained in water. The parts missing in an incomplete program are usually a combination of both water and islands which makes this approach less suitable for extracting hierarchical structure on its own.

3 Bridge Parsing

We now present *bridge parsing* as a technique to recover scope information from an incomplete or erroneous source file. It combines island parsing with layout sensitive scope recovery. In [12] Moonen defines island grammars [13, 19] as follows:

*An **island grammar** is a grammar that consists of two parts: (i) detailed productions that describe the language constructs that we are particularly interested in (so called **islands**), and (ii) liberal productions that catch the remainder of the input (so called **water**).*

In bridge parsing, tokens which open or close scopes are defined as islands while the rest of the source text is defined as water or reefs. Reefs are described further in Section 3.1. This light-weight parsing strategy is used to detect open scopes and to close them. The end product of the bridge parser is a recovered source representation suitable for any parser that supports phrase level recovery.

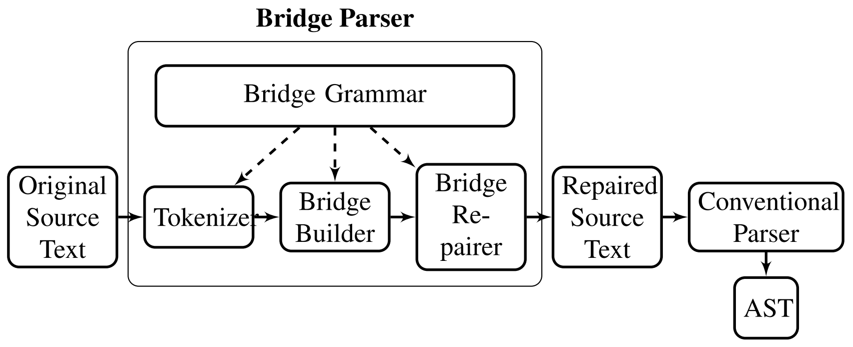


Figure 1: A bridge parser consists of three parts – a tokenizer returning a token list, a bridge builder taking a token list and returning a bridge model and a bridge repairer which takes a bridge model and generates a repaired source text.

A bridge parser is built from three parts, as illustrated in Figure 1. The first part, the tokenizer, takes a source text and produces a list of tokens based on definitions in the *bridge grammar*. The second part, the bridge builder, constructs a bridge model from the token list and the last part, the bridge repairer, analyses and repairs the bridge model.

3.1 Tokenizer

The goal of the tokenizer is to produce a list of tokens from a source text. Token definitions are provided by a bridge grammar which extends the island grammar formalism with bridges and reefs:

*A **bridge grammar** extends an island grammar with the notions of **bridges** and **reefs**: (i) reefs are attributed tokens which add information to nearby islands, and (ii) bridges connect matching islands. All islands in a bridge grammar are seen as potential bridge abutments.*

We are primarily interested in tokens that define hierarchical structure through nested scopes. Tokens that open or close a scope are therefore represented as islands in the grammar, e.g., braces and parentheses in Java, while most other syntactic constructs are considered water. However, additional tokens in the stream may be interesting to help match two islands that open and close a particular scope. Typical examples include indentation and delimiters, and we call such tokens *reefs*. Reefs can be annotated with attributes which enables comparison between reefs of the same type. Indentation reefs may for instance have different indentation levels. We call such tokens attributed tokens:

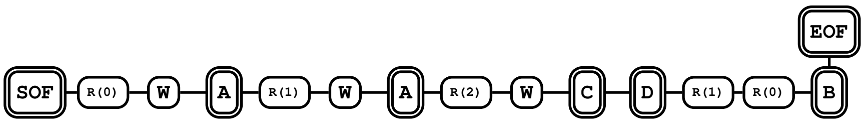


Figure 2: The token list is a simple double-linked list of *islands* (A, B, C, D), *reefs* (R) and *water* (W). Two additional nodes representing *start of file* (SOF) and *end of file* (EOF) are added first and last in the list. The numbers within parentheses show the attribute values of the reefs.

***Attributed tokens** are tokens which have attributes, which potentially makes them different from other attributed tokens of the same type. This difference makes it possible to compare attributed tokens to each other.*

The first step when defining a bridge grammar, is to specify islands and reefs. The following example, which we will use as a running example, shows the first part of a bridge grammar relevant to the tokenizer:

Listing II.1: Tokenizer Definitions

```
islands SOF, EOF, A=.., B=.., C=.., D=..
reefs R(attr)=..
```

The bridge grammar defines four island types and one reef type. The SOF and EOF islands, represent *start of file* and *end of file*. The A and B islands could, for instance, be open and close brace and C and D open and close parenthesis. The reef could represent indentation where the value of *attr* is the level of indentation. In our implementation each island corresponds to a class that accepts and consumes the substring matching its lexical representation. The information given so far in the example grammar is sufficient to create a lexer which will produce a list of tokens, e.g., the example in Figure 2.

3.2 Bridge Builder

The bridge builder takes a token list and produces a bridge model defined as follows:

*A **bridge model** is a token list where matched islands are linked with bridges in alignment with the nesting structure in a source text. Bridges can be enclosing other bridges or be enclosed by other bridges, or both, but they never cross each other in an ill-formed manner. Unmatched islands point out broken parts of the nesting structure.*

In the bridge model, islands opening and closing a scope should be linked with a bridge. For the bridge builder to know between which islands to build bridges we need to add definitions to the bridge grammar.

Listing II.2: Bridge Builder Definitions

```
bridge from SOF to EOF { ... }
bridge from [a:R A] to [b:R B] when a.attr = b.attr { ... }
bridge from [a:R C] to [b:R D] when a.attr = b.attr { ... }
```

The first bridge, the *file bridge*, between the SOF island and EOF island does not need any additional matching constraints i.e., constraints that define when two islands of given types match. For other bridges additional constraints besides type information i.e., an island of type A and type B match, are usually needed. In the example above the islands of type A and B match if both islands have reefs of type R to the left which are equal. The constraints for the third bridge are similar.

The order of the bridge abutments in the definition should correspond to the order in the source text e.g., the bridge start of type C is expected to occur before the bridge end of type D.

With the information given so far in our example bridge grammar we can construct a bridge model. The BRIDGE-BUILDER algorithm will construct as many bridges as possible. If at some point no matching island can be found the algorithm will jump over the unmatched island and eventually construct a bridge enclosing the island. The algorithm is finished when the file bridge has been constructed.

The BRIDGE-BUILDER algorithm, listed in Figure 4, will iterate through the token list until it manages to build the file bridge. For each unmatched bridge start it will try to match it to the next island. If there is a match a bridge will be built and otherwise the algorithm will continue with the next unmatched bridge start. Each time a bridge is encountered it will be crossed. In this way the number of encountered unmatched islands will decrease each iteration.

The algorithm will try to match an unmatched island to the next unmatched island within the current tolerance. The tolerance defines how many unmatched islands that are allowed below a bridge. This value is only changed between iterations and depends on the success of the last iteration. The tolerance should be as low as possible. We start out with a tolerance of zero, meaning no unmatched islands under a bridge. This value will increase if we fail to build bridges during an iteration. If we have a successful iteration we reset the tolerance to zero. If we apply this algorithm to the running example we end up with four iterations, illustrated in Figure 3.

3.3 Bridge Repairer

The goal of the bridge repairer is to analyze the bridge model and repair it if necessary. The BRIDGE-REPAIRER algorithm needs to locate remaining unmatched

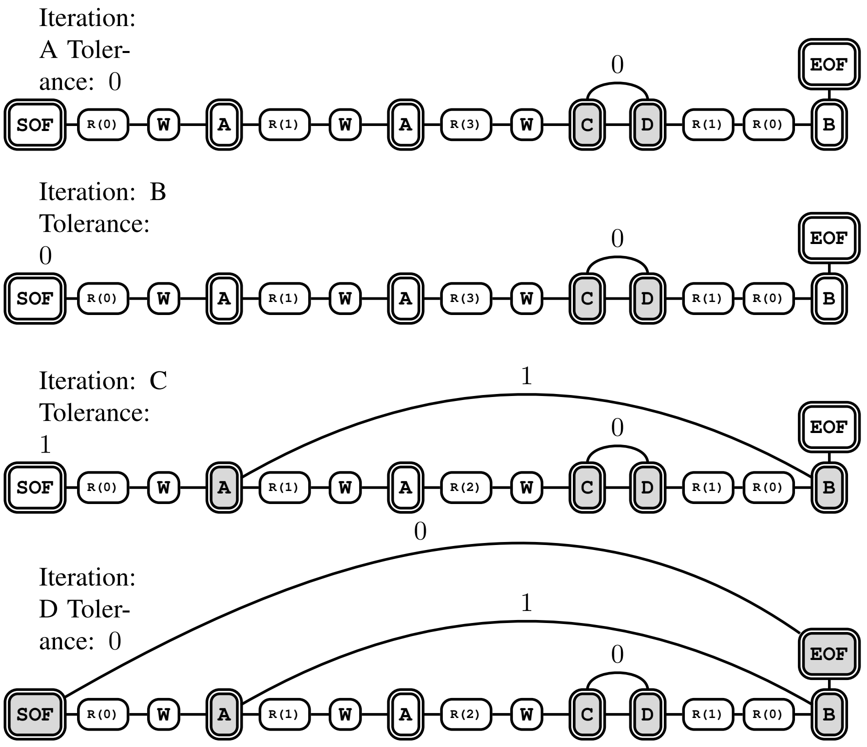


Figure 3: The resulting bridge model after running the BRIDGE-BUILDER algorithm. No bridges were built during iteration B which results in an increased tolerance in iteration C. During iteration C a bridge is built which means the tolerance is reset to zero in iteration D.

```
BUILD-BRIDGES($sof$)
tol ← 0
while ¬HAS-BRIDGE($sof$)
    do $start \leftarrow sof$
       $change \leftarrow$ FALSE
       while $start \neq$ NIL
           do $end \leftarrow$ NEXT-UNMATCHED-ISLAND($start, tol$)
              if BRIDGE-MATCH($start, end$)
                then BUILD-BRIDGE($start, end$)
                     $change \leftarrow$ TRUE
                     $start \leftarrow$ NEXT-UNMATCHED-START-ISLAND($end$)
                else $start \leftarrow$ NEXT-UNMATCHED-START-ISLAND($start$)
       if ¬ $change$
         then $tol \leftarrow tol + 1$
         else if $tol > 0$
                then $tol \leftarrow 0$
```

Figure 4: The BUILD-BRIDGES algorithm constructs a bridge model from a token list. The NEXT-UNMATCHED-ISLAND returns the next unmatched island to the right of the given island. The tolerance defines the number of unmatched islands to jump over before the procedure returns.
NEXT-UNMATCHED-START-ISLAND is similar but only looks for bridge starts.

islands and to add *artificial islands* which will match them. Each island defined in the bridge grammar can potentially be missing. When an island is missing, the algorithm will search for an appropriate *construction site* where an artificial island can be inserted. The search for a construction site starts from the bridge start if the bridge end is missing, and vice versa, and ends when a match is found or an enclosing bridge is encountered. With this in mind we add additional definitions to our example bridge grammar:

Listing II.3: Bridge Repair Definitions

```
bridge from [a:R A] to [b:R B] when a.attr = b.attr {
    missing[A] find [c:R] where c.attr = b.attr insert after
    missing[B] find [c:R] where c.attr = a.attr insert after
}
bridge from [a:R C] to [b:R D] when a.attr = b.attr {
    missing[C] find [c:R] where c.attr = b.attr insert after
    missing[D] find [c:R] where c.attr = a.attr insert after
}
```

If an island of type A is missing, a construction site will be found in the interval starting at the unmatched island of type B and ending at the start of the enclosing bridge. The first reef of type R which is equal to the reef to the left of the unmatched island of type B points out a construction site. The final information we need is how to insert the artificial island. In this case the artificial island should be inserted after the reef. The definitions for the remaining islands are similar.

The BRIDGE-REPAIRER algorithm, listed in Figure 5, recursively repairs unmatched islands under a bridge, starting with the file bridge. When an unmatched island is encountered the MEND algorithm, listed in Figure 6, will locate a construction site and insert an artificial island.

In the running example, an island of type A is missing an island of type B. The island of type A has a reef of type R on its left hand side with value 1 which means the algorithm will search for for a another reef the same type with the same value. The search is stopped when either a reef is found or a closing island of an enclosing scope is encountered. In this case there is a reef of the right type and value, before the enclosing island of type B, which points out a construction site.

The result of the BRIDGE-REPAIRER algorithm is shown in Figure 7. An artificial island of type B has been inserted to form a bridge with the previously unmatched island of type A.

4 Bridge Parsing for Java

To construct a bridge parser for Java we need to create a bridge grammar which can provide information to each of the three parts of the bridge parser, illustrated

```
BRIDGE-REPAIRER($bridge$)
 1  $start \leftarrow$ START($bridge$)
 2  $end \leftarrow$ END($bridge$)
 3  $island \leftarrow$ NEXT-ISLAND($start$)
 4  while $island \neq end$
 5      do if $\neg$HAS-BRIDGE($island$)
 6            then if START-OF-BRIDGE($island$)
 7                    then MEND-RIGHT($island, end$)
 8                    else MEND-LEFT($start, island$)
 9         $bridge \leftarrow$ BRIDGE($island$)
10         BRIDGE-REPAIRER($bridge$)
11         $island \leftarrow$ NEXT-ISLAND(END($bridge$))
```

Figure 5: The BRIDGE-REPAIRER algorithm constructs artificial islands to match unmatched islands. The MEND-RIGHT algorithm is described further in Figure 6. The MEND-LEFT algorithm is symmetric to the MEND-RIGHT algorithm.

```
MEND-RIGHT($broken, end$)
1  $node \leftarrow$ NEXT($broken$)
2  while $node \neq end$
3      do if HAS-BRIDGE($node$)
4            then $node \leftarrow$ NEXT(BRIDGE-END($node$))
5            else if POSSIBLE-CONSTRUCTION-SITE($broken, node$)
6                    then CONSTRUCT-ISLAND-AND-BRIDGE($broken, node$)
7                         return
8            else $node \leftarrow$ NEXT($node$)
9  CONSTRUCT-ISLAND-AND-BRIDGE($broken$, PREVIOUS($end$))
```

Figure 6: The MEND-RIGHT algorithm constructs an artificial bridge end in the interval starting at the unmatched bridge start (*broken*) and ending at the end of the enclosing bridge (*end*).

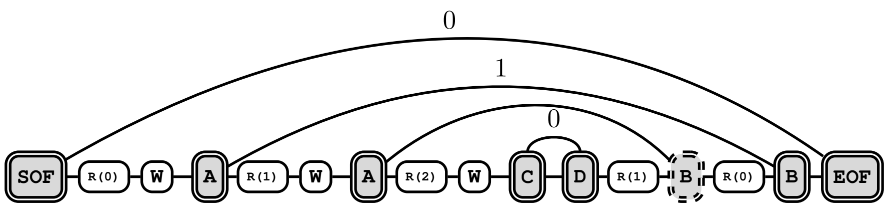


Figure 7: The changes in the example bridge model after the BRIDGE-REPAIRER algorithm is done. The artificial island is marked with a dashed double edge.

in Figure 1. We will define this bridge grammar for Java in three steps which gradually will include more complex language-specific information. The performance impact of these different levels of language sensitivity is evaluated in Section 5.

For Java, and other languages with similar language constructs, we need to consider how to deal with comments and strings. These constructs might enclose text which would match as reefs or islands but which should be ignored. In our implementation we handle this separately in the lexer implementation.

4.1 Scopes

The first level of language sensitivity is to only consider tokens directly defining scopes. We therefore include braces and parentheses as islands since they have a direct impact on the nesting structure of Java code. Indentation is also included to enhance matching of islands in incomplete code. The complete bridge grammar looks like this:

Listing II.4: Bridge Repairer Definitions

```
islands SOF, EOF, LBRACE, RBRACE, LPAREN, RPAREN
reefs INDENT(pos)

bridge from SOF to EOF

bridge from [a:INDENT LBRACE] to [b:INDENT RBRACE]
    when a.pos = b.pos {
    missing [RBRACE]
        find [c:INDENT] where (c.pos <= a.pos) insert after
    missing [LBRACE]
        find [c:INDENT] where (c.pos <= a.pos) insert after
}

bridge from [a:INDENT LPAREN] to [b:INDENT RPAREN]
    when a.pos = b.pos {
    missing [RPAREN]
        find [c:ISLAND] insert before
        find [c:INDENT] where (c.pos <= a.pos) insert after
    missing [LPAREN]
        find [c:ISLAND] insert before
        find [c:INDENT] where (b.pos <= c.pos) insert before
}
```

The pos attribute of the INDENT reef corresponds to the indentation level. Comparing two reefs of this type corresponds to comparing their pos attribute.

For the islands corresponding to right and left parentheses there are two **find** conditions. For cases like these the first occurrence that fulfills all its conditions decide which action to take.

4.2 Delimiters

To improve matching of parentheses we add additional reefs for delimiters. The following code snippet illustrates the benefit of defining commas as reefs during recovery:

```
void m(int a) {
  n(o(, a);    // Recover to "n(o(),a)" and not to "n(o(,a))"
}
```

We have a call to o() as the first argument in the call to n() in the method m(). The comma tells us that we are editing the first element in the list of arguments and that the call to o() should be closed right before the comma rather than after reading the a parameter. If we modify the code snippet and remove the other end parenthesis instead we end up with a different scenario:

```
void m(int a) {
  n(o(), a;    // Recover to "n(o(),a);" and not to "n(o(),a;)"
}
```

The analysis should ideally place a closing parenthesis somewhere after the comma but before the semicolon. The reason is that the comma separates elements within the parentheses and the call to o() is within the call to n(), while the semicolon separates elements in the block. To deal with these scenarios we define additional reefs to match delimiters and add additional **find** declarations to the missing parenthesis blocks.

Listing II.5: Bridge Repairer Definitions

```
reefs .., COMMA, DOT, SEMICOLON

missing [RPAREN]
    ..
    find [c:COMMA] where (previous(c) != WATER) insert before
    find [c:DOT] where (previous(c) != WATER) insert before
    find [c:SEMICOLON] insert before

missing [LPAREN]
    ..
    find [c:COMMA] where (next(c) != WATER) insert after
    find [c:DOT] where (next(c) != WATER) insert after
    find [c:SEMICOLON] insert after
```

These definitions should be seen as extensions to the bridge grammar presented in the previous section. In the above **find** declaration for RPAREN we have added actions for when we encounter a COMMA, DOT or SEMICOLON while searching for a construction site.

4.3 Keywords

To further improve matching we can add keywords as reefs. This can be useful since keywords separate statements from each other. The following code snippet shows an example:

```
boolean m(boolean a) {
  if a == true)      // Recover to "if (a == true)"
    return false;    // and not to "(if a == true)"
  return true;
}
```

Ideally, the analysis should put the missing left parenthesis after the `if` keyword If the keyword has been defined as a reef this is possible, otherwise not since then keywords will be considered to be water. To deal with scenarios such as these we add additional keywords and **find** definitions to the bridge grammar:

Listing II.6: Bridge Repairer Definitions

```
reefs KEYWORD (if, for, while ..)

missing [RPAREN]
    find [c:KEYWORD] insert before

missing [LPAREN]
    find [c:KEYWORD] insert after
```

5 Evaluation

We have chosen to evaluate bridge parsing on common editing scenarios for Java using the specifications described in Section 4. Our bridge parsing implementation has been combined with a set of Java parsers, generated using state of the art parser generators based on LALR(1) and LL(variable lookahead) grammars. The parsers were generated with the following parser generators and settings:

- **Antlr** Generated using Antlr (v.3.0).
- **AntlrBT** Generated using a forth-coming version of Antlr (v.3.1 beta) with backtracking turned on. This version of Antlr introduces new error recovery not yet available in the latest stable version.
- **Beaver** Generated with Beaver (v.0.9.6.1).
- **BeaverEP** Generated with Beaver (v.0.9.6.1), with error productions manually added to the grammar.
- **LPG** Generated using LPG (v.2.0.12). While this is the newest version of LPG it does not yet provide a complete automatic scope recovery as suggested by Charles [3]. To allow comparison with Charles approach the test cases where manually edited to correspond to the suggested recovery from the generated parser.

In lack of an existing appropriate benchmark suite we have created a test suite for incomplete and erroneous Java programs to use during evaluation. The test suite consists of several series of tests, each series with one *correct program* and one or more *broken programs*. The correct program in each series corresponds to the intention of the programmer, while the broken programs illustrate variations of the correct program where one or more tokens are missing. Broken programs in this setting illustrate how programs may evolve during an editing session. For each test series and parser we build a tree representing the correct program and try to do the same for each broken program. For the Antlr, AntlrBT and LPG we build parse trees, while for the Beaver and BeaverEP we build ASTs.

As a metric of how close a tree constructed from one of the broken programs is to the tree constructed for the correct program we use tree alignment distance, as described in [10]. To calculate tree alignment distance, a cost function is required which provides a cost for insertion, deletion and renaming of nodes. We use a simple cost function where all these operations have the cost one. As a complementary classification of success we use the categorization of recovery as excellent, good, or poor by Pennello and DeRemer [15].

5.1 Benchmark examples

The test suite consists of 10 correct test cases which have been modified in various ways to illustrate possible editing scenarios. The test suite provides a total of 41 tests. Full documentation of the test suite can be found at [17]. We have focused on three editing scenarios:

Incomplete code Normally, programs are written in a top-down, left-right fashion. In this scenario we put test cases with incomplete code, to illustrate how code evolves while being edited in an IDE. An example where a user is adding a method `m()` to a class `C` which already contains two fields `x` and `z` may look like this:

```
class C {
    int x;
    void m() {
        int x;
        if (true) {
            int y;

    int z;
}
```

Missing start This scenario highlights situations which might occur when the normal course of writing a program top-down, left-right is interrupted. A typical example is that the user goes back to fix a bug or to change an implementation in an existing program. Consider the example below where the programmer has started changing a while loop which causes a missing opening brace:

```
class C {
    void m() {
        // while (true) {
            int a;
        }
    }
}
```

Tricky indentation Since bridge parsing relies on indentation it is reasonable to assume that tricky indentation is its Achilles heel. We therefore included a set of test cases with unintuative indentation to evaluate its performance on such programs. An example of nested blocks where indentation is not increasing is shown below:

```
class C {
    void m() {
}
 }
```

Another scenario leading to a similar situation is when a programmer pastes a chunk of code with different indentation in the middle of her program, as illustrated by this example:

```
class C {
    void n() { .. }
void m() {
}
}
```

5.2 Results

The results, after running through the test suite with our parser suite, are shown in Table 8, 9 and 10. The first table shows the results for tests with incomplete code, the second table shows results for tests with missing starts and the last table shows results for tests with tricky indentation. Each table has a column for each parser generator containing a set of tree editing distances: without bridge parsing, with bridge parsing using the scopes version, with bridge parsing using the delimiters version. After each tree editing distance set the result for the test case and parser generator is summarized with a letter indicating excellent (**E**), improved (**I**), status quo (**S**) or worse (**W**). A tree alignment distance of 0 indicates a full recovery and an excellent result while a missing value indicates total failure which is when no AST or parse tree could be constructed.

The leftmost column for each parser in the tables shows that the test suite presents many challenges to all parsers which manifest themselves in less than excellent recoveries, except for the test cases in Table 10 where only indentation is changed to trick the bridge parser. Without bridge parsing, LPG is much better than the other parser generators, most likely due to the built-in support for scope recovery.

The second column for each parser shows that all parser generators benefit vastly from bridge parsing in most cases, of 41 cases 19 improve from good or poor recovery to excellent. LPG still has the edge over the other generators with superior error recovery. We notice that using indentation can indeed improve scope recovery since the LPG results are improved in many cases.

The third column in each column set shows that there is almost no change when we use the delimiters version of the bridge parser instead of the scopes version. Generally, nothing changes or there are small improvement.

There are some problems with bridge parsing, as shown in Table 10, when there are inconsistencies in the layout. This problem could be alleviated by IDE support to automatically correct indentation during editing and pasting. Because of the current problems with some layout scenarios the bridge parser is only run when the parser fails to construct an AST and there is no other way to acquire an AST.

We have run tests with keyword sensitive bridge parsing as well, but saw no improvement using our test suite. There are certainly cases where this could yield an improvement but we could not easily come up with a convincing realistic editing scenario to include in the test suite.

Test	Antlr	AntlrBT	Beaver	BeaverEP	LPG
`A1`	-, 0, 0 **E**	65, 0, 0 **E**	63, 0, 0 **E**	63, 0, 0 **E**	6, 0, 0 **E**
`A2`	-, 0, 0 **E**	65, 0, 0 **E**	63, 0, 0 **E**	63, 0, 0 **E**	4, 0, 0 **E**
`A3`	75, 0, 0 **E**	1, 0, 0 **E**	63, 0, 0 **E**	63, 0, 0 **E**	0, 0, 0 **E**
`A4`	-, 0, 0 **E**	65, 0, 0 **E**	63, 0, 0 **E**	63, 0, 0 **E**	2, 0, 0 **E**
`B1`	75, 0, 0 **E**	28, 0, 0 **E**	73, 0, 0 **E**	30, 0, 0 **E**	0, 0, 0 **E**
`B2`	-, 0, 0 **E**	11, 0, 0 **E**	73, 0, 0 **E**	73, 0, 0 **E**	0, 0, 0 **E**
`B3`	29, 2, 0 **E**	29, 1, 0 **E**	33, -, 0 **E**	33, -, 0 **E**	8, 2, 1 **E**
`B4`	1, 0, 0 **E**	1, 0, 0 **E**	-, 0, 0 **E**	-, 0, 0 **E**	0, 1, 1 **E**
`B5`	75, 0, 0 **E**	1, 0, 0 **E**	73, 0, 0 **E**	73, 0, 0 **E**	8, 0, 0 **E**
`C1`	-, 7, 7 **I**	249, 7, 7 **I**	207, 5, 5 **I**	207, 5, 5 **I**	9, 5, 5 **I**
`C2`	249, 0, 0 **E**	29, 0, 0 **E**	207, 0, 0 **E**	123, 0, 0 **E**	0, 0, 0 **E**
`C3`	-, 0, 0 **E**	33, 0, 0 **E**	207, 0, 0 **E**	207, 0, 0 **E**	19, 0, 0 **E**
`D1`	168, 0, 0 **E**	114, 0, 0 **E**	124, 0, 0 **E**	81, 0, 0 **E**	12, 0, 0 **E**
`D2`	168, 0, 0 **E**	37, 0, 0 **E**	124, 0, 0 **E**	124, 0, 0 **E**	16, 0, 0 **E**
`D3`	168, 0, 0 **E**	65, 0, 0 **E**	124, 0, 0 **E**	105, 0, 0 **E**	2, 0, 0 **E**
`D4`	168, 0, 0 **E**	15, 0, 0 **E**	124, 0, 0 **E**	124, 0, 0 **E**	10, 0, 0 **E**
`E1`	31, -, - **W**	28, 18, 17 **I**	109, 109, 109 **S**	47, 47, 109 **W**	18, 12, 10 **I**
`E2`	-, -, - **S**	38, 18, 17 **I**	-, 109, 109 **I**	-, 109, 37 **I**	24, 10, 10 **I**
`E3`	125, 0, 0 **E**	16, 0, 0 **E**	109, 0, 0 **E**	109, 0, 0 **E**	11, 0, 0 **E**
`F1`	151, 0, 0 **E**	67, 0, 0 **E**	106, 0, 0 **E**	54, 0, 0 **E**	25, 0, 0 **E**
`F2`	151, 0, 0 **E**	44, 0, 0 **E**	106, 0, 0 **E**	54, 0, 0 **E**	9, 0, 0 **E**
`F3`	151, 0, 0 **E**	48, 0, 0 **E**	106, 0, 0 **E**	-, 0, 0 **E**	9, 0, 0 **E**
`G1`	1, 0, 0 **E**	1, 0, 0 **E**	-, 0, 0 **E**	-, 0, 0 **E**	0, 0, 0 **E**
`G2`	-, 1, 1 **I**	13, 1, 1 **I**	154, 0, 0 **E**	114, 0, 0 **E**	11, 0, 0 **E**
`G3`	-, -, - **S**	36, 34, 34 **I**	154, 154, 154 **S**	154, 154, 154 **S**	2, 2, 2 **S**
`H1`	116, 2, 0 **E**	116, 1, 0 **E**	96, -, 0 **E**	96, -, 0 **E**	13, 2, 1 **I**
`H2`	-, 2, 0 **E**	97, 1, 0 **E**	73, 11, 0 **E**	73, 11, 0 **E**	16, 2, 0 **E**
`H3`	-, -, - **S**	57, 4, 4 **I**	-, 117, 117 **I**	-, 50, 50 **I**	4, 1, 1 **I**
`H4`	-, -, - **S**	8, 7, 7 **I**	117, 15, 15 **I**	117, 15, 15 **I**	13, 5, 5 **I**
`I1`	1, 1, 1 **S**	2, 1, 1 **I**	19, 19, 19 **S**	19, 19, 19 **S**	0, 0, 0 **S**
`I2`	2, 1, 1 **I**	2, 1, 1 **I**	21, 21, 21 **S**	21, 21, 21 **S**	15, 15, 15 **S**
`I5`	0, 5, 5 **W**	0, 3, 3 **W**	15, 105, 105 **W**	15, 15, 15 **W**	0, 1, 1 **W**
`I4`	0, 0, 0 **E**	0, 1, 1, **W**	15, -, -, **W**	15, -, -, **W**	0, 3, 3, **W**
`I6`	1, 1, 1 **S**	2, 1, 1 **W**	23, 23, 23 **S**	23, 23, 23 **S**	0, 0, 0 **S**

Figure 8: Results for test cases with incomplete code

6 Conclusions

We have presented bridge parsing as a technique to recover from syntactic errors in incomplete programs with the aim to produce an AST suitable for static seman-

Test	Antlr	AntlrBT	Beaver	BeaverEP	LPG
A5	9, 0, 0 **E**	9, 0, 0 **E**	63, 0, 0 **E**	63, 0, 0 **E**	5, 0, 0 **E**
C4	248, 4, 2 **I**	193, 193, 2 **I**	207, 207, 34 **I**	153, 153, 34 **I**	4, 4, 1 **I**

Figure 9: Results for test cases with missing starts

Test	Antlr	AntlrBT	Beaver	BeaverEP	LPG
A6	0, 3, 3 **W**	0, 3, 3 **W**	0, 2, 2 **W**	0, 2, 2 **W**	0, 1, 1 **W**
B6	0, 10, 10 **W**	0, 64, 64 **W**	0, 73, 73 **W**	0, 23, 23 **W**	0, 4, 4 **W**
I3	0, -, 0 **E**	0, 14, 0 **E**	0, -, 105 **W**	0, -, 23 **W**	0, 4, 4 **W**
J1	0, 17, 17 **W**	0, 59, 59 **W**	0, 49, 49 **W**	0, 49, 49 **W**	0, 15, 15 **W**
J2	0, 0, 0 **E**	0, 0, 0 **E**	0, 13, 13 **W**	0, 13, 13 **W**	0, 3, 3 **W**

Figure 10: Results for test cases with tricky indentation

tic analysis. This enables tool developers to use existing parser generators when implementing IDEs rather than writing parsers by hand. The approach has proven successful when combined with several parser generators in our IDE generator for JastAdd based compilers in Eclipse.

The approach is highly general and can be used in combination with many different parsing technologies. We have validated this claim by showing how it improves error recovery for three different parser generators in common interactive editing scenarios.

One of the main goals of this work is to lower the burden on language developers who want to provide IDE support for their language. It is pleasant to notice that the language models for bridge parsing are very light-weight, yet yield good recovery on complex languages as exemplified by Java in this paper. We believe that it would be easy to adjust the bridge parser presented in this paper to support other languages as well.

As future work we would like to investigate the possibility of integrating history based information into the bridge model, work on improving the handling of incorrect layout and investigate how to derive bridge grammars from existing base-line grammars [11]. An other area we would like to look into is to improve the test suite by observing editing patterns passively from existing code and actively during development.

Bibliography

[1] Pavel Avgustinov, Torbjörn Ekman, and Julian Tibble. Modularity First: A Case for Mixing AOP and Attribute Grammars. In *AOSD*. ACM Press, 2008.

[2] Michael G. Burke and Gerald A. Fischer. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Trans. Program. Lang. Syst.*, 9(2):164–197, 1987.

[3] Philippe Charles. *A practical method for constructing efficient LALR(K) parsers with automatic error recovery*. PhD thesis, New York, NY, USA, 1991.

[4] Pierpaolo Degano and Corrado Priami. Comparison of syntactic error handling in LR parsers. volume 25, pages 657–679, New York, NY, USA, 1995. John Wiley & Sons, Inc.

[5] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[6] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM.

[7] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. volume 39, pages 111–122, New York, NY, USA, 2004. ACM.

[8] Susan L. Graham, Charles B. Haley, and William N. Joy. Practical LR error recovery. In *SIGPLAN '79: Proceedings of the 1979 SIGPLAN symposium on Compiler construction*, pages 168–175, New York, NY, USA, 1979. ACM.

[9] Shan Shan Huang, Amir Hormati, David Bacon, and Rodric Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In Jan Vitek, editor, *Proceedings of ECOOP 2008*. Springer, 2008.

[10] Tao Jianga, Lusheng Wang, and Kaizhong Zhang. Alignment of trees - an alternative to tree edit. In *Theoretical Computer Science*, volume 143, pages 137–148. Elsevier Science B.V., 1995.

[11] Steven Klusener and Ralf Lämmel. Deriving tolerant grammars from a base-line grammar. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 179, Washington, DC, USA, 2003. IEEE Computer Society.

[12] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings. Eighth Working Conference on Reverse Engineering*, pages 13–22. IEEE Computer Society Press, October 2001.

[13] Leon Moonen. Lightweight impact analysis using island grammars. In *Proceedings of the 10th IEEE International Workshop of Program Comprehension*, pages 219–228. IEEE Computer Society, 2002.

[14] Emma Nilsson-Nyman, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. Declarative intraprocedural flow analysis of Java source code. In *Proceedings of the Eight Workshop on Language Description, Tools and Applications (LDTA 2008)*, Electronic Notes in Theoretical Computer Science. Elsevier B.V., 2008.

[15] Thomas J. Pennello and Frank DeRemer. A forward move algorithm for LR error recovery. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 241–254, New York, NY, USA, 1978. ACM.

[16] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for Java. In Gail E. Harris, editor, *OOPSLA*, pages 277–294. ACM, 2008.

[17] Emma Söderberg. Bridge Parsing Test Suite, 2008. `http://cs.lth.se/Emma_Soderberg` [Accessed May 2010].

[18] Masaru Tomita. *An efficient context-free parsing algorithm for natural languages and its applications*. PhD thesis, Pittsburgh, PA, USA, 1985.

[19] Arie van Deursen and Tobias Kuipers. Building documentation generators. In *IEEE International Conference on Software Maintenance*, pages 40–49, August 1999.

[20] Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments*. PhD thesis, Berkeley, CA, USA, 1998.

[21] Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing. *ACM Trans. Program. Lang. Syst.*, 20(5):980–1013, 1998.

PAPER III

NATURAL AND FLEXIBLE ERROR RECOVERY FOR GENERATED PARSERS

Abstract

Parser generators are an indispensable tool for rapid language development. However, they often fall short of the finesse of a hand-crafted parser, built with the language semantics in mind. One area where generated parsers have provided unsatisfactory results is that of error recovery. Good error recovery is both natural, giving recovery suggestions in line with the intention of the programmer; and flexible, allowing it to be adapted according to language insights and language changes. This paper describes a novel approach to error recovery, taking into account not only the context-free grammar, but also indentation usage. We base our approach on an extension of the SGLR parser that supports fine-grained error recovery rules and can be used to parse complex, composed languages. We take a divide-and-conquer approach to error recovery: using indentation, erroneous regions of code are identified. These regions constrain the search space for applying recovery rules, improving performance and ensuring recovery suggestions local to the error. As a last resort, erroneous regions can be discarded. Our approach also integrates bridge parsing to provide more accurate suggestions for indentation-sensitive lan-

Maartje de Jonge, Emma Nilsson-Nyman, Lennart C.L. Kats, Eelco Visser
In the proceedings of the *2nd Conference on Software Language Engineering* (SLE'09), Denver, Colorado, USA October 5th-6th, 2009 *Software Language Systems*, *Lecture Notes of Computer Science* 5969:204–223 © 2010 Springer Berlin / Heidelberg

guage constructs such as scopes. We evaluate our approach by comparison with the JDT Java parser used in Eclipse.

1 Introduction

Domain-specific languages offer substantial gains in expressiveness and ease of use for a particular problem domain. To efficiently construct and use domain-specific languages, language development environments should be used, such as IMP [6], the Meta-Environment [27], MontiCore [14], openArchitectureWare [8], or Spoofax/IMP [13]. With these tools, languages are constructed using a grammar as the principal artifact. Using a parser generator, a grammar can be used to automatically generate a parser. When deployed, the parser constructs abstract syntax trees (ASTs) from programs, used to provide the user with syntactical and semantic editor services, such as an outline view and error marking.

Parser generators are an indispensable tool for rapid language development, allowing the language to be quickly changed according to new domain insights and needs. Yet general-purpose programming languages are often still constructed using handcrafted or partially handcrafted parsers. For example, the Java parser used in the popular Eclipse JDT Java editor, is based on a parser generated by JikesPG (now known as LPG) [5]. However, the parser employs handwritten recovery rules as well as a number of large, customized Java components.

The reason often stated for not using a purely generated parser is that they fall short of the finesse of a handcrafted one, built with the language semantics in mind. A particular area where generated parsers have provided unsatisfactory results is that of error recovery, which is essential for parsing incomplete and syntactically incorrect programs, and thus indispensable for interactive editors. Problems with error recovery in generated parsers are the quality of the recovered program and the reported errors, and finding a good trade-off between recovery quality and performance.

Some parser generators allow custom recovery rules to improve error recovery quality [2, 5, 10, 12]. Custom recovery rules allow a language engineer to inspect and improve an error recovery strategy. Compared to a handcrafted parser, a rule-based recovery specification is much easier to maintain, especially as languages are changed or reused to build new languages. Another way to improve error recovery is through grammar analysis, such as LPG's scope detection [5].

In previous work we introduced an approach to error recovery that derives properties from grammars to produce explicit, customizable recovery rules [12]. Using scannerless generalized-LR (SGLR) parsing, the approach supports languages with a complex lexical syntax, such as AspectJ [3], and language embeddings and extensions, such as the Stratego program transformation language with embedded Java fragments [30]. Using generalized parsing, SGLR can parse ambiguous grammars. By considering the different ambiguous meanings of a syntactically incorrect program, through inspection of an expanding search space for applying the set of recovery rules, the approach can provide recovery suggestions that local recovery methods cannot.

An open problem we identified with our approach is that some search space-based suggestions are too "creative" and not natural (i.e., as a programmer would suggest them) [12]; in some cases it is simply better to ignore a small part of the input file, rather than to try and fix it using a combination of insertions and discarded substrings. Another open problem is that for tight clusters of errors, it is not always feasible to provide good suggestions in an acceptable time span.

In order to provide better, more natural suggestions, the present paper proposes an approach to identify the region in which a parse error is found. By restricting the search space for applying the recovery rules to this region, it becomes much less likely that the user is presented with "creative" suggestions that are nowhere near the original error. Using a smaller search space also helps performance. To further help performance, we add a form of "panic mode" [7]: if no solution of applying the recovery rules is found within an acceptable time span, the entire region can be skipped and marked as erroneous. This way, the parser can still continue, report other errors, and construct a partial AST.

We select erroneous regions based on indentation usage. Using indentation, programs typically form logical, nested regions of code. The approach of using layout information for partitioning files has been inspired by the technique of bridge parsing [20]. Bridge parsing is a supplementary technique to grammar-based error recovery. It uses structural information, such as typical use of indentation for bracket placement, to improve recovery quality. To further improve the quality of recovery suggestions, we adapted the bridge parsing approach to be usable with an SGLR parser.

We have identified and focus our paper on two open issues with error recovery for generated parsers. The first is the quality of corrections, which is often lacking since a generic solution is not aware of the semantics or typical structure of a language. The second is that given high-quality recovery, a good balance with the performance of error recovery must be maintained. To address these issues, this paper provides the following contributions:

- The use of layout to select regions of code that enclose a syntax error. These can be analyzed in detail by a secondary strategy, or discarded if no recovery is found within an acceptable time span.

- The application of bridge parsing based on a context-free (tokenizer) grammar rather than a scanner, showing how bridge parsing can be integrated into a parser rather than used as a preprocessor, improving results.

- The use of grammars for automatic construction of a tokenizer grammar and the heuristic derivation of a bridge parser specification.

We begin this paper with background on error recovery and setting out a number of requirements for good error recovery. In Section 3, we show how regions around a syntax error can be selected and used for coarse-grained error recovery. Section 4 describes how these regions can be used to apply recovery production rules. We refine error recovery for scopes based on bridge parsing in Section 5. Finally, Section 6 evaluates our approach and compares different configurations, using the Eclipse JDT parser as a baseline.

2 Error Recovery

Parsers serve two purposes: determining the grammatical structure of an input program, and syntactically validating it. Given the grammatical structure, the parser constructs an abstract syntax tree (AST), used for semantic analysis in tools such as compilers or editors. While performing syntactic validation, a parser also reports any errors that exist in the input.

A good parser does not only report the first character or token that is not valid according to the grammar, but also provides the user with a more sophisticated diagnosis. It can for example report missing constructs (e.g., "} expected here"). An even better parser also supports error recovery: based on the analysis of an error, it can recover from an error and continue parsing the rest of a file. Recovery techniques can be divided into correcting error recovery, which tries to transform the input string into a syntactically correct one, and non-correcting error recovery, which tries to continue the analysis by skipping parts of the input [7].

Error recovery plays an important role in modern, interactive development environments (IDEs). IDEs parse a file as it is typed in, making incomplete programs and syntax errors the common case rather than the exceptional one. Using error recovery, a parser can still construct a partial abstract syntax tree, allowing the IDE to perform semantic analysis and provide the user with interactive feedback (e.g., error marking, content completion).

In their comparative study, Degano and Priami [7] set out a number of quality criteria for good error recovery strategies, on which we will elaborate here. We distinguish between aspects that impact users and developers of a language. Firstly, there are three main criteria with respect to the end user's experience:

- Constructing a good AST: The recovered program should be as close to the program as intended by the programmer as possible. Since the AST is used for syntactic and semantic editor services in the IDE (e.g., the outline and error markers), the quality of the reconstructed AST is of great importance for the user experience.

- Providing good feedback: The parser should provide the user with good suggestions of how to fix the program. Spurious error messages should be avoided; instead, a small number of natural suggestions should be reported.

- Delivering adequate performance: For interactive use, the error recovery mechanism must not incur an unacceptable overhead. As their last criterion, Degano and Priami have suggested to only take performance degradation into account only if greater than a fixed maximum value.

Important criteria for developers of a language or an IDE (plugin) are:

- Flexibility: The approach must be easily adaptable to language insights and language changes.
- Language independence: an error recovery algorithm should be independent of a particular language. It should be usable with any given grammar, without introducing a prohibitive amount of work.
- Transparency: it should be clear why a particular recovery is presented. The grammar engineer should have insight into how the recovery works for a given grammar.

3 Coarse-grained Error Recovery

A parser that supports error recovery typically operates by consuming tokens (or characters) until an erroneous token is found. At the point of detection of an error, the recovery mechanism is activated. Simple, local approaches to error recovery will then attempt to make a modification to the input so that at least one more original symbol can be parsed [7]. For most cases, this works quite well. There are cases, however, particularly for complex languages, where these algorithms choose a poor repair that leads to further problems as the parser continues ("spurious errors").

Spurious errors are the result of one of the major problems in error recovery: the difference between the point of detection and the actual location of an error in the source program [7]. In contrast to local methods, global recovery methods examine the entire program and make a minimum of changes to repair all syntax errors [2, 17]. While these give the "best" repair, they are not efficient.

An alternative approach to local or global recovery is to consider only the direct context of the error, by identifying the region of code in which the errors reside [16, 18, 21]. Using regions for error recovery has three main advantages. Firstly, they reduce the search space for a recover algorithm. Secondly, they constrain the recovery suggestions to a particular part of the file, avoiding suggestions that are spread out all over the file. And thirdly, they can be used as a secondary recovery strategy [7], i.e. erroneous regions can be discarded entirely if a detailed analysis of the region does not provide a better recovery solution.

3.1 Nested Structures as Regions

Code constructs such as "while" statements and method bodies form good regions for regional error recovery. They form free standing blocks, in the sense that they can be omitted without influencing the interpretation of other blocks. Erroneous

```
class X {
   int i;

   void methodX(){
      i=1;
      if(true){
         foo();
         bar();
      }
      m();
   }
}
```

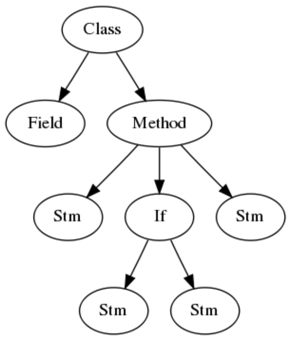


Figure 1: Indentation closely follows the hierarchical structure of a program.

free standing blocks can simply be skipped, providing a coarse recovery that allows the parser to continue. A typical technique to select such regions is to look for certain marker tokens in the context of an error, such as the fiducial tokens of Pai and Kieburtz [21]. These tokens depend on the language used. For example, for Java, keywords such as `class` and `while` could be used. We will take a more language-independent approach in this paper.

The method presented in this section is based on the use of indentation to detect code constructs. Indentation typically follows the logical nesting structure of a program, as illustrated in Figure 1. The relation between constructs can be deduced from the layout. An indentation shift to the right indicates a parent-child relation; the same indentation indicates a sibling relation.

Indentation usage is not enforced by the language definition. Proper use of layout is a convention, being part of good coding practice. We generally assume that most programmers apply layout conventions, but should keep in mind the possibility of inconsistent indentation usage.

Proper recognition of nesting structures prevents bad recoveries, obtained by merging structures that do not belong together. Figure 2 illustrates this idea with the example of a method that is missing a closing brace. The parser tries to parse the method header of the second method as a statement, which leads to a failure at the open brace in the method header. Indentation suggest that both methods should be considered as separate constructs. An indentation-based region selector will detect the erroneous if-block; which leads to the recovery presented in the middle part of the figure. An inferior recovery would be obtained by removing tokens surrounding the error detection point. The example at the right shows the result, merging the erroneous method with the correct method.

3.2 Indentation-based Region Selection

We follow an iterative process to select an appropriate region that encloses a syntax error. Each iteration, a different *candidate region* is considered. This candidate is

```
class X {

   int methodX(){
      if(true){
         foo(); //}
      return 5;
   }

   void methodY(){
      int i=5;
      bar(i);
   }
}
```

```
class X {

   int methodX(){


      return 5;
   }

   void methodY(){
      int i=5;
      bar(i);
   }
}
```

```
class X {

   int methodX(){
      if(true){
         foo(); //}
      return 5;
   }


      int i=5;
      bar(i);
   }
}
```

Figure 2: Erroneous code (left), discarded erroneous region (middle), and merged constructs (right)

then either validated or rejected; in case of a rejected candidate, another candidate is considered. We show example scenarios in Figure 3.

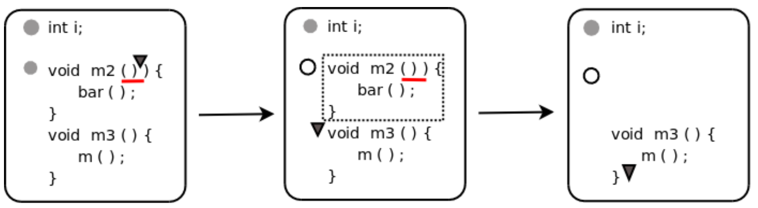
Figure III.3(a) shows a syntax error and the point of detection, indicated by a triangle (left figure). A candidate region can be selected based on the alignment of the **`void`** keyword and the closing bracket (middle figure). The candidate is then successfully validated by discarding the region, and attempting to parse the remainder of the file (right figure). After validation, the parser can be reset to its previous state (indicated by the circle, which represents a choice point for the parser). A detailed analysis of the region may be used to attempt to repair the erroneous region, as we will see in the following sections.

Figure III.3(b) illustrates a rejected candidate region. Based on the point of detection, an obvious candidate region may be the `m2` method (middle figure). However, an attempt to parse the construct that follows it leads to a premature parse failure; the region is rejected. Figure III.3(c) revisits the example. Another candidate region is selected, this time one preceding the point of detection. This region is successfully validated.

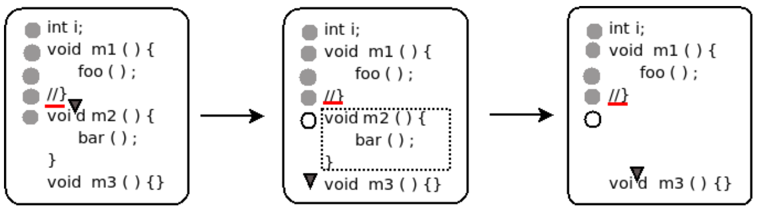
The region validation criteria should balance the risk of selecting the wrong candidate, which may lead to spurious errors, and the risk of rejecting a correct candidate region. The latter typically occurs in the context of multiple errors, in which a new, unrelated error causes the parser to fail again. Both cases lead to large regions, which should be avoided. We currently consider a region valid if the two lines of code succeeding it parse correctly, which has shown good practical results.

Selection Schema

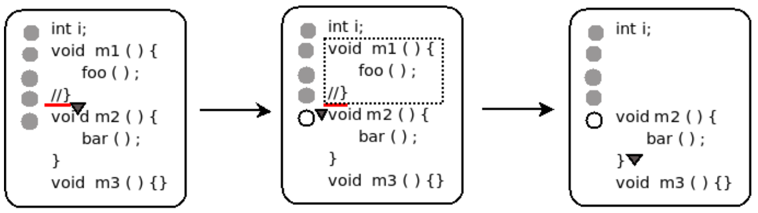
The candidate regions are explored in an ordered fashion, with the aim to find the smallest fragment enclosing the error first.



(a) A candidate region is validated and successfully discarded.



(b) A candidate region is rejected.



(c) An alternative candidate region is validated and successfully discarded.

Figure 3: Recovery by discarding of regions

```
while(true)){
    foo();
}
```

Figure 4: Extra)

Current structure The first candidate region is the construct starting from the error detection location. The region is recognized by a forward skip until the end of the construct is found. The construct ends with the last child (more indentation), including the closing bracket after the last child (same indentation). In Figure 4, the parser fails after reading the mistakenly inserted second brace. Discarding the entire `while` statement resolves the error.

Previous structure The second candidate is the structure preceding the error detection location. The region is detected by a backwards skip, using the indentation information stored in the choice points. Typical problems that are solved by discarding the previous structure are uncompleted lines and scope errors caused by a missing closing brace. The error in Figure 5 is detected after the `bar();` statement, while the preceding line caused the error.

```
void methodX() {
    foo(
    bar();
}
```

Figure 5: Missing);

```
if(true){
    foo();
else
    bar();
```

Figure 6: Missing }

Siblings Regions that are mutually dependent should be discarded as a whole. A typical example is provided in Figure 6. The unclosed "then" clause cannot be discarded, because the "else" clause cannot exist in isolation. The "sibling-procedure" deals with this situation. The procedure starts with the current structure as discarded region. Then it successively includes the prior sibling and the next sibling, until a valid erroneous region is found or all siblings have been considered.

Parent The next region to consider is the parent structure, identified through a forward and backward search for a decrease in indentation. Identifying the parent structure can be useful when a child that is missing or erroneous. Parent child dependencies are rarely seen in common programming languages, but they can occur in DSLs. The example in Figure 7 shows a simple person data language with an error in the required field `email`. Apart from solving errors in parent-child dependencies, the parent selection scheme adds some robustness with respect to inconsistent indentation.

While the selection schemata have been designed to be generally applicable, the success of our approach depends on assumptions of indentation conventions and language characteristics. Conventions for widely used programming languages seem to meet the assumption that the indentation follows the logical nesting structure of the program. A more problematic issue is the (mis)use of indentation by programmers. Inconsistent indentation usage decreases the quality of the results, although some robustness for small deviations can be expected. The second assumption we make is that programs have free standing blocks, i.e. that

```
Person
    Name: John
    email: ????
```

Figure 7: Illegal email property

```
bar();
while(true)
    foo();
doX();

while(true)
{
    foo();
}
```

Figure 8: Different indentation styles

discarding a region still yields a valid program. Again, conventional programming languages seem to meet this requirement. However, some (declarative) languages use constructs that cannot be discarded because they are syntactically obligatory. Such languages can lead to large regions.

3.3 Implementation Considerations

We implemented the region selection method in SGLR, in order to use it in collaboration with recovery rules [12]. The selection method does not depend on specific features of generalized parsing and can be implemented in other LR parsers as well.

Layout conventions for braces Varying conventions for closing and opening braces are used. They can be omitted in some situations, besides the position of the opening brace can vary. The figure illustrates the problem with a concrete example. Two code fragments with the same indentation characteristics, have a different decomposition in regions.

The need to cover all different notations in a language independent way, has greatly increased the complexity of the implementation. A simple solution would be provided by an explicit recognition of those tokens. This would make the algorithm more precise and the implementation straight forward. However, this will introduce a language dependency. We have chosen to stick with our language independent approach, and deal with the support for various brace-conventions in the code. In case of doubt, we assume the notation including the braces on separate lines. The main disadvantage is that sometimes one or two correct lines are included in the selected region.

Whitespace parse A simple discarding of erroneous regions will offer a recovery and allows the parser to continue the analysis, however the information about line and column numbers may be lost. This will cause problems in the scenario of interactive editor support. A simple solution is offered by whitespace

```
class X{
    ....
    void methX(){...}
    void methY(){...}
    void methZ(
```

Figure 9: An uncompleted class

parsing. All symbols, except newlines and tabs are parsed as whitespace. Information about skipped regions can be used to generate error messages.

Parse tree completion We maintain only a limited number of choice points to backtrack to, to ensure that there is only a negligible overhead when parsing (parts of) files without errors. This limitation means that in some cases the layout-based region selection cannot provide a candidate region. For example, the class construct in Figure 9 is unfinished, and can only be discarded as a whole. Complementary to the region selection schemata, we implemented a technique that completes the parse tree for an unfinished code fragment. In this way, at least the already recognized part of the code can be reported to the programmer. A program prefix missing only a few closing tokens at the end, can be completed to a valid program by inspection of the parse table. Although the missing next token is not known, a list with possible tokens can be retrieved from the parse table. The completion method creates separate stack branches for each possible "next state", deduced from the list of possible tokens. After a number of parse steps using this branching mechanism, an accepting state will be found. Generalized parsers like SGLR and GLR provide native support for branching. The method works efficient if only a few general branching steps are required, which corresponds to a small number of missing tokens. Therefore, we apply the method on the location of the last big reduction, the closing brace of `methY`, in the example.

4 Fine-grained Error Recovery

We can improve upon the coarse-grained recovery approach by using it in conjunction with a more fine-grained, correcting error recovery method. In this section we outline how the error recovery productions of [12] can be used to perform fine-grained error recovery inside erroneous regions.

Error recovery productions allow for a high-level, grammar-oriented way of customizing a recovery strategy [2, 10]. Because the language engineer must design them a priori, they have sometimes been criticized for being language-dependent [7]. In [12] we introduced a way to derive recovery rules from a grammar, and added general rules that can simply skip over erroneous code fragments.

Following [12], recovery productions are written just as any other production, annotated with the `{recover}` annotation. We use the flexible SDF syntax definition formalism [29] for the specification of grammars and their recovery rules. As an example of an SDF production, consider the following Java production:

```
"{" BlockStm* "}" -> Block {cons("Block")}
```

This rule specifies that a { literal, followed by a list of `BlockStm` symbols and a closing } literal, can be parsed as a `Block`. The `{cons}` annotation specifies the name used for the node in the abstract syntax tree. Based on this rule (and taking global properties of the grammar into consideration, as outlined in [12]), the following recovery production rule can be derived:

```
-> "}" {recover, cons("INSERT")}
```

This production specifies that a possible recovery is to parse the empty string (hence the empty left-hand side) instead of the closing } literal. Annotated `{recover}`, this *insertion recovery rule* is only used when recovery is required.

In addition to insertion recovery rules, [12] also specifies lexical “catch-all” production rules to discard unparsable substrings. Together, these rules could parse any string, distinguishing only “words” and “separators”:

```
[A-Za-z0-9\_]*          -> WATERWORD {recover}
~[A-Za-z0-9\_\ \t\r\n] -> WATERSEP  {recover}
```

Each application of recovery rule incurs a cost of 1. A minimal-cost solution may be the best possible match with the programmer’s intention. However, considering all candidate recoveries for a complete file results in a search space that is too large to inspect within reasonable time. In [12] we applied an unbounded, expanding search space to discover a recovery solution with a minimum cost. For most cases, this approach is effective, but for a number of pathological cases, the unbounded search leads to unacceptable recovery times, or to far-fetched, non-local recovery suggestions.

By restricting the search space to a selected region of code, recovery performance and locality can be improved. Smaller regions (fewer than four lines) are reparsed, applying a bounded number (three in the current implementation) of recover productions. For larger regions, we assume that the error can be corrected in the three lines nearest to the parse failure, which seems to be the case in most practical examples. If the application of recovery rules does not lead to a successful repair, the entire region can be discarded using the whitespace parse approach discussed in the previous section.

5 Bridge Parsing

One of the most common errors made by programmers is omitting closing brackets of scopes, since scopes are recursive structures need to be properly balanced [5]. A parser can recover in these cases by inserting the missing braces. Unfortunately, there are often many possible locations where a missing brace can be inserted. Consider for example the Java fragment of Figure 10. This fragment might be recovered by inserting a closing brace at the start of line 2, 3, or 4. However, the

```
class C {
  void m() {
    int y;
  int x;
}
```

Figure 10: Missing }

use of indentation suggests the best choice may be just before the `int x;` declaration. Bridge parsing [20] provides an algorithm to improve error recovery based on indentation. Provided with knowledge of typical usage in Java programs, it can correctly recover cases such as the example above. It can be configured to work for any given language, and works independently of a particular parser technology.

Inspired by island grammars [19, 28], a bridge parser employs a scanner that only recognizes tokens that make up scoping structures ("islands") and important tokens for determining how those islands should be connected ("reefs"). All other tokens ("water") are skipped. Given a list with these kind of tokens, the bridge parser constructs a bridge model, which captures the scopes in the input. A scope in this context corresponds to two islands connected with a bridge. Two islands will only match if a pre-defined set of conditions is fulfilled. Missing bridges in the bridge model reveal broken scopes. They can be repaired by locating an appropriate "construction site" for inserting a new, artificial island, matching the island in need of recovery. A new bridge can then be constructed. An algorithm for incrementally constructing multiple bridges is given in [20].

5.1 Scannerless Bridge Parsing

Composed languages and languages with a complex lexical structure (such as AspectJ) cannot or can only with great difficulty be parsed using a separate scanner [3]. For example, the scanner for the Java language recognizes `enum` as a keyword. This means that it can never be parsed as an identifier. When Java is extended or composed with another language, this restriction also applies for the combined language. Using the same scanner, a composition of Java and SQL cannot parse programs where `enum` is an SQL identifier. Using scannerless parsing [24], these issues can be elegantly addressed [3].

Since bridge parsing as presented in [20] is based on the notion of a scanner, it cannot support languages that depend on scannerless parsing or parsing with a context-sensitive scanner. Still, bridge parsing only depends on a small set of tokens, such as brackets and keywords, not on a full scanner definition. So why can we not just construct a scanner for those literals in the grammar? The problem is that each sequence of characters, there can potentially be many different lexical and literal interpretations. Again consider `enum`, which is keyword in Java, but may also be an identifier in the composed Java-SQL or Stratego-Java languages.

```
module Java-SQL-Tokenizer

context-free start-symbols
  Class Stm Expr ...

context-free syntax %% token list definitions for all start symbols
  ClassToken Class -> Class {cons("Cons")}
                   -> Class {cons("Nil")}

context-free syntax %% tokens and the {cons} name of their production
  EnumDecHeadToken -> ClassToken {cons("EnumDecHead")}
  SQLId            -> ClassToken {cons("Id")}

lexical syntax %% lexical token definitions
  "enum"     -> EnumDecHeadToken
  [A-Za-z]+ -> SQLId
```

Figure 11: A (partial) generalized tokenizer definition for the Java-SQL language.

To overcome the difficulties of a scanner-based approach, we introduce the notion of a *generalized tokenizer*. This tokenizer constructs all possible token interpretations, forming an ambiguous token stream. We implement this tokenizer based on the grammar of a language. For example, given the Java-SQL definition, we mechanically strip all context-free productions and retain only definitions for literals and lexical symbols. For each sort in the grammar, we then generate a start symbol that parses the different lexicals and literals reachable from that state. A (partial) tokenizer grammar for Java-SQL is illustrated in Figure 11. Using the `Class` start symbol, this grammar constructs the following token stream (a list of `Cons` and `Nil` nodes) for the string `enum Color{}`:

```
[ amb([EnumDecHead("enum"), Id("enum"])
, LAYOUT(" "), Id("Color")
, amb([EnumBody("{"), ClassBody("{"), Block("{"), ...])
, amb([EnumBody("}"), ClassBody("}"), Block("}"), ...])]
```

where ambiguities in the token stream are indicated with an `amb` term. For composed languages, these token streams quickly grow more complex as the number of different token interpretations increases.

We simplify the token stream by considering only those tokens that are of interest to the bridge parser, and by flattening the ambiguities to create multiple, possible interpretations that have no deep ambiguities. After that, the bridge parser can assign different island classes, reef classes, and water to the tokens:

```
[KeywordReef("enum"), LayoutReef(" "), Water("Color"), LBrace, RBrace]
```

In this list, the `LBrace` and `RBrace` classes encompass all interpretations for the { literal. In the remainder of this section we will discuss how the binding between these tokens and the bridge parsing island classes is specified.

5.2 The Bridge Parsing Specification

A bridge parser is generated from a bridge parser specification (or bridge grammar) [20]. It defines all islands, reefs, and rules for matching and recovering islands. Attributes can be added to islands and reefs to help with matching in rule

```
grammar Layout;

abstract island LayoutStart;
abstract island LayoutEnd;
abstract reef Layout;

bridge from LayoutStart to LayoutEnd;

attr Layout LayoutStart.indent =
    [first left Layout];
attr Layout LayoutEnd.indent =
    [first left Layout];
java-attr int Layout.pos =
    ...embedded java code...

recover LayoutStart find [a:Layout]
where (a.pos <= this.indent.pos)
insert bridge-end before a;
...
```

Figure 12: A generic bridge grammar.

```
grammar SimpleJava;

import Layout;

island LBrace : LayoutStartIsland = "{"
  for-sglr("EnumBody"|"ClassBody"|...);
island RBrace : LayoutEndIsland = "}"
  for-sglr("EnumBody"|"ClassBody"|...);
reef Indent : LayoutReef =
  NEWLINE|(WS|TAB)+

bridge from LBrace to RBrace;
...
```

Figure 13: A bridge grammar for Java.

expressions. These specifications are composable and can be extended in several steps. Generic behavior such as “closest match recovery” or “layout-based recovery” is defined in a generic specification that can be reused and redefined by other grammars. Figure 12 lists parts of a generic bridge grammar that specifies layout-based recovery. The grammar specifies abstract `LayoutStart` and `LayoutEnd` islands that must be connected by a bridge. It also adds `indent` and `pos` attributes that can be used to do layout-based matching, as described in [20]. The **`recover`** rule uses these attributes to construct an artificial `LayoutEnd` island to repair bridges from `LayoutStart` islands.

We extended the bridge parser specification language for the purpose of integrating it with SGLR, adding a new **`for-sglr`** clause to capture the different possible interpretations for one token or character. For example, braces in Java have several possible interpretations according to the SGLR tokenizer grammar. Figure 13 shows how these can be captured using the **`for-sglr`** clause. The arguments of the clause correspond to the node types in the generalized token stream, seen in the previous subsection. The grammar in Figure 13 imports the generic layout grammar, and provides concrete implementations for the abstract islands and reef defined in that grammar.

Using the bridge grammar, the bridge parser derives a bridge model from the token stream (illustrated in the previous subsection). Because of ambiguous interpretations, there may be multiple possible token streams. By assigning island and reef classes that may encompass multiple node types, some of these can be eliminated. In case more than one alternative remains, we currently pick the interpretation with the fewest number of broken bridges.

5.3 Deriving Bridge Parser Specifications

SDF grammars are fully declarative, and do not allow semantic actions or callbacks to native code. This property makes SDF grammars well-suited for analysis. In previous work we applied automated analysis of SDF grammars to derive recovery productions [12]. To help language engineers efficiently employ bridge parsing with an SDF grammar, we do the same for a bridge parser specification.

Island definitions are central to the bridge parser specification. Typical candidates for island definitions are scoping constructs, such as `{ }` in curly brace programming languages. Scoping constructs are generally nestable structures, which means that their grammar productions are recursive. For example, scopes in Java are defined as follows:

```
"{" BlockStm* "}" -> Block {cons("Block")}
Block             -> Stm
Stm               -> BlockStm
```

We consider a production $\overline{p}\ \alpha\ \overline{q}\ \beta\ \overline{r}\ \text{->}\ S$ to define a scoping construct for opening literal α and closing literal β to form a scoping construct if the following conditions are satisfied:

- the production is recursive;
- literals α and β are not identical;
- literals α and β appear only in productions of the form $\overline{p}\ \alpha\ \overline{q}\ \beta\ \overline{r}\ \text{->}\ S$ where α and β are not part of patterns $\overline{p}$, $\overline{q}$, or $\overline{r}$;
- the literals do not appear in a production with the `{bracket}` annotation.

The second condition excludes literals such as ` in shell scripts, since they are typically not nestable. The third condition ensures that we only select literals that appear in a balanced fashion throughout the grammar, ensuring that the bridge parser does not try to introduce opening or closing literals for unbalanced literals. The final condition ensures that we do not select constructs that define parentheses. Unlike scopes, parentheses (marked with the `{bracket}` annotation in SDF) have no direct semantic meaning other than modifying the priority of other operators. Because of this property, parentheses are typically not indented the same way as scopes.

For each opening and closing literal, we generate island definitions and bridge rules similar to those in Figure 13. To complete the bridge parsing specification, we also generate reef rules for all reserved words in the language. Reserved words in SDF are defined using a `{reject}` annotation that indicates they cannot be used as an identifier. For composed languages where these words may not be globally reserved; the bridge parser then considers both interpretations.

Automatically deriving recovery rules helps maintain a valid, up-to-date recovery rule set as languages evolve and are extended or embedded into other languages. Grammar engineers may also customize the derived specification to handle further cases and to introduce different indentation styles.

5.4 Combining Fine-grained Error Recovery and Bridge Parsing

Bridge parsing excels at correcting scope errors, while fine-grained recovery is the designated approach to recover more localized errors like a missing semicolon. In case the erroneous region contains both types of errors, a combination of both techniques is required to find an optimal recovery. To do this, we extend the fine-grained recovery process to handle suggestions provided by the bridge parser. Each suggestion gives rise to an extra stack branch that is explored in parallel with the other recovery branches. In this way, the bridge parser suggestions are taken into account, but only applied if they lead to a least-cost recovery.

6 Evaluation

We implemented our approach based on JSGLR, a Java implementation of SGLR [11], extending it with support for coarse-grained recovery and refining the support for recovery rules of [12]. The bridge parser implementation, also written in Java, is based on the implementation of [20], and adapted to support ambiguous token streams and recovery of regions rather than complete files. We evaluate our error recovery according to the criteria set out in Section 2. We study the quantitative criteria through evaluation of the parser using a set of test files written in Java. Java was selected because of its ubiquity in software development and in modern IDEs such as the Eclipse JDT, offering a challenging comparison. We will also argue that our approach satisfies the qualitative criteria of providing good feedback, flexibility, language independence, and transparency.

Construction of the Test Set We evaluate using an extended version of the test set used in [20]. The base test set was originally constructed for testing structural recovery of Java code, and focused on syntax errors such as missing braces pr parenthesis. The extended test set includes tests for both structural and non-structural errors, and is available from [1]. We intentionally included some cases with inconsistent use of indentation, since those are difficult to handle for the bridge parser, i.e. basic layout recovery depends on good indentation information. The test set contains three major categories of tests; *missing* – structural tokens for grouping, closure or division are missing (65 tests), *extra* – there are too many structural tokens (8 tests), and *other* – remaining errors like erroneous statement, or missing comment end (3 tests). Together, these total to a set of 76 test cases.

Setting up the Experiment All tests are run in an automated fashion, comparing the pretty-printed ASTs for the erroneous files to the pretty-printed ASTs for the original, correct files they were derived from. We use two methods for comparison: First, we do a manual inspection, following the quality criteria of

Pennello and DeRemer [22]. Following these criteria, an *excellent* recovery is one that is exactly the same as the intended program, a *good* recovery is close to this result, and a *poor* recovery introduces spurious errors. Since this is arguably a subjective comparison, we also count the number of lines of code that changed in the recovered result (the “diff”). The advantage of this approach is that it is objective, and assigns a larger penalty to recoveries where a larger area of the text does not correspond or is placed in an incorrect scope. The resulting figures are also arguably easier to interpret than comparing tree distances.

Various Approaches We compare the integrated recovery approach presented in this paper to different configurations of the individual techniques and to the parser used by Eclipse’s JDT. We apply the test set with the following parser configurations; the JDT parser; the JDT parser with a bridge parser (BP) preprocessor, as suggested in [20] (BP$\rightarrow$JDT); our approach without using bridge parsing (Course Grained (CG) $\rightarrow$ Fine Grained (FG)); our approach with the bridge parser as a preprocessor (BP$\rightarrow$CG$\rightarrow$FG); the fully integrated approach (CG$\rightarrow$BP+FG); and finally the same approach with a tuned bridge parser specification (CG$\rightarrow$BP$_{tun}$+FG).

Except for the final configuration, the three SGLR-based parsers use fully automatically derived recovery specifications. In contrast, specialized, handwritten recovery rules and classes related to recovery are used for the JDT parser. For the tests we used the JDT parser with statement-level recovery enabled, following [15]. In some of the test cases, particularly those with multiple errors, the parser was unable to recover the entire body of a method. For content completion, Eclipse uses a secondary parser that can analyze these method bodies. Because of its specialized nature, we have not included it in our experiments. Both the bridge parser used as a preprocessor and the one integrated into SGLR use the same recovery rules and node types.

Results

The diff values acquired for the various approaches are shown in Figure 15 and the same values with a quality distinction are shown in Figure 14. Considering both diagrams, we see that the SGLR parser, parsing using different steps and granularity, consistently outperforms the JDT parser. When fully integrated with the bridge parser, the best results are obtained.

Using the bridge parser in a preprocessor setting was shown to be effective for a number of different parsers in [20]. For the JDT parser, we can see that the results are improved using the bridge parser as a preprocessor. When combined with SGLR, however, we see that the preprocessing approach does not work well. We speculate that these results arise because the bridge parser can only insert braces to recover scopes, never remove them, since it does not have enough knowledge of the complete language. However, when it is actually integrated into SGLR,

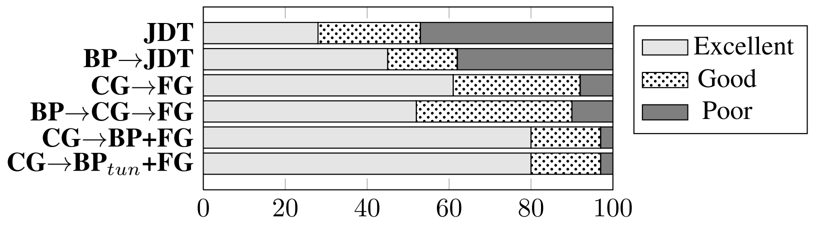


Figure 14: Quality of Recovery The x axis shows percentage of tests. Each bar shows the percentage of recoveries which where excellent, non-excellent or failed for each approach. **CG** - Coarse Grained, **FG** - Fine Grained, **BP** - Bridge Parsing, **JDT** - Java Developer Toolkit

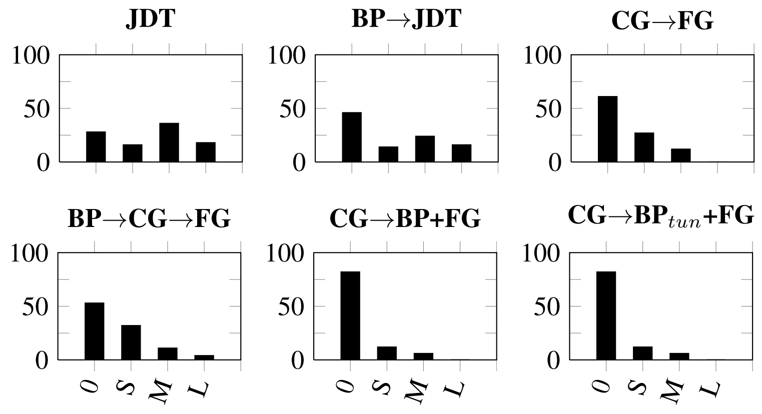


Figure 15: Diffs for Various Approaches The y axis shows percentage of tests and the y axis shows categories of number of diff lines – No diff (0), **S**mall diff $(1 - 10)$, **M**edium diff $(11 - 20)$ and **L**arge diff (> 20). **CG** - Coarse Grained, **FG** - Fine Grained, **BP** - Bridge Parsing, **JDT** - Java Developer Toolkit

the bridge parser's suggestions lead to the best results. Manual inspection of the non-excellent results for each approach reveals more in-depth knowledge:

- **JDT** (49 *missing*, 4 *extra*, 2 *other*): A majority of the cases are in the missing category. The most common recovery is for JDT to skip the whole content of a block if there is an error. This explains why the diff values for JDT tend to be higher.

- **BP→JDT** (34 *missing*, 5 *extra*, 2 *other*): The bridge parser helps to reduce the number of cases in the *missing* category. However, it fails to improve cases which are of out of scope for the bridge parser, for example, missing semicolons or extra structural tokens.

- **CG→FG** (24 *missing*, 4 *extra*, 3 *other*): Also has a majority of cases in the *missing* category, particularly missing braces (both start and end).

- **BP→CG→FG** (27 *missing*, 5 *extra*, 3 *other*): This combination does not work out very well. The bridge parser manages to slightly improve cases in the *missing* category, but makes things worse in some of the other cases.
- **CG→BP+FG** (8 *missing*, 3 *extra*, 3 *other*): This is the best option. The robustness of SGLR evens out the rough edges of the bridge parser, using it more like a consultant and discarding bad advice. In practice, this means that tests in the *missing* category see a huge improvement. There is a slight improvement in the *extra* category, while the others categories stay the same.
- **CG→BP**$_{tun}$**+FG** (8 *missing*, 3 *extra*, 3 *other*): There are no visible changes using this tuned bridge parser. The partial recoveries performed by the bridge parser show a small improvement, i.e., if there is more than one error one of the two gets a better recovery but the end result is the same.

Our experiments have not indicated that using a "tuned" bridge parser specification helps results. Tuning in this context can be quite tricky due to the various uses of, for example, a keyword. Turning all keywords into reefs will potentially ruin recoveries. For example, considering a `for` loop missing a left parenthesis, with too many keywords defined as reefs, the bridge parser might insert a left parenthesis too early. If both `int` and `for` are keywords and there is a rule stating that a recovery shall not pass a reef, then the left parenthesis will be inserted before the `int` and not before the `for`. This indicates that keywords must be chosen with care and the set should probably be quite small.

Concerning the selection of error fragments by the coarse grained recovery approach, manual inspection revealed that the right segment is identified for most of the test cases – both in position and size. Generally, however, there can be cases where selecting the right error fragment is difficult, which can result in a poor recovery.

While we have not performed an in-depth performance study, we set a maximum of 1 second for completing each test run, to allow for good responsiveness when used in an interactive environment (where the parser runs in a background thread). All tests complete within this time limit. The pathological cases previously identified for the Stratego-Java language [30] used to take much longer than this time limit [12], but with the addition of the coarse-grained recovery mechanism now also complete within this time limit. By constraining the expensive fine-grained recovery rules to a small region, setting an upper bound for the number of cases to consider per region, and introducing the possibility to fall back to discarding an entire region, the performance issues seem to have been resolved.

The Impact of Indentation Usage Since our approach depends on layout, one issue to address is robustness in case of inconsistent indentation. The tab size used greatly affects the indentation levels in a file. The tab size might change, and tabs and white spaces are often mixed. IDEs such as Eclipse can automatically insert spaces for tabs and maintain indentation settings per project, avoiding some of these problems. Possible strategies for more robustness are: 1) Using averages to determine the indentation shift, and in that way handle different indentation shifts within a file or project. 2) Rounding off exact indentation offsets to their approximate indentation level. Some times the exact indentation position has an off-by-one position, e.g., there might be three spaces when the indentation shift is four. This situation can cause indentation matching problems. The two strategies can be combined, normalizing the indentation levels to match the indentation shift in the rest of the file or fragment.

Qualitative Evaluation When working with interactive parsing the most important thing is to provide a good service to the user. We integrated our approach into Eclipse based on the Spoofax/IMP editor environment [13]. Based on the recovery productions, the editor gives accurate feedback. Following [12], every class of recovery rule is associated with a particular message (e.g., "} expected").

For the language engineer, flexibility, language independence, and transparency of the approach are important qualitative criteria. Our approach is highly flexible as it allows for customization of the high-level bridge parsing and recovery rules specifications. Yet, it maintains language independence by deriving defaults for these specifications, ensuring it is in line with the expectations of parser generators. By deriving explicit, customizable specifications, the approach is also highly transparent.

7 Related Work

In previous work, we introduced error recovery for SGLR, based on parse error productions that can be automatically derived from a grammar [12], and described its integration in Spoofax/IMP [13]. The present paper refines this work, constraining the application of recovery rules to coarse-grained regions and adding support for bridge parsing.

Bridge parsing was previously applied purely as a preprocessor for other parsers [20], ensuring that it repaired scope-related errors before other errors are recovered. We found that this approach was ineffective in combination with the production-based recovery approach of SGLR (see Section 6). Furthermore, using a scanner, the bridge parser was unable to cope with the lexical complexity of composed languages. The present work introduces a scannerless tokenizer and fully integrates the bridge parser into SGLR to address these issues.

Using SGLR parsing, our approach can be used to parse languages with a complex lexical syntax and composed languages. In related work, only a study by Valkering [26], based on substring parsing [23], offered a partial approach to error recovery with SGLR parsing. Composed languages are also supported by parsing expression grammars (PEGs) [9]. PEGs lack the disambiguation facilities [29] that SDF provides for SGLR. Instead, they use greedy matching and enforce an explicit ordering of productions. To our knowledge, no automated form of error recovery has been defined for PEGs. However, existing work on error recovery using parser combinators [25] may be a promising direction for recovery in PEGs. Furthermore, based on the ordering property of PEGS, a "catch all" clause is sometimes added to a grammar, which is used if no other production succeeds. Such a clause can skip erroneous content up to a specific point (such as a newline) but does not offer the flexibility of our approach.

There are several different forms of error recovery techniques for LR parsing [7]. These techniques can be divided in *correcting* and *non-correcting* techniques. The most common non-correcting technique is *panic mode*. On detection of an error, the input is discarded until a synchronization token is reached. Then, states are popped from the stack until the state at the top enables the resumption of the parsing process. Our coarse-grained recovery algorithm can be used in a similar fashion, but selects discardable regions discarded based on layout.

Correcting recovery methods for LR parsers typically attempt to insert or delete tokens nearby the location of an error, until parsing can resume. Successful recovery mechanisms often combine more than one technique [7]. For example, panic mode is often used as a fall back method if the correction attempts fail.

Burke and Fisher [4] present a method based on three phases of recovery. The first phase looks for simple correction by the insertion or deletion of a single token. If this does not lead to a recovery, one or more open scopes are closed. The last phase consists of discarding tokens that surround the parse failure location. We improve on their work by taking indentation into account, for the scope recovery using an adapted version of bridge parsing [20], as well as for the coarse recover technique. In addition, by starting with region selection, the performance as well as the quality of the fine-grained technique [12], is improved.

Regional error recovery methods [16, 18, 21] select a region that encloses the point of detection of an error. Typically, these regions are selected based on nearby marker tokens (also called fiducial tokens [21]), which are language-dependent. In our approach, we assign regions based on layout instead.

The LALR Parser Generator (LPG) [5] is incorporated into IMP [6] and is used as a basis for the Eclipse JDT parser. LPG can derive recovery behavior from a grammar, and supports recovery rules in the grammar and through semantic actions. Like our approach, LPG detects scopes in grammars. However, unlike our approach, it does not take indentation into account for scope recovery.

8 Conclusion

Source code has a hierarchical structure that generally is reflected in the usage of layout and indentation. We have shown that this property can be exploited to confine syntax errors to small regions of code, and to provide better, more natural error recovery suggestions. Our approach to error recovery provides language independence by automatically deriving language-specific recovery behavior from grammars. Yet by allowing customization of the recovery behavior, using fine-grained recovery rules and a high-level bridge parsing specification, the approach maintains flexibility.

Bibliography

[1] The permissive grammars project. `http://strategoxt.org/Stratego/PermissiveGrammars`.

[2] A.V. Aho and T. G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1:305, 1972.

[3] Martin Bravenboer, Eric Tanter, and Eelco Visser. Declarative, formal, and extensible syntax definition for AspectJ. A case for scannerless generalized-LR parsing. In William R. Cook, editor, *OOPSLA 2006*, pages 209–228. ACM Press, October 2006.

[4] Michael G. Burke and Gerald A. Fisher. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Trans. Program. Lang. Syst.*, 9(2):164–197, 1987.

[5] Philippe Charles. *A practical method for constructing efficient LALR(K) parsers with automatic error recovery*. PhD thesis, New York University, 1991.

[6] Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton, Jr. IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *Automated Software Engineering (ASE 2007)*, pages 485–488. ACM, 2007.

[7] Pierpaolo Degano and Corrado Priami. Comparison of syntactic error handling in LR parsers. *Software – Practice and Experience*, 25(6):657–679, 1995.

[8] S. Efftinge et al. openArchitectureWare User Guide. Version 4.3. Available from `http://openarchitectureware.org/pub/documentation/`, 2008.

[9] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *International Conference on Functional Programming (ICFP'02)*, volume 37 of *SIGPLAN Notices*, pages 36–47. ACM, October 2002.

[10] Susan L. Graham, Charles B. Haley, and William N. Joy. Practical LR error recovery. In *SIGPLAN '79: Symposium on Compiler Construction*, pages 168–175. ACM, 1979.

[11] Karl Trygve Kalleberg et al. JSGLR. `http://www.spoofax.org/`.

[12] Lennart C. L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco Visser. Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In Gary T. Leavens, editor, *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, volume 44 of *ACM SIGPLAN Notices*, pages 445–464, New York, NY, USA, October 2009. ACM Press.

[13] Lennart C. L. Kats, Karl Trygve Kalleberg, and Eelco Visser. Domain-specific languages for composable editor plugins. In T. Ekman and J. Vinju, editors, *Language Descriptions, Tools, and Applications (LDTA 2009)*, ENTCS. Elsevier Science Publishers, April 2009.

[14] H. Krahn, B. Rumpe, and S. Völkel. MontiCore: Modular development of textual domain specific languages. In R.F. Paige and B. Meyer, editors, *TOOLS EUROPE 2008*, volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315. Springer-Verlag, 2008.

[15] Thomas Kuhn and Olivier Thomann. Eclipse corner: Abstract syntax tree. `http://eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html`, 2006.

[16] Jean-Pierre Lévy. *Automatic Correction of Syntax Errors in Programming Languages.* PhD thesis, Ithaca, NY, USA, 1971.

[17] Gordon Lyon. Syntax-directed least-errors analysis for context-free languages: a practical approach. *Commun. ACM*, 17(1):3–14, 1974.

[18] J. Mauney and C.N. Fischer. Determining the extent of lookahead in syntactic error repair. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 10(3):456–469, 1988.

[19] Leon Moonen. Generating robust parsers using island grammars. In *Working Conference on Reverse Engineering (WCRE'01)*, pages 13–22. IEEE, Oct 2001.

[20] Emma Nilsson-Nyman, Torbjörn Ekman, and Görel Hedin. Practical scope recovery using bridge parsing. In Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering (SLE 2008)*, volume 5452 of *LNCS*, pages 95–113. Springer, 2008.

[21] A.B. Pai and R.B. Kieburtz. Global Context Recovery: A New Strategy for Syntactic Error Recovery by Table-Drive Parsers. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 2(1):18–41, 1980.

[22] Thomas J. Pennello and Frank DeRemer. A forward move algorithm for LR error recovery. In *Principles of programming languages (POPL '78)*, pages 241–254. ACM, 1978.

[23] Jan Rekers and Wilco Koorn. Substring parsing for arbitrary context-free grammars. *SIGPLAN Not.*, 26(5):59–66, 1991.

[24] D.J. Salomon and G.V. Cormack. The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical report, TR 95/06, Dept. of Comp. Sci., University of Manitoba, Winnipeg, Canada, 1995.

[25] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury et al., editors, *Advanced Functional Programming, Second International School*, volume 1129 of *LNCS*, pages 184–207. Springer, 1996.

[26] Ron Valkering. Syntax error handling in scannerless generalized LR parsers. Master's thesis, University of Amsterdam, Aug 2007.

[27] M. G. J. van den Brand, M. Bruntink, G. R. Economopoulos, H. A. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J. J. Vinju. Using the Meta-Environment for maintenance and renovation. In *European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 331–332. IEEE, 2007.

[28] Arie van Deursen and Tobias Kuipers. Building documentation generators. In *IEEE International Conference on Software Maintenance (ICSM '99)*, page 40. IEEE Computer Society, 1999.

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

[30] Eelco Visser. Meta-programming with concrete object syntax. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *LNCS*, pages 299–315. Springer-Verlag, October 2002.

INCLUDED TECHNICAL REPORTS

TECHNICAL REPORT I

AUTOMATED SELECTIVE CACHING FOR REFERENCE ATTRIBUTE GRAMMARS

Abstract

Reference attribute grammars can be used for compiler generation, and are based on dynamic attribute evaluation with caching. Practical performance can be improved by selecting to not cache certain attributes. We present a profiling-based technique for automatically finding a good caching configuration. The technique has been evaluated on a generated Java compiler, compiling programs from the DaCapo benchmark suite. Based on profiling of a single program in the suite, with only 50.2% attribute coverage, we obtained a mean compilation speedup of 23.5%, which is close to that obtained by a manual expert configuration (31.7%).

1 Introduction

Reference attribute grammars (RAGs) [9] have been shown useful for implementing extensible compilers of high quality and good performance [2, 7], and are being used in an increasing number of metacompilation systems [10, 13, 17, 19]. RAGs are based on dynamic attribute evaluation where attributes are evaluated

Emma Söderberg, Görel Hedin

Technical report, LU-CS-TR:2010-245, ISSN 1404-1200, Report 94, 2010

Lund University

on demand, and their values are cached (memoized) for obtaining optimal performance [11]. Caching has a cost in both execution time and memory consumption, and performance can therefore be improved in practice by *selective caching*, caching only a subset of the attributes, a *caching configuration*.

But determining a good caching configuration is not easy to do manually. It requires a good understanding of how the underlying attribute evaluator works, and a lot of experience is needed to understand how different input programs can affect the caching inside the compiler. Ideally, the compiler developer should not need to worry about this, but let the system compute the configuration automatically.

To help solve this problem, we have developed a technique for automatically computing a caching configuration, based on profiling. We have evaluated the approach experimentally on a generated compiler for Java [7]. This compiler is implemented using JastAdd [8], which is a metacompilation system based on RAGs.

The rest of this paper is structured as follows. Section 2 gives background on reference attribute grammars and their evaluation, explaining the JastAdd caching scheme in particular. Section 3 introduces the concept of an AIG, an attribute instance graph with call information, used as the basis for the caching analysis. Section 4 introduces our technique for computing a cache configuration. Section 5 presents an experimental evaluation of the approach. Section 6 discusses related work, and Section 7 concludes the paper with a discussion and future work.

2 Reference Attribute Grammars

Reference Attribute Grammars (RAGs) [9], extend Knuth-style attribute grammars [14] by allowing attributes to be references to nodes in the abstract syntax tree (AST). This is a powerful notion because it allows the nodes in an AST to be connected into the graphs needed for compilation. For example, a type graph connecting subclasses to superclasses [6], or a control-flow graph between statements in a method [15].

In attribute grammars, attributes are defined by equations in such a way that for any attribute instance in any possible AST, there is exactly one equation defining its value. The equations can be viewed as side-effect-free functions which make use of constants and of other attribute values.

In RAGs, it is allowed for an equation to define an attribute by following a reference attribute and accessing its local attribute. For example, suppose node n_1 has attributes a and b, where b is a reference to a node n_2, and that n_2 has an attribute c. Then a can be defined by an equation as follows:

$a = b.c$

For Knuth-style attribute grammars, dependencies are restricted to attributes in parents or children. But the use of references gives rise to non-local dependencies, i.e., dependencies that are independent of the AST hierarchy: a will be dependent on b and c, where the dependency on b is local, but the dependency on c is non-local: the node n_2 referred to by b could be anywhere in the AST. The resulting attribute dependency graph cannot be computed without actually evaluating the reference attributes, and it is therefore difficult to statically precompute evaluation orders based on the grammar alone. Instead, evaluation of RAGs is done using a simple but general dynamic evaluation approach, originally developed for Knuth-style attribute grammars, see [11]. In this approach, attribute access is replaced by a recursive call which evaluates the equation defining the attribute. To speed up the evaluation, the evaluation results can be cached (memoized) in order to avoid evaluating the equation for a given attribute instance more than once. Caching all attributes results in optimal evaluation in that each attribute instance is evaluated at most once. Because this evaluation scheme does not require any pre-computed analysis of the attribute dependencies, it works also in the presence of reference attributes.

Caching is necessary to get practical compiler performance for other than the tiniest input programs. But caching also implies an overhead. As compared to caching all attributes, *selective caching* may improve performance, both concerning time and memory.

2.1 The JastAdd Caching Scheme

In JastAdd, the dynamic evaluation scheme is implemented in Java, making use of an object-oriented class hierarchy to represent the abstract grammar. Attributes are implemented by method declarations, equations by method implementations, and attribute accesses by method calls. Caching is decided per attribute declaration, and cached attribute values are stored in the AST nodes using two Java fields: one field is a flag keeping track of if the value has been cached yet, and another field holds the value. Figure 1 shows the implementation of the equation $a = b.c$, both in a non-cached and a cached version. It is assumed that a is declared in a class N and that the equation is declared in the subclass $SubN$. It is furthermore assumed that a is of type A. The example shows the implementation of a so called *synthesized attribute*, i.e., an attribute defined by an equation in the node itself. The implementation of a so called *inherited attribute*, defined by an equation in an ancestor node, is slightly more involved, but uses the same technique for caching. The implementation is also simplified as compared to the actual implementation in JastAdd which takes into account, for example, circularity checking. These differences are, however, irrelevant to the caching problem.

This caching scheme gives a low overhead for attribute accesses: a simple test on a flag. On the other hand, the caching pays off only after at least one attribute instance has been accessed at least twice. Depending on the cost of the value computation, more accesses than that might be needed for the scheme to pay off.

JastAdd allows attributes to have parameters. A *parameterized attribute* has an unbounded number of values, one for each possible combination of parameter values. To cache accessed values, the flag and value fields are replaced by a map

```
// non-cached version
abstract class N {
  abstract A a();
}

class SubN extends N {
  A a() {return b().c();}
}
```

```
// cached version
abstract class N {
  boolean a_cached = false;
  A a_value;
  abstract A a();
}

class SubN extends N {
  A a() {
    if (! a_cached) {
      a_value = b().c();
      a_cached = true;
    }
    return a_value;
  }
}
```

Figure 1: Caching scheme for non-parameterized attributes

where the actual parameter combination is looked up, and the cached values are stored. This is a substantially more costly caching scheme, both for accessing attributes and for updating the cache, and more accesses per parameter combination will be needed to make it pay off.

3 Attribute Instance Graphs

In order to decide which attributes that may pay off to cache, we build a graph that captures the attribute dependencies in an AST. This graph can be built by instrumenting the compiler to record all attribute accesses during a compilation. By analyzing such graphs for representative input programs, we would like to identify a number of attributes that are likely to improve the performance if left uncached. We define the *attribute instance graph* (AIG) to be a directed graph with one vertex per attribute instance in the AST. The AIG has an edge (a_1, a_2) if, during the evaluation of a_1, there is a direct call to a_2, i.e., indirect calls via other attributes do not give rise to edges. Each edge is labelled with a *call count* that represents the number of calls. This count will usually be 1, but in an equation like $c = d + d$, the count on the edge (c, d) will be 2, since d is called twice to compute c.

The main program is modelled by an artificial vertex $main$, with edges to all the attribute instances it calls. This may be many or few calls, depending on how the main program is written.

To handle parameterized attributes, we represent each accessed combination of parameter values for an attribute instance by a vertex. For example, the evaluation

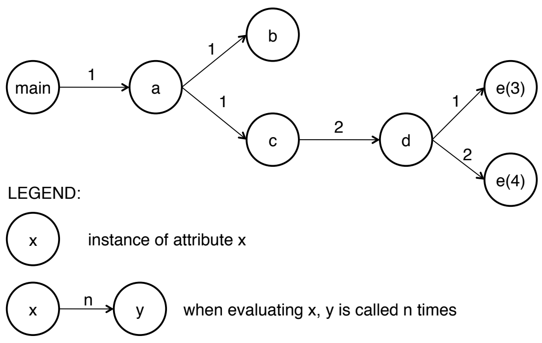


Figure 2: Example AIG

of the equation $d = e(3) + e(4) + e(4)$ will give rise to two vertices for e, one for $e(3)$ and one for $e(4)$. The edges are, as before, labelled by the call counts, so the edge $(d, e(3))$ is labelled by 1, and the edge $(d, e(4))$ by 2, since it is called twice. Figure 2 shows an example AIG for the following equations:
$a = b.c$
$c = d + d$
$d = e(3) + e(4) + e(4)$
and where it is assumed that a is called once from the main program.

3.1 An Example Grammar

Figure 3 shows parts of a typical JastAdd grammar for name and type analysis. The abstract grammar rules correspond to a class hierarchy. For example, `Use` (representing a use of an identifier) is a subclass of `Expr`. The first attribution rule:

```
syn Type Expr.type();
```

declares a synthesized attribute of type `Type`, declared in `Expr` and of the name `type`. All nodes of class `Expr` and its subclasses will have an instance of this attribute. Different equations are given for it in the different subclasses of `Expr`. For example, the equation

```
eq Use.type() = decl().type();
```

```
abstract Expr;
Use : Expr ::= ...;
Literal : Expr ::= ...;
AddExpr : Expr ::=
    e1:Expr e2:Expr;

Decl ::= Type ... ;

abstract Type;
Integer : Type;
Unknown : Type;

...
```

```
syn Type Expr.type();
syn Type Decl.type() = ...;

eq Literal.type() =
  stdTypes().integer();
eq Use.type() = decl().type();
eq AddExpr.type() =
  (left.type().sameAs(right.type()) ?
  left.type() : stdTypes.unknown();

syn Decl Use.decl() = lookup(...);
inh Decl Use.lookup(String name);
inh Type Expr.stdTypes();

syn boolean Type.sameAs(Type t) = ...;
...
```

Figure 3: Example JastAdd attribute grammar

says that for a `Use` node, the value of `type` is defined to be `decl().type()`. The attribute `decl()` is another attribute in the `Use` node, refererring to the appropriate declaration node, possibly far away from the `Use` node in the AST. The `decl()` attribute is in turn defined using a parameterized attribute `lookup`, also in the `Use` node. The `lookup` attribute is an inherited attribute, and the equation for it is in an ancestor node of the `Use` node (not shown in the grammar). For more information on name and type analysis in RAGs, see [6].

Figure 4 shows parts of an attributed AST for the grammar in Figure 3. The example program contains two declarations: `"int a"` and `"int b"`, and two add expressions: `"a + b"` and `"a + 5"`. For the `decl` attributes of `Use` nodes, the reference values are shown as arrows pointing to the appropriate `Decl` node. Similarly, the `type` attributes of `Decl` nodes have arrows pointing to the appropriate `Type` node. The nodes have been labelled `A`, `B`, and so on, for future reference.

Figure 5 shows parts of the AIG for this example. In the AIG we have grouped together all instances of a particular attribute declaration, and labelled each attribute instance with the node to which it belongs. For instance, since the node `D` has the three attributes (`type`, `decl`, and `lookup`), there are three vertices labelled `D` in the AIG. For parameterized attribute instances, there is one vertex per actual parameter combination, and their values are shown under the vertex. For instance, the `sameAs` attribute for `I` is called with two different parameters: `J` and `K`, giving rise to two vertices. (`K` is a node representing integer literal types and is not shown in Figure 4.) All call counts in the AIG are `1` and have therefore been omitted.

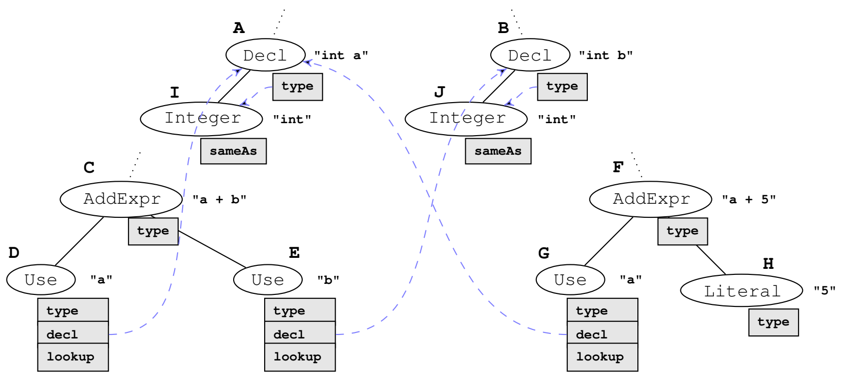


Figure 4: An example attributed AST

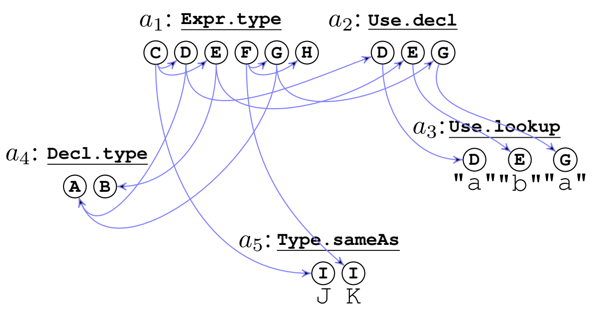


Figure 5: Parts of the AIG for the example

4 Computing a Cache Configuration

Our goal is to automatically compute a good cache configuration for a RAG specification. A cache configuration is simply the set of cached attributes, CACHED, which is a subset of the full set of attribute declarations, FULL. The FULL set is furthermore divided into two disjoint sets PARAM and NONPARAM, for parameterized and nonparameterized attributes respectively. There are some attributes that will always need to be cached, due to properties of the specification, e.g., circular attributes. We let the set PRE denote this set of attributes.

As a basis for the computation, we do profiling runs of the compiler on a set of test programs, producing the AIG for each program. These runs are done with all attributes cached, allowing us to use reasonably large test programs, and making it easy to compute the AIG which reflects the theoretically optimal evaluation with each attribute instance evaluated at most once.

It cannot be assumed that a single test program uses all attributes. We let USED be the set of attribute declarations for which at least one instance is called, and UNUSED the complementary set, UNUSED $=$ FULL $\setminus$ USED.

4.1 Excessive Calls

The $calls$ label on the edges in the AIG reflects the number of attribute calls in an all cached configuration. To find out if a certain attribute is worth uncaching, we define $extra\_evals(a_i)$, i.e., the number of extra evaluations of the attribute instance a_i that will be done if the attribute a is not cached:

$$extra\_evals(a_i) = \begin{cases} calls(a_i) - 1, & \text{if } a \in \text{NONPARAM} \\ \sum_{p \in params(a_i)} (calls(p) - 1), & \text{if } a \in \text{PARAM} \end{cases} \tag{1}$$

where $params(a_i)$ is the set of vertices in the AIG representing different parameter combinations for the parameterized attribute instance a_i. The number of extra evaluations is a measure of what is lost by not caching an attribute. The total number of extra evaluations for an attribute a is simply the sum of the extra evaluations of all its instances:

$$extra\_evals(a) = \sum_{a_i \in I_{called}(a)} extra\_evals(a_i); \tag{2}$$

where $I_{called}(a)$ is the set of attribute instances of a that are called at least once. Finally, we define the set ONE containing attributes with no extra evaluations of any of its instances. ONE $= \{a \in \text{USED} \setminus \text{PRE} : extra\_evals(a) = 0\}$.

4.2 Selection of Test Programs

Naturally, it is desirable that the test program has a large attribute coverage, i.e., as large a USED set as possible, in order to be able to compute a good configuration. To obtain as complete coverage as possible, it is advisable to also run erroneous programs which may use attributes specific to error checking. If the cache configuration is based on test programs with incomplete coverage, the attributes in UNUSED should be selected to be cached, in order to not risk performance degradation for other programs using those attributes.

5 Evaluation

To evaluate our approach we have applied it to the Java compiler JastAddJ [7]. This compiler is specified with RAGs using the JastAdd system. The specification comes with a manual cache configuration composed by the compiler author, an expert on RAGs, making an effort to obtain as good compilation speed as possible. The compiler performs within a factor of three as compared to the standard javac compiler, which is good considering that it is generated from a specification. It is clearly an expert configuration, and it cannot be expected that a better one can be obtained manually. In terms of attribute sets, JastAddJ has a FULL set of size 1031, a PRE set of size 30 and manual configuration of size 381 (MANUAL).

In our experiment, we have profiled the compilation of a single Java program, and used the resulting AIG to compute different cache configurations. We have then selected the best configuration and used it to benchmark the compilation of a suite of programs, comparing our configuration with MANUAL and FULL.

5.1 Experimental setup

All measurements were run on a high-performing computer with two Intel Xeon Quad Core @ 3.2 GHz processors, a bus speed of 1.6 GHz and 32 GB of primary memory. The operating system used was MacPro 3.1 and the Java version was Java 1.6.0._15. We used a documented measurement approach and benchmark suite [4].

Measuring of performance The JastAddJ compiler is implemented in Java (generated from the RAG specification), so measuring its compilation speed comes down to measuring the speed of a Java program. This is notoriously difficult, due to dynamic class loading, just-in-time compilation and optimization, and automatic memory management [4]. To eliminate as many of these factors as possible, we use the multi-iteration approach suggested in [5]. We start by warming up the compiler with a number of non-measured compilations (5), thereby allowing class loading and optimization of all relevant compiler code to take place, in order to reach a steady state. Then we turn off the just-in-time compilation and run a couple of extra unmeasured compilations (2) to drain any JIT work queues. After that we run several (20) measured compilation runs of which we compute confidence intervals of 95%. In addition to this, we start each measured run with a forced garbage collection (GC) in order to obtain as similar conditions as possible for each run. Memory usage is measured by checking of available memory in the Java heap after each forced GC call and after each compilation. The measurements are given with a confidence interval of 95%.

Benchmark suite We used the projects from the DaCapo suite as the basis for profiling and testing. **ANTLR**, an LL(k) parser generator (ca 35 000 LOC). **Bloat**, a program for optimization and analysis of Java bytecode (ca 41 000 LOC). **Chart**, a program for plotting of graphs and rendering of PDF files (ca 12 000 LOC). **FOP**, parses XSL-FO file and generates PDF files (ca 136 000 LOC). **HsqlDb**, a database application (ca 138 000 LOC). **Jython**, a Python interpreter (ca 76 000 LOC). **Lucene**, a program for indexing and searching of large text corpuses (ca 87 000 LOC).

PMD, a program for analyzing Java classes for a range of source code problems (ca 55 000 LOC). **Xalan**, a program for transformation of XML documents into HTML (ca 172 000 LOC). The DaCapo suite also contains Eclipse, but this project was excluded due to compilation issues.

Cache configurations We want to compare the result of using different kinds of cache configurations. To start with we have access to an expert configuration in the MANUAL set and two basic configurations:

M MANUAL, cache a manually selected set of attributes

B1 FULL, cache all attributes in the specification

B2 PRE, cache only attributes that are inherently cached, e.g., circular attributes

By applying our profiling scheme, counting number of calls and evaluations of attributes on a test program, we can obtain the following three profiled configurations:

P1 USED $\cup$ PRE, like **B2**, but cache also all used attributes

P2 (USED $\setminus$ ONE) $\cup$ PRE, like **P1**, but do not cache attributes that are evaluated only once

P3 (USED $\setminus$ ONE) $\cup$ PRE $\cup$ UNUSED, like **P2**, but cache also attributes not used by the profiling program

The expert configuration **M** is interesting to compare to, as it would be nice if we could obtain similar results with our automated methods, and possibly even better results.

The full configuration **B1** is interesting as it is easily obtainable and robust with respect to performance: there is no risk that a particular attribute will be evaluated very many times for a particular input program, and thereby degrade performance.

The least possible configuration, **B2**, is interesting as it provides a lower bound on the memory needed during evaluation. However, this configuration will in general be useless in practice, leading to compilation times that increase exponentially with program size.

P1 is interesting because it is an easy-to-compute configuration that will perform better for the profiled program than **B1**, by leaving out the initialization costs for unused attributes. For other input programs, however, there is a risk of exponential behavior.

P2 and **P3** are like **P1** and **B1**, but avoid caching of attributes that are used only once for the profiling program. If we assume that this behavior is representative for all input programs, i.e., that *if* these attributes are used, they are used only once, then it will pay off to remove them from the caching configuration used by the compiler. Depending on how much the attributes in UNUSED are used in different input programs, it might pay off to exclude or include them. For the profiled program it clearly pays of to exclude them. However, for other input programs, there is a risk of exponential behavior. So performance will be more robust if they are included.

5.2 Profiling using Hello World

As an experiment we profiled a simple Hello World program. We compiled Hello World using configurations **B1**, **B2**, **P1**, **P2** and **M**. The results are shown in Figure 5.2. It is clear from these results that **B2** is not a good configuration, even on this small program. Even though it provides excellent memory usage, the execution time is several times slower than any of the other configurations. For a larger application the **B2** configuration would be useless.

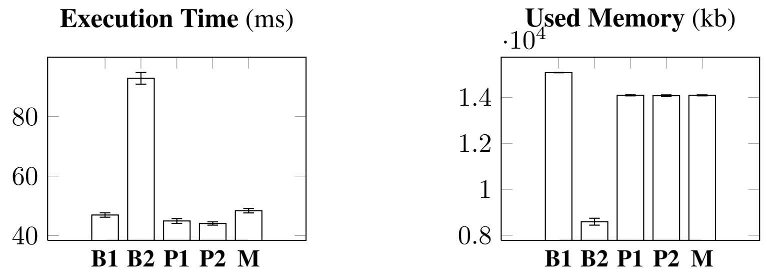


Figure 6: Compilation of Hello World The plots show the results of compiling Hello World using different cache configurations. **P1** and **P2** are acquired from profiling of the same application (Hello World). The I-shaped markers show the confidence interval.

5.3 Profiling using ANTLR

As an example of a normal-sized Java program to use for profiling we chose ANTLR from the DaCapo suite [4,18]. This choice was made arbitrarily: ANTLR is the first program in the suite. When compiling ANTLR using JastAddJ, 50.2 % of the attributes in the FULL set are used, which is a reasonable number considering that a lot of attributes deal with semantics not found in the average Java application.

We ran the profiler on ANTLR and then compiled ANTLR using **M**, **B1**, **B2**, **P1**, **P2** and **P3**. The results are shown in Figure 7. We see that all the configurations from the profiler end up somewhere between **B1** and **M**. **P2** performs best, among the configurations from the profiler, with a mean speed-up of execution time at 23%, which is close to the speed-up of **M** (26%). **P3** is clearly a better option than

P1 showing that it pays off to remove the ONE set. **P3**, with a speed-up of 16% is slower than **P2**. This is due to the inclusion of unused attributes, resulting in more memory usage and initialization costs. However, we consider the performance of **P3** to be good considering that this configuration provides extra robustness.

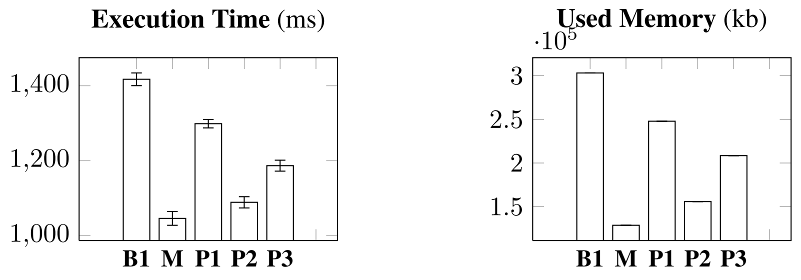
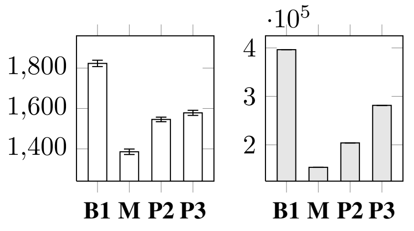


Figure 7: Compilation of ANTLR The plots show the results of compiling ANTLR. The I-shaped markers show the confidence intervals.

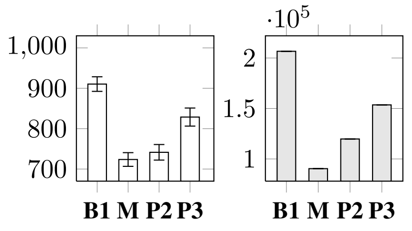
Compiling other projects from the suite We apply the best configuration from our experiments compiling ANTLR (**P2**), along with the best robust version (**P3**), to the remaining projects in the benchmark suite. We also include **B1** and **M** for each project as a reference. The compilation results are shown in Figure 8. The figure shows that the lack of robustness in **P2** results in very bad performance for half of the projects (FOP, Lucene, PMD and Xalan). The **P3** configuration, on the other hand, compiles all of the projects with good performance, close to **M** in two cases (HsqlDb and Xalan) and better than **M** in two cases (Lucene and PMD). Considering the speed-up of **P3** as a geometric mean for all projects, we get a speed-up of 23.5%. The corresponding mean for **M** is 31.7%.

Observations from the ANTLR trace In order to try to find out more about the caching behavior, we generated an extended trace for ANTLR using full caching. In this extended trace we note that for some attributes the number of evaluations of a cached attribute is *far more than one*, as shown in Table 5.3.

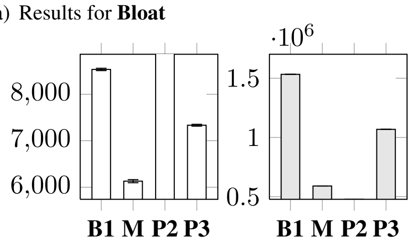
These extra evaluations are due to rewrites included in the JastAddJ specification. Rewrites are evaluated at first access to a subtree in an AST and are triggered via expressions containing attributes. Attributes are cached during rewrites because changes in the AST may affect attribute values.



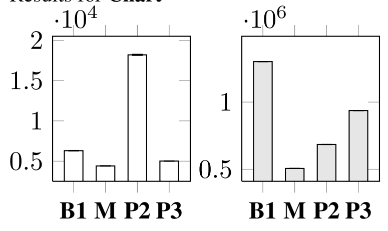
(a) Results for **Bloat**



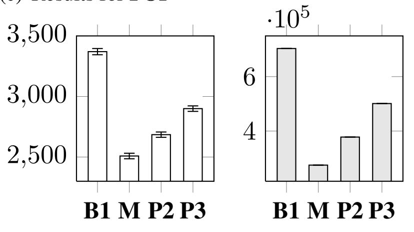
(b) Results for **Chart**



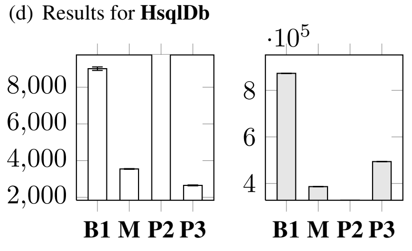
(c) Results for **FOP**



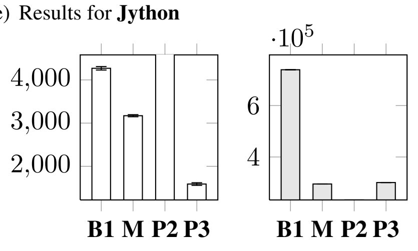
(d) Results for **HsqlDb**



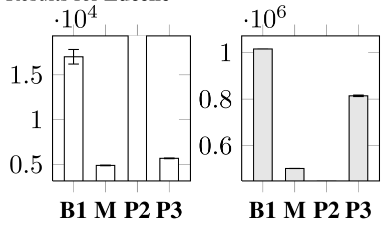
(e) Results for **Jython**



(f) Results for **Lucene**



(g) Results for **PMD**



(h) Results for **Xalan**

Figure 8: Compilation of remaining projects The plots show the results of compiling the remaining projects in the benchmark suite. The white bars show **execution time** (ms), gray bars show **memory usage** (kb) and I-shaped markers show confidence intervals. Time bars reaching the top of the diagram are slower than **B1** due to uncached attributes. These time bars do not have a corresponding memory bar (not measured).

Attributes ($a \in A$)	$\lvert I_a \rvert$	$\overline{calls(i)}$	$\overline{evals(i)}$	$\overline{cached(i)}$
AccessControl				
syn ArrayDecl.accessibleFrom(TypeDecl)	10	54.8	**7.4**	48.4
syn ConstructorDecl.accessibleFrom(TypeDecl)	316	10.6	**2.9**	8.7
syn FieldDeclaration.accessibleFrom(TypeDecl)	254	0.8	**1.8**	0.0
syn MethodDecl.accessibleFrom(TypeDecl)	2 910	13.8	**4.5**	10.3
syn TypeDecl.accessibleFrom(TypeDecl)	264	99.9	**8.5**	92.3
syn TypeDecl.accessibleFromExtend(TypeDecl)	42	3.0	**4.0**	0.0
syn TypeDecl.accessibleFromPackage(String)	230	6.1	**1.9**	5.2
AccessTypes				
syn Expr.isPackageAccess()	10 861	0.0	**3.4**	0.0
syn Expr.isSuperAccess()	692	0.9	**1.0**	0.9
syn Expr.isTypeAccess()	3 446	0.5	**1.0**	0.5
AncestorMethods				
syn TypeDecl.ancestorMethods(String)	192	29.0	**22.3**	7.7
Annotations				
inh Modifiers.lookupType(String,String)	2 918	0.0	**1.0**	0.0
syn ConstructorDecl.isDeprecated()	241	4.9	**1.0**	4.9
syn FieldDeclaration.isDeprecated()	958	10.3	**1.0**	10.3
syn MethodDecl.isDeprecated()	1 461	8.4	**1.0**	8.4
syn Modifiers.annotation(TypeDecl)	2 918	0.0	**1.0**	0.0
syn Modifiers.hasDeprecatedAnnotation()	2 918	0.0	**1.0**	0.0
syn TypeDecl.isDeprecated()	258	49.1	**1.0**	49.1
...				

Table 1: Extract from the ANTLR trace Shows some of the attributes (A) used under the compilation of ANTLR using full caching. The columns show attributes ($a \in A$) (grouped after aspects), number of attribute instances ($|I_a|$), average number of calls to attribute instances of the attribute ($\overline{calls(i)}, i \in I_a$), average number of calls resulting in an evaluation ($\overline{evals(i)}, i \in I_a$), and average number of calls using cached values ($\overline{cached(i)}, i \in I_a$).

6 Related Work

There has been a substantial amount of research on optimizing the performance of attribute evaluators and to avoid storing all attribute instances in the AST. Much of this effort is directed to optimize static visit-oriented evaluators, where a number of attribute evaluation visits is computed statically from the dependencies in an attribute grammar. For RAGs, such static analysis is, in general, not possible due to the reference attributes. As an example, Saarinen introduces the notion of *temporary attributes* that are not needed outside a single visit, and shows how these can be stored on a stack rather than in the AST [16]. The attributes we have classified as ONE correspond to such temporary attributes: they are accessed only once, and can be seen as stored in the stack of recursive attribute calls. Other static analyses of attribute grammars are aimed at detecting *attribute lifetimes*, i.e., the time between the computation of an attribute instance until its last use. Attributes whose instances have non-overlapping lifetimes can share a global variable, see, e.g., [12]. Again, such analysis cannot be directly transfered to RAGs due to the use of reference attributes.

Memoization is a technique for storing function results for future use, and is used, for example, in dynamic programming [3]. Our use of cached attributes is a kind of memoization. Acar et al. present a framework for selective memoization in a function-oriented language [1]. However, their approach is in a different direction than ours, intended to help the programmer to use memoized functions more easily and with more control, rather than to find out which functions to cache. There are also other differences between memoization in function-oriented programming, and in our object-oriented evaluator. In function-oriented programming, the functions will often have many and complex arguments that can be difficult or costly to compare, introducing substantial overhead for memoization. In contrast, our implementation is object-oriented, reducing most attribute calls to parameterless functions which are cheap to cache. And for parameterized attributes, the arguments are often references which are cheap to compare.

7 Discussion and Future Work

We have presented a profiling technique for automatically finding a good caching configuration for compilers generated from RAG specifications. Since the attribute dependencies in RAGs cannot be computed statically, but depend on the evaluation of reference attributes, we have based the technique on profiling of test programs. We have introduced the notion of an attribute dependency graph with call counts, extracted from an actual compilation. Experimental evaluation on a generated Java compiler shows that by profiling on only a single program with an attribute coverage of only 50.2%, we get a mean compilation speed-up of 23.5%, as compared to caching all attributes. This is close to the performance obtained for a manually composed expert configuration (31.7%). We find this result very encouraging and intend to continue this work with more experimental evaluations. In particular, we plan to combine the profiling of several programs to obtain a higher attribute coverage, hopefully obtaining even better results, and to apply the technique to compilers for other languages. Further, we would like study the effects on rewrites in order to automatically obtain even better cache configurations.

8 Acknowledgements

We are grateful to Torbjörn Ekman for fruitful discussions and for implementing support in JastAdd for profiling and separate cache configurations.

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *POPL*, pages 14–25. ACM, 2003.

[2] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, In Press, 2009.

[3] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM.

[5] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, 2008.

[6] Torbjörn Ekman and Görel Hedin. Modular name analysis for Java using JastAdd. In *Generative and Transformational Techniques in Software Engineering, International Summer School (GTTSE 2005)*, volume 4143 of *LNCS*. Springer, 2006.

[7] Torbjörn Ekman and Görel Hedin. The Jastadd Extensible Java Compiler. In *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007)*, pages 1–18. ACM, 2007.

[8] Torbjörn Ekman and Görel Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

[9] Görel Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.

[10] Görel Hedin and Eva Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[11] Martin Jourdan. An optimal-time recursive evaluator for attribute grammars. In *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 167–178. Springer, 1984.

[12] Uwe Kastens. Lifetime analysis for attributes. *Acta Informatica*, 24(6):633–651, 1987.

[13] Lennart C. L. Kats, Anthony M. Sloane, and Eelco Visser. Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming. In *Compiler Construction, 18th International Conference*, volume 5501 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2009.

[14] Donald E. Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (1971).

[15] Emma Nilsson-Nyman, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. Declarative intraprocedural flow analysis of Java source code. In *Proceedings of the Eight Workshop on Language Description, Tools and Applications (LDTA 2008)*, Electronic Notes in Theoretical Computer Science. Elsevier B.V., 2008.

[16] Mikko Saarinen. On constructing efficient evaluators for attribute grammars. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 62 of *Lecture Notes in Computer Science*, pages 382–397. Springer, 1978.

[17] Anthony M. Sloane, Lennart C.L. Kats, and Eelco Visser. A Pure Object-Oriented Embedding of Attribute Grammars. In *Proceedings of the 9th Workshop on Language Descriptions, Tools and Applications (LDTA 2009)*, 2009.

[18] The DaCapo Project. The DaCapo Benchmarks, 2009. `http://dacapobench.org`.

[19] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an Extensible Attribute Grammar System. *Electr. Notes Theor. Comput. Sci.*, 203(2):103–116, 2008.

Technical Report II

A Semantic Editing Model in Support of Reference Attribute Grammars

Abstract

Good programming tools, like semantic editors with knowledge of the meaning of a language, can help developers produce better programs. Sophisticated tool support like this should be available to all language communities, but tool development is often to costly for small communities. One way to reduce this cost is to use tool generators. In this report we give an early account of *JedGen – a semantic editor generator*. In particular, we describe the *semantic editing model* of JedGen in support of reference attribute grammars (RAGs). RAGs are a means for expressing the semantics of a language in a concise and modular way. The presented semantic editing model makes use of RAGs to define services in a generated semantic editor.

Emma Söderberg
Technical report LU-CS-TR-2010-246, ISSN 1404-1200, Report 95, 2010
Department of Computer Science, Lund Univeristy

1 Introduction

Good programming tools like *semantic editors*, editors with knowledge of the semantics of a programming language, are essential for efficient software development. A sophisticated semantic editor can provide users with *semantic services* like name completion, detection of code smells and refactorings, which help programmers efficiently produce high quality code.

In order for a textual semantic editor to provide semantic services, the editor needs to construct a semantic model of the edited text. Normally, this semantic model is in the form of an *abstract syntax tree* (AST), a model common in language processors like compilers. With an available AST, an editor can provide users with feedback of the current state of the AST. For instance, the editor can show users semantic errors like missing variable declarations or unknown types. Users expect semantic editors to provide feedback like this in response to change. In order for an editor to provide up-to-date feedback, it needs to keep the underlying AST synchronized with the edited text. As long as a user makes changes to the edited text, this synchronization work continues. We call this continuous response to change, in order to provide feedback, the *feedback cycle*.

While maintaining the feedback cycle, the editor needs to provide users with semantic services. These services can either be provided as a response to updates of the AST, like semantic errors, or be requested by the users, like refactorings. We call the mechanism handling the feedback cycle and providing semantic information via the AST the *semantic editing model*.

In this report we describe the semantic editing model supported by *JedGen*, a tool with the purpose of *generating semantic editors*. Preferably, every language should have sophisticated tool support like semantic editors, but tool development is time-consuming and error-prone making it too costly for a lot of language communities. Tool generators can reduce costs by generating tools from specifications. The intention of JedGen is to generate semantic editors with support for services like name completion and refactorings.

Besides lowering development costs, tool generators have the potential of generating better tools, compared to hand-crafted alternatives. For example, LALR parser generators can generate complex parsers which would be difficult to implement manually. This potential of generating better tools is due to the benefits of a formal specification. For example, a parser is described using a formal context-free grammar specification. The JedGen tool supports a formal specification of semantics using *reference attribute grammars* (RAGs) [9], an extension of *attribute grammars* (AGs) by Knuth [13]. AGs extend an context-free grammar with context-sensitive semantic rules (*attributes*) in order to specify the semantic of a language. RAGs extend AGs by letting these attributes have references to AST nodes as values, which makes it easier to express, for example, declaration-use relations in a language. RAGs present a powerful means for describing semantics and have been used to describe compilers for languages like Java , the JastAddJ compiler [5], and Modelica, the JModelica compiler [1].

The generation of semantic editors using RAGs is related to other tool generators like the Synthesizer Generator by Reps and Teitelbaum [20], which generates AG-based syntax-directed editors. Other examples include the Mjølner/ORM system by Magnusson and Hedin [10] which generates AG-based integrated environments for object-oriented languages, and the Lrc system by Kuiper and Saraiva [14], generating semantic editors, with semantics described using *higher-order attribute grammars* (HAGs), another extension to AGs. JenGen's support for RAGs make it different from these tool generators. Part of the goal with JedGen is to utilize the power of RAGs in semantic editors. Promising work by Schäfer et al. [21,22] show that RAGs can be used to support sophisticated sevices like refactorings. RAGs can also be used to specify code smell services like detection of dead code, as described by Nilsson-Nyman et al. in [19].

Besides tool generation, reuse of libraries and generic components can speed up tool development. One example of a platform facilitating tool development by reuse, is the Eclipse platform [4, 7]. Eclipse provides an *integrated development environment* (IDE), which can be extended with new tools. The platform seamlessly integrates extending tools, called *plug-ins*, with other tools on the platform. The semantic editors generated by JedGen are intended to run on the Eclipse platform. As a consequence, the semantic editing model supported by JedGen is built on top of Eclipse. Besides Eclipse, the JedGen tool uses the support for RAGs offered by the JastAdd system [6] for generation of semantic service information.

The JedGen tool is still in its infancy, but an alpha version of the tool supporting the semantic editing model has been used by Schäfer et al. in their exploration of refactorings [21,22]. JedGen has also been used by several undergraduate students, as a part of their thesis work [15–17], and in a graduate course on RAGs.

The rest of this report starts with a historical note on the development of JedGen in Section 2, followed by some background about the Eclipse platform and the JastAdd system in Section 3. Section 4 describes the semantic editing model and Section 5 gives a description of semantic services. Finally, Section 6 gives some concluding remarks along with an outline of future work.

2 A Historical Note on the Development

The work on what is now called JedGen was started during a visit by the author to Oxford during the summer of 2007. The original goal was to develop a semantic editor for Java on top of Eclipse for refactoring experiments.

The Eclipse platform at the time already hosted the successful Eclipse Java development tool (JDT), which is hand-crafted on top of the Eclipse platform. An initial approach, extending the JDT implementation, was considered, but rejected in favor of generating parts of a semantic editor extending the existing Java compiler JastAddJ, developed using RAGs. This second approach allowed for a

RAG-based AST and more flexibility in developing RAG-based refactorings. As the result of an exploratory addition of Java services, comparable to those found in the Eclipse JDT, a semi-generated semantic Java editor reached a somewhat shaky alpha state at the beginning of the fall of 2007. At this early stage, where we were trying to build an environment with the same set of services as the JDT, we chose, for example, to support textual editing.

This choice of supporting textual editing shed light on some of the difficulties of maintaining a feedback cycle for textual editing. It was concluded that a semantic editing model for textual editing needs robust parsing of text in order to continuously construct internal ASTs. Several attempts were made to get the existing LALR parser, generated by Beaver, robust by adding error productions. It was noted that certain errors, which to the human eye seemed simple, totally failed to parse. Typically, these were errors where characters defining the structure of the code were missing, for example, a missing end brace for a Java method. These errors might not look so bad to a user due to the secondary notation – indentation. Hence, the idea to use indentation to facilitate error recovery via *scope recovery* was born. This idea evolved during 2008 to what is now called *bridge parsing* [3, 18].

The work on JedGen continued a couple of months during the summer of 2008, but the tool was mainly polished during this period. The semantic editing model went through a larger refactoring during the winter of 2009 moving it closer to its existing design. During the first part of this year (2010) the, earlier nameless, tool has been given the name *JedGen* for *JastAdd-based semantic editor generator*, as the first step of what is to come.

3 Background

The intention is for JedGen to generate semantic editors which are to run on top of the Eclipse platform. As a consequence, the semantic editing model is designed with consideration of notions like *plug-ins* etc. supported by Eclipse. Besides the influence of the Eclipse platform, the editing model is designed to support RAG-based ASTs, as they are generated by JastAdd, a system supporting RAGs and other AG extensions.

3.1 The Eclipse Platform

Eclipse, initiated in 2001 by a consortium of companies and organizations such as IBM, Rational Software and RedHat Linux, became an *not-for-profit corporation* [7]. The Eclipse framework provides a platform for exploratory work as well as large-scale software development. The platform has gained large acceptance in industry, as well as in research, partly due to the well-known Eclipse Java Development Toolkit (JDT), which is similar to other Java development tools like

NetBeans [2] and IntelliJ IDEA [12]. Several research groups use Eclipse as a platform for their research.

The Plug-in Architecture The Eclipse platform is designed using a plug-in architecture. A *plug-in* is a wrapping of a tool which describes how the tool either depends on or extends other plug-ins. The platform is, except for a small hand-coded core, entirely constructed using plug-ins. Except for a small set of default plug-ins which are loaded on start-up, the platform loads new plug-ins when they are needed by depending plug-ins.

For example, when a certain file type is double-clicked in a file browser the editor matching the content-type of the file should be opened. The first time this type of file is opened, the plug-in extending the platform with a suitable editor is activated by the platform.

The platform offers a number of possible *extension points* which a plugin can extend with functionality. One such example is the `org.eclipse.ui.editors` extension point which provides a means for adding a new editor to the platform. In addition to the extensions offered by the platform, new plug-ins can add new extension points. All plug-ins extending the platform can be found via a *plug-in registry* maintained by the platform.

Resources and Builders Eclipse uses the notion of a workspace in which a user can create *projects*. Projects are the top level elements in the Eclipse workspace. A project may contain a number of *folders* and *files*, but may not be nested within other projects. All these *resources* – projects, folders and files, correspond to entities in the file system of the underlying operating system.

Much in Eclipse depend on the concept of projects as top-level elements. Configurations are set on the project level, for example, class paths and compiler versions. The logic of setting things like compiler version on a project is due to Eclipse's mapping of builders to projects. A *builder* is a plug-in extension that creates an in-memory model, like an AST, by parsing a resource. Typically, a builder is called when a resource, like a file, is saved or when a user requests a re-build of a project. A project will typically have one or several builders listed in its project configuration. These builders are called one by one when a build action is requested. The calling of the builder plug-ins is completely handled by the Eclipse platform.

3.2 The JastAdd System

JedGen uses the JastAdd system to generate semantic service information. The JastAdd system supports RAGs, and other AG extensions, and the specification of an *abstract grammar* defining the nodes of an AST. The abstract grammar and the RAG notation in JastAdd support object-oriented inheritance and aspect-oriented inter-type declarations, both contributing with means for modular development and extensibility.

Reference Attribute Grammars RAGs are an extension of attribute grammars (AGs) presented by Knuth [13], which are a means for extending a context-free grammar with context-specific semantic rules called attributes. The original AG version supported two kinds of attributes – *synthesized* and *inherited*. Synthesized attributes provide a means for propagating information upwards in the AST, in a fashion similar to methods in Java. Inherited attributes provide a means for propagating information downwards in the AST. Attributes are defined on node types. Each instance of a certain node type will have its own *attribute instances* of all attributes defined for the node type. Attributes are similar to functions in that they are defined in terms of other attributes without any side effects. In Knuth's version of AGs, attributes are not allowed to depend on themselves (*circular dependencies*) and attributes are defined in terms of child nodes and parent nodes of the node on which the attribute is defined.

RAGs extend the original kind of AGs with the notion of *reference attributes*. These are attributes that may have references to other nodes in the AST as values. This is useful for defining definition-use relations and facilitates modular specifications.

Caching of Attribute Values JastAdd supports dynamic demand-driven evaluation of attributes, that is, attribute are evaluated when they are needed. Attributes may depend on other attributes, which means that the evaluation of one attribute may trigger evaluation of other attributes. For attribute values which are requested several times, evaluation more than once is unnecessary, unless the state of the AST has changed since the last request. In order to avoid unnecessary evaluation of attributes, they can be defined to be *cached*, that is, their computed value can be saved and reused in later requests.

In case the AST has changed since the last request of a cached attribute value, the value needs to be *flushed*, that is, removed since it may depend on information that has changed. Small changes of an AST may only affect a small subset of all the attributes in an AST. Preferably, only those attribute affected by a certain change of the AST, should be flushed. In order to accomplish an optimized flushing of attributes, the dependencies between attribute need to be known. Automated support for re-evaluation, *incremental updating*, of RAGs is an unsolved problem. In JedGen, we therefore currently flush all attributes after any kind of change to an AST.

4 The Semantic Editing Model

The purpose of the semantic editing model is to maintain the feedback cycle and provide semantic service information. To describe the details of the semantic editing model, we study an instance of a semantic editor showing the content of a single file. We consider an editor to be an entity which can be instantiated in several *editor instances* which are active at the same time. An editor instance shows the content of a file modeled using a *file AST*, an AST with content corresponding to the text in the editor instance. An editor instance queries its file AST for semantic service information. In order for the file AST to provide its editor instance with fresh semantic information it needs to stay up to date with the content of the editor instance. The editing model needs to provide means for the following:

- *Notification of change* in an editor instance, in order to initiate a re-build of a corresponding file AST.
- *Parsing of files* in editor instances in order to construct corresponding file ASTs.
- *Updating of ASTs* in order to replace an old file AST with a new AST constructed in response to a change in an editor instance.
- *Notification of updates* to editor instances, in order for them to know when new semantic information is available in their file AST.
- *Provision of semantic service information* from a file AST to the corresponding editor instance.

The builder mechanism on the Eclipse platform provides a means for notification of change, at least, changes discovered when a file is saved. We use this mechanism to define a *JedGen builder*.

To handle parsing of files, suitable resource parsers need to be known to the JedGen builder. The extension point mechanism in Eclipse provides a means for adding extensions which can be found via the plug-in registry. We use this mechanism to add a *resource parser extension* accessible by the JedGen builder. For a resource parser to update an existing file AST with a new instance we provide an *update interface*.

For notification of change, the Eclipse platform makes extensive use of the *observer pattern* [8], where an object maintains a list of observers which are notified of changes in the state of the object. We make use of this pattern (calling observers listeners) by letting editor instances listen to changes in their corresponding file AST via a *listener interface*. Finally, to make the semantic information in the file AST available to an editor instance, we define a *service interface* for each semantic service in the editor instance.

The primary purpose of a file AST is to model the content of a file shown in a corresponding editor instance. It is not to maintain, for example, listener lists

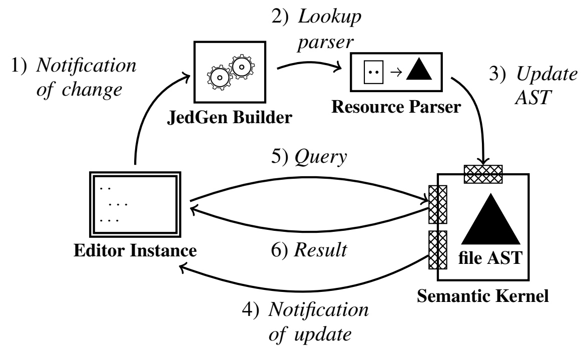


Figure 1: An overview of the feedback cycle for one editor instance 1) The editor instance sends a notification to the builder when a user saves a file. 2) The builder looks up a suitable resource parser. 3) The resource parser translates the file content to a file AST and sends it to the *update interface* of the semantic kernel. 4) The kernel stores the file AST and notifies listeners (the editor instance) of an update via the *listener interface*. 5) The editor instance queries the semantic kernel for semantic information from its file AST via the *service interface*. 6) The semantic kernel sends results via the service interface.

and notification. For tasks like these we use an other entity we call the *semantic kernel*. The semantic kernel provides a *repository for ASTs* and maintains the three interfaces presented above – the update, service and listener interfaces. In maintaining, for example, the listener interface the semantic kernel handles registration, removal and notification of listeners, and in handling the service interfaces the semantic kernel maps interface calls to attribute values in a file AST.

Figure 1 shows an editor instance, its corresponding file AST and the enclosing semantic kernel providing update, listener and service interfaces. The figure illustrates the feedback cycle as an outer loop, and the provision of service information as an inner loop. The outer loop reacts to save events in the editor instance by calling the JedGen builder. The JedGen builder looks up a resource parser suitable to parser the the saved file. The resource parser translates the file to a file AST and sends it to the semantic kernel via the update interface. The semantic kernel updates the internal representation of the file AST and notifies the listening editor instance via the listener interface. In the inner loop, the editor instance queries the semantic kernel for semantic information via a method call and the semantic kernel returns results based on the query. The editor instance may, for example, query the semantic kernel as a response to an update notification, or as a response to a user request.

4.1 Synchronization

Normally, in an interactive environment there are *several active threads* sharing the work-burden in order to provide users with reasonable response times. This is also true for the Eclipse platform, where, for example, builders run in a background thread while editor instances run in a foreground thread dedicated to graphical components. With this in mind, the semantic kernel needs to *synchronize access* to the contained file AST. The synchronization involves the following:

- *No escaping node references*, meaning that no references to nodes of an AST maintained by the semantic kernel are allowed outside of the kernel. This is because external references may become out-dated due to updates of a file AST.
- *One locked access point*, meaning that all access to the file ASTs, maintained by the semantic kernel, are accessed through one well-defined access point protected by a lock (mutual exclusion). Typically, the top node of the AST, the root node, is a good access point.

With the semantic kernel already maintaining the repository of ASTs, we can extend its responsibilities to also include synchronization and the prevention of escaping node references.

4.2 Cross-References between Files

Cross-references in a RAG-based AST may be references to variable declarations from variable uses, or the reverse, references from variable declarations to variable uses. In large applications the implementation is, normally, spread out over several files. Preferably in this scenario, a use should be able to point to its declaration even if it is defined in another file. In order to handle this using file ASTs, we need to have references between ASTs. This is problematic since then we are referring to nodes inside another AST and these nodes may be updated, hence, the reference may become inconsistent. To handle references between ASTs we can, either, try to manually manage the references, or we can introduce a new AST enclosing file ASTs from, say, the same project. We choose the second alternative and introduce an enclosing AST we call the *project AST*.

Nesting of file ASTs inside project ASTs requires no changes in the semantic kernel. In fact, the semantic kernel can maintain both project ASTs and file ASTs in the AST repository, or any other AST with a root connected to a resource on the Eclipse platform. Figure 2 illustrates the structure of the project AST.

4.3 Loosely-Coupled Node References

Direct references to AST nodes from outside the AST, are not allowed but some kind of references are necessary for updating, notification and querying for semantic service information. For this purpose, we have added the concept of *node*

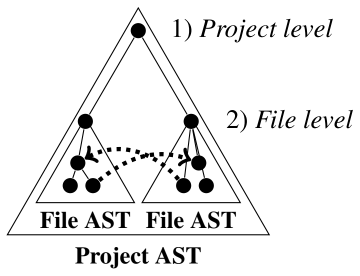


Figure 2: An overview of the project AST 1) The root of the AST corresponds to a project. 2) The children of the root correspond to files in the parent project. The dotted edges show how the enclosed file ASTs can support cross-references between files.

keys, loosely coupled node references allowed outside of the semantic kernel. Each node in an AST has a node key which is unique to the node. A node key contains information of the enclosing project, file and a path to the node, based on child indices.

All communication through the compiler, listener and service interfaces use node keys to specify which AST node that should be regarded. Any node can be asked for its node key. However, editor instances and resource parsers have no direct access to nodes in the semantic kernel. Instead, they create node keys based on a specific resource they are working with. For example, an editor instance can create a node key for a file and then use this node key to communicate with the semantic kernel. The semantic kernel will register listeners to a certain node key, update ASTs for a certain node key, and provide semantic service information for a certain node key. Figure 3 illustrates the *lookup* of a node using a node key.

Offset-based Mapping for Textual Editing In textual editing, text positions and intervals need to be mapped to AST nodes in order for the semantic editor to provide certain kinds of services. For example, in code browsing a user selects a name and asks for its declaration. The intended node is given by position information about the text selection and the file containing the selection. Eclipse regards a file as a sequence of characters, and represent positions as *offsets* in that sequence. An offset can also be communicated using line and column information. A selection is normally given using an *interval of offsets*.

In order to support services in need of this kind of offset-to-node mapping, we introduce the concept of *mapping node keys*. A mapping node key works like an ordinary node key, with the exception of the node path, which is replaced with an offset interval. Figure 4 illustrates how a mapping node key is used to find a specific node.

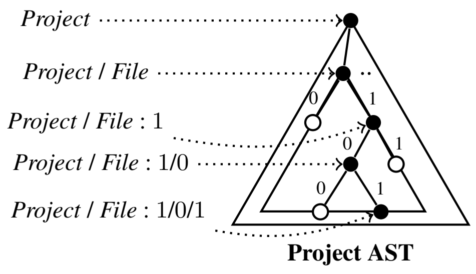


Figure 3: An example of a node key The column to the left shows the node keys for the black nodes in the project AST to the right. Each node key corresponds to a specific node, illustrated with a dotted arrow. A node key contains information about the *enclosing project*, the *enclosing file* and the *index path* to a node. The length of the path depends on the corresponding level of the project AST. The numbers on the edges in the project AST show child indices used in the node paths.

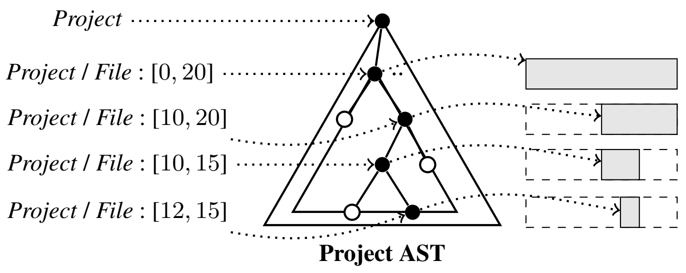

Project
Project / File : [0, 20]
Project / File : [10, 20]
Project / File : [10, 15]
Project / File : [12, 15]
Project AST

Figure 4: An example of a mapping node key The column to the left shows the mapping node keys for the black nodes in the project AST in the middle. The column to the right shows an *offset interval* from a file, that is a section of the text. Each node key corresponds to a specific node, illustrated with a dotted arrow. Each node within an AST corresponds to an offset interval, as illustrated by the dotted arrows, emanating from the black nodes.

Handling of Outdated Node Keys Node keys stored outside of the semantic kernel may become outdated. For example, an editor instance may try to ask for semantic information using a node key connected to an old version of its corresponding file AST. A file AST may, in fact, have been updated several times between a call to a certain service.

To deal with outdated node keys, we introduce a basic version handling of ASTs using *build version IDs*. Each AST (project AST of file AST) entering the update interface of the semantic kernel, is given a build version ID from a counter. This ID is set at the root of the subtree and propagated to descending nodes in the subtree. The result being that child AST nodes may have higher build IDs than their parent nodes. For example, a file AST may be updated after a project AST has been updated and, hence, get a higher build ID. The reverse, that a child node would have a lower build ID than its parent is a malformed state and not allowed. This cannot happen, because the update of a parent node results in updates of its child nodes.

During node lookup, the build version ID of the regarded node key is compared to the build version ID of encountered nodes on the path. If a node with a higher build version ID is encountered on the path, the tree has been updated. This will interrupt the lookup of the node and return an empty result, along with a notification of change for the node key.

Handling of Unknown Node Keys Node keys used to communicate with the semantic kernel may be unknown to the kernel. For example, this may happen the first time a resource parser creates a project AST and sends it to the update interface of the semantic kernel. Another example is when an editor instance queries the service interface before the referenced AST, referred to by the node key, has been added to the AST repository. In fact, all the interfaces of the semantic kernel need to handle unknown node keys.

For the listener interface, the handling of unknown node keys is easy, because listeners are registered and removed in a list separate from the content of the AST repository. For the service interfaces, on the other hand, it is impossible to provide semantic service information using an unknown node key. A service information request with an unknown node key will result in an empty result. Finally, for the update interface, the occurrence of an unknown node key corresponds to a registration of a new entry in the AST registry.

4.4 Updating

The update interface of the semantic kernel allows resource parsers to update project ASTs and file ASTs in the AST repository of the kernel. Project ASTs are updated by replacement in the AST repository, while file ASTs are updated via node replacement in a project AST. File ASTs with their own place in the AST repository (due to a missing enclosing project AST) may also be updated by replacement in the repository. This may happen when a file from outside of the Eclipse workspace is being edited in the workspace.

Updating of Attribute Values Node replacement in a RAG-based AST is non-trivial in that there may be attribute references to the node being replaced, or to children of the node. In order to maintain a consistent AST, cached attribute values in the enclosing AST need to be flushed (removed). This is to avoid attribute references to nodes in an old version of the AST.

Currently, flushing is performed on all nodes in the AST being updated, that is, via the root of the project AST. This is not optimal since it most likely means that unaffected attribute values are removed, but unavoidable without support for incremental evaluation of RAGs.

5 The Service Flora

The service flora of a state-of-the art semantic editor of today contains a large amount of services. A recent study by Hou and Wang [11], studying features (services) in the Eclipse JDT, makes the following grouping of visible features for code manipulation: *reading* (e.g., display of program information (like errors and outlines), browsing (code search and navigation)) and *writing*. The writing category is further divided into three subcategories: *editing* (e.g., automatic indentation, generation of code fragments), *refactoring* (behavior-preserving code transformations, e.g., rename, inline method), and *code assist* (inferred code manipulation actions, e.g., quick fix, name completion). Some of these services require semantic analysis, while some require only lexical or syntactical analysis, for example, syntax highlighting or auto indentation. We are interested in the first category – the *semantic services*. The following services end up in this category: refactorings, code assist and reading (with regard to display of errors and browsing). Outlines can also be semantic if they are constructed using semantic information.

5.1 Implementation of Services in JedGen

In JedGen, a semantic service is supported via the following:

- a *service interface* supported by the semantic kernel (mentioned earlier in this report (Section 4)),
- a *node interface* supported by nodes in the AST, and
- a *default behavior*, implemented using RAGs.

Most services also require graphical components, of which some can be provided by JedGen as generic components. Each service has its own set of these entities. However, those features needed by all services, or by the semantic kernel, are shared and, hence, defined as a *core service of JedGen*. The core service has no visual components and provides updating of ASTs and lookup of AST nodes.

Core Node Interface The ASTs placed in the AST repository of the semantic kernel may represent several different languages. For JedGen to communicate with the ASTs in a well-defined way the nodes of all ASTs in the repository need to implement a basic interface called `IJedGenNode`, with the following methods:

```
public interface IJedGenNode {

    // Keys and Lookup
    public NodeKey nodeKey();
    public boolean hasResourceConnection();
    public IJedGenNode lookupNode(NodeKey key);
    public IJedGenNode lookupNode(MappingNodeKey key);

    // Updating
    public void flushAttributes();
    public void replaceWith(IJedGenNode node);

    // Position
    public int getBeginLine();
    public int getEndLine();
    public int getBeginColumn();
    public int getEndColumn();
}
```

These methods are needed by the semantic kernel to handle updating and lookup of nodes.

Core Default Behavior All of the methods in the `IJedGen` interface, except the position methods which need specific lexer and parser information, are generic and can be provided with a default implementation. This default behavior is provided in a JastAdd aspect using inter-type declarations and attributes (with signatures matching the interface methods):

```
aspect JedGen {

    ASTNode implements IJedGenNode;

    syn lazy NodeKey ASTNode.nodeKey() = ...;
    syn boolean ASTNode.hasResourceConnection() = ...;

    syn IJedGenNode ASTNode.lookupNode(NodeKey key) {...}
    syn IJedGenNode ASTNode.lookupNode(MappingNodeKey key) {...}

    public void ASTNode.flushAttributes() {...}
    public void ASTNode.replaceWith(IJedGenNode node) {...}
}
```

ASTNode, the top class of the inheritance hierarchy of RAG-based AST nodes supported by JastAdd, is defined to implement the IJedGen interface. The methods of the interface are added using inter-type declarations, that is, by using the ASTNode class name as a prefix to the attribute signature. The interface methods returning a value are implemented as **syn**thesized attributes, which behaves in a fashion similar to methods in Java. The **lazy** keyword defines the nodeKey() attribute to be cached which is useful in case we think an attribute will be called several times. We leave the implementation of the position methods of the IJedGen interface to each semantic editor using JedGen.

Core Service Support The core service interface does not have any service interface of its own but contribute with functionality needed by other services. Most services need to communicate node information via their service interface but are not allowed to return references to AST nodes. Still, they need to communicate node information. To handle this we use a hierarchy of small objects containing a node key and position information, corresponding to that offered by the IJedGenNode interface. At the top of this hierarchy we define an abstract class JedGenNode:

```
public abstract class JedGenNode {

    protected NodeKey key;
    protected int startLine, endLine, startCol, endCol;

    public JedGenNode(NodeKey key, int startLine, int endLine,
            int startCol, int endCol) {
        this.key = key;
        this.startLine = startLine;
        // ...
    }
    public NodeKey getNodeKey() {
        return key;
    }
    public int getStartLine() {
        return startLine;
    }
    // ...
}
```

This class can be extended with service specific subclasses which can be returned via a service interface.

As an example of a service offered by JedGen, we will describe the browsing service in more detail. By browsing, we mean operations such as *find the declaration* of a use or *find the references* of a declaration.

5.2 Example: The Browsing Service

The browsing service has a node interface called `IBrowsingNode` defining methods suitable for extracting browsing information from an AST node:

```
public interface IBrowsingNode extends IJedGenNode {

    public IBrowsingNode declaration();
    public Collection<IBrowsingNode> references();
}
```

The `IBrowsingNode` interface extends the `IJedGenNode` interface with two methods, one for the finding of a declaration and one for the finding of references. Both of the methods return results in terms of the `IBrowsingNode` interface. To further follow the example of the core service, the browsing service has an aspect with default behavior defined on the `ASTNode` class:

```
aspect GenericBrowsing {

    ASTNode implements IBrowsingNode;

    syn IBrowsingNode ASTNode.declaration() = null;
    syn Collection<IBrowsingNode> ASTNode.references() =
        new ArrayList<IBrowsingNode>();
}
```

The default behavior is to return **`null`** and an empty collection. This behavior is not very useful in itself but is intended to be extended by semantic editors using JedGen. Typically, a specialization of the behavior of an AST node in JastAdd is done by defining an **`eq`**uation for an attribute on a subclass of where the attribute was defined. A specialization for a language with two node types defining, for example, variable uses (`Use`) and declarations (`Decl`) may be defined like this:

```
aspect LanguageSpecificBrowsing {

    eq Use.declaration() = decl();
    eq Decl.references() = uses();
}
```

Here, the `decl()` attribute may be a reference attribute pointing to the declaration of the node, and `uses()` may be an attribute returning the uses of a declaration.

The final entity needed to implement the browsing service is the service interface supported by the semantic kernel. We start to define a class (`BrowsingNode`) for returning of service results:

```
public class BrowsingNode extends JedGenNode {

    public BrowsingNode(IBrowsingNode node) {
        super(node.nodeKey(),
            node.getBeginLine(), node.getEndLine(),
            node.getBeginColumn(), node.getEndColumn());
    }
}
```

The `BrowsingNode` class only extends the abstract class `JedGenNode` without adding any additional information. As the final step, we use this class to define a simple service interface provided by the semantic kernel:

```
public class SemanticKernel  {
    // ...
    public BrowsingNode findDeclaration(MappingNodeKey fileKey) {
        // ...
    }
    public Collection<BrowsingNode>
        findReferences(MappingNodeKey fileKey) {
        // ...
    }
}
```

A find declaration, or find references, operation is activated with a text selection in an editor instance as input. In order to handle text selections (offset intervals) we define that the interface should make use of a mapping node key.

6 Current Status and Future Work

Recent efforts have been made to make the implementation of JedGen more robust, involving several refactoring phases. The core ideas have stayed the same but the implementation has become more harmonized with the mechanisms in Eclipse. Currently, service implementations are being migrated to the latest version. There are numerous ways to continue to improve this work. Some examples include:

- *Improving performance*

  Synchronization and flushing could both possibly be done on a lower level than the project AST level.

  A possible more *fine-grained synchronization* scheme would be to have mutual exclusion of subtrees in the project AST. These locks would have to have a notion of the tree hierarchy in order to work. For example, an update of the project AST should not be possible when one of its file ASTs are locked.

  A more *fine-grained flushing* scheme requires incremental updating of RAGs which is an unsolved problem and an interesting research direction. In our case, we would like a means for flushing of attributes affected by the update of a certain subtree in the AST.

- *Handling of libraries*

  Currently, there is no special support for handling of libraries in JedGen. Libraries can, for example, be considered as resources on the Eclipse platform and stored as project ASTs, or similar, in the AST repository of the semantic kernel. However, to link uses and declarations in a project and a library there would need to be references between two project ASTs. One solution to this problem is to add another level enclosing the project ASTs in the AST repository, a *workspace AST*. The down side of this solution is that it adds another path segment to all node keys. Another solution is to add a new mechanism which can handle *inter-AST references* in a controlled way between project ASTs in the semantic kernel. In order to comply with the prevention of not letting direct references of AST nodes outside of the semantic kernel, this mechanism would be added *inside* the kernel.

- *Providing feedback between builds*

  The presented feedback cycle reacts to changes when a user saves a file. Preferably, the editor should provide fresh feedback also in between saves. Eclipse offers support for *light-weight reconciling* of files which can be used to keep up-to-date with user changes on a more fine-grained level. One possible extension of JedGen is to support a *shorter feedback cycle* using reconciling of the current file being edited.

7 Acknowledgements

We are very grateful to Torbjörn Ekman, for guidance and support during the early stages of the development of JedGen, and to Max Schäfer and other early users of JedGen.

Bibliography

[1] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Development of a Modelica compiler using JastAdd. In *Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 117–131. Elsevier B.V., April 2008.

[2] Oracle Corporation. NetBeans IDE, 2010. `http://netbeans.org/` [Accessed May 2010].

[3] Maartje de Jonge, Emma Nilsson-Nyman, Lennart C. L. Kats, and Eelco Visser. Natural and flexible error recovery for generated parsers. In Mark van den Brand, Dragan Gasevic, and Jeff Gray, editors, *SLE*, volume 5969 of *Lecture Notes in Computer Science*, pages 204–223. Springer, 2009.

[4] J. Des Rivières and J. Wiegand. Eclipse: a platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[5] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM.

[6] Torbjörn Ekman and Görel Hedin. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69(1–3):14–26, December 2007.

[7] The Eclipse Foundation. The Eclipse Platform, 2010. `http://www.eclipse.org/` [Accessed May 2010].

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[9] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.

[10] Görel Hedin and Boris Magnusson. The Mjølner Environment: Direct interaction with abstractions. In Stein Gjessing and Kristen Nygaard, editors, *ECOOP*, volume 322 of *Lecture Notes in Computer Science*, pages 41–54. Springer, 1988.

[11] Daqing Hou and Yuejiao Wang. An empirical analysis of the evolution of user-visible features in an integrated development environment. In *CASCON '09: Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 122–135, New York, NY, USA, 2009. ACM.

[12] JetBrains. IntelliJ IDEA, 2010. `http://www.jetbrains.com/idea/` [Accessed May 2010].

[13] Donald E. Knuth. Semantics of context-free languages. *Journal Theory of Computing Systems*, 2(2):127–145, June 1968.

[14] Matthijs F. Kuiper and João Saraiva. Lrc - a generator for incremental language-oriented tools. In Kai Koskimies, editor, *CC*, volume 1383 of *Lecture Notes in Computer Science*, pages 298–301. Springer, 1998.

[15] Jesper Mattsson. The JModelica IDE: Developing an IDE reusing a JastAdd compiler. Master's thesis, Lund University, Lund, Sweden, August 2009.

[16] Erik Mossberg. Inspector – tool for interactive language development. Master's thesis, Lund University, Lund, Sweden, October 2009.

[17] Philip Nilsson. Semantic editing compiler extensions using JastAdd. Master's thesis, Lund University, Lund, Sweden, June 2010. To be presented.

[18] Emma Nilsson-Nyman, Torbjörn Ekman, and Görel Hedin. Practical scope recovery using bridge parsing. In Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk, editors, *Proceedings of the Internation Conference on Software Language Engineering (SLE 2008)*, volume 5452 of *LNCS*, pages 95–113, 2008.

[19] Emma Nilsson-Nyman, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. Declarative intraprocedural flow analysis of Java source code. In *Proceedings of the Eight Workshop on Language Description, Tools and Applications (LDTA 2008)*, Electronic Notes in Theoretical Computer Science. Elsevier B.V., 2008.

[20] Thomas Reps and Tim Teitelbaum. The Synthesizer Generator. In Peter B. Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 19 of *SIGSOFT Software Engineering Notes*, pages 42–48, Pittsburgh, Pennsylvania, USA, May 1984. ACM.

[21] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for Java. In Gail E. Harris, editor, *OOPSLA*, pages 277–294. ACM, 2008.

[22] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping stones over the refactoring rubicon. In Sophia Drossopoulou, editor, *ECOOP*, volume 5653 of *Lecture Notes in Computer Science*, pages 369–393. Springer, 2009.