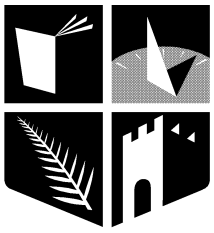
A test-driven development strategy for the construction of grammar-based software.

by

Mark Hennessy



NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Dissertation submitted in partial fulfillment of the requirements
for candidate for the degree of

Doctor of Philosophy

Department of Computer Science,
National University of Ireland, Maynooth, Co. Kildare, Ireland.

Supervisor: Dr. James F. Power
October 2006

Contents

Declarations

I confirm this is my own work and the use of all material from other sources has been properly cited and fully acknowledged. Part of the work in this thesis has been presented in the publications listed in Section 1.4.

National University of Ireland Maynooth — Mark Hennessy
October 2006

Acknowledgements

First and foremost, I must offer my sincere thanks to my erudite supervisor Dr. James F. Power. His support, vision, motivation, and enthusiasm ensured that I was able to conduct and most importantly complete this work within a very encouraging environment. I am very grateful.

Secondly, I would like to thank my folks for their unwavering support; both emotionally and occasionally financial over the past number of years, they have shared both the good times and not so good times with me.

I would like to thank all of my friends, too many to mention, but I'll focus on the "25 to life" group. Gaz, Karol, Pnut and Steve, thanks for all of the good times we've had and lets hope for many more. To the rest of the group and the Dancers, I say that this year will be my year to add the coveted title of "Family Feud" world champion to my extensive canon of titles!

I would also like to acknowledge my wonderful housemates; Aidan, Chris and Joe. I couldn't ask to live with a nicer bunch of people.

Finally, I would like to offer special thanks to Kitty. She has had to deal with me more than anyone else over the years and her support and patience has been tremendous. Thank You.

For Uncle Frank.

Abstract

Grammar-based software is increasingly becoming a prominent and well defined subset of software engineering through the popularity of analysis tools, metrics evaluators and the increasing prevalence of software tools that rely upon extensible data such as XML. Software-testing plays a crucial role in the lifecycle of any modern software system, hence the correct and adequate testing of grammar-based software is essential.

This thesis provides a review of the current research and practice in the field of grammar-based software. The motivation behind software-testing is examined and a summary of existing testing techniques is presented. This leads to a detailed empirical comparison of test-suites for grammar-based software. The effectiveness of test-suite generation via methods such as Purdom's algorithm is contrasted with test-suite reduction via a novel elaboration of the notion of coverage for a grammar. The effectiveness of this strategy is discussed with regard to two testing analysis criteria: code coverage and fault detection.

The second contribution of this thesis is the definition of a framework for developing grammar-based software within a strict test-driven environment. We describe the techniques used to develop a static analysis tool for the ISO C++ grammar and the verification of its correct operation via standardised test-suites. This work was completed in two distinct phases.

The first phase involves porting an existing system from C++ to Java by combining eXtreme Porting with an order for porting derived from a novel use of Object Relation Diagrams. The second phase involves a large automated testing process including black-box testing, coverage comparisons and comparisons of reverse-engineered UML sequence diagrams from program traces.

The third contribution of this thesis involves the utilisation of the generalised parsing algorithm, GLR, in the implementation of a parser generator. We describe the design and test-driven development of this parser generator, and its practical application as part of an analysis tool for ISO C++. Finally, we provide a study of the performance of a generated GLR parser when parsing ISO C++ programs, and a comparison with an existing system based on a generated backtracking parser.

Chapter 1

Introduction

In this chapter we present an outline of the motivations behind this research and the contributions provided by this work. We also provide an overview of the structure of this thesis.

1.1 Motivation

The term *grammar-based software* describes software whose input can be specified by a context-free grammar [KLV05]. This grammar may occur explicitly in the software, in the form of an input specification to a parser generator, or implicitly, in the form of a hand-written parser, or other input-verification routines. Grammar-based software includes not only programming language compilers, but also tools for program analysis [Ana06], reverse engineering [Sci06], software metrics [Tec06] and documentation generation [vH06]. Such tools often play a crucial role in automated software development, and ensuring their completeness and correctness is a vital prerequisite for their use.

The grammars that define the language these tools receive as input can be either implicit or explicit. An implicit grammar is one where the specification is embedded in the code. This grammar can be difficult to obtain as it requires access to the source code of a system. An explicit grammar represents a standalone entity

that is used to describe the language.

Tools to support program comprehension are typically built around a *parser front-end*, which is used to process the input and build up a model of that input. The program comprehension tool can then perform its operations based upon the model built up by the parser front-end.

A parser-front end is typically built automatically by passing an explicit grammar as input to a *parser generator*. The parser generator expects a grammar to be in a specific subset of the context-free grammars, such as those amenable to LL or LR parsing algorithms, but not every programming-language grammar falls neatly into this subset. For example, the grammar described by the ISO C++ standard [ISO03], and more recently for the C# standard [ISO06], indicate a move away from grammars that are easy to process using existing tools. Given the popularity of the C++ programming language, and its inherent complexity, it is vital that automated tools for processing the language be robust and accurate. Therefore support for the automated creation of parsers of C++ and other languages of similar complexity is essential. Facilitating the automatic generation of parser front-ends for complex grammars like the ISO C++ and C# is a principal motivation for this thesis.

In this thesis we describe the development of a parser generator capable of producing useful parser front-ends for complex programming languages. We demonstrate the robustness and practical utility of the parser generator, by using it to implement a new parser for the ISO C++ analysis tool, keystone [PM00, GMP03a].

keystone is itself written in C++ and contains a generated parser front-end which is used to drive the back-end, which is responsible for building up a symbol table for an input program. Our work focuses on using best-practice software engineering techniques to create a Java version of keystone via test-driven development [Ast03, Bec02]. Specifically, we apply these techniques to port the keystone back-end to Java and unify this Java version of the back-end with a generated parser from the new parser generator.

1.2 Contributions of the Work

There are four main contributions emanating from our work.

We outline the results of a large scale empirical study into the creation of a standardised minimal test-suite for applications that receive ISO C++ programs as input. We present a case study outlining the effectiveness of these minimised test-suites with respect to both code-coverage and fault detection for three applications that process C++ programs.

We then describe a new technique that facilitates the porting of a system from one programming language to another by specifying the order in which the classes can be ported. This order reduces the amount of class stubs needed during the unit-testing stage of the development. We also outline an automated strategy for both black- and white-box testing the ported system. The white-box testing stage relies upon the comparison of reverse-engineered dynamic UML sequence diagrams generated through the use of aspect-oriented programming. We apply this strategy to the porting of the keystone back-end from C++ to Java.

The test-driven development of a new GLR parser generator for Java is then described. We have enhanced an existing LALR parser generator, JavaCup, to generate GLR parsers. We demonstrate the robustness of this GLR extension through the utilisation of a test-suite containing grammars for a variety of the popular current programming languages including dialects of C++. We test the generated parsers by using Purdom's algorithm to generate test-cases from a variety of grammars [Pur72].

The final contribution of the work is creation of a working GLR parser front-end for C++ with semantic actions fully enabled. We validate the applicability of the GLR algorithm for parsing ISO C++ by plugging in the generated GLR parser into the ported back-end to create a Java version of keystone, known as jKeystone. We demonstrate that jKeystone has similar functionality but greater robustness than keystone.

1.3 Structure of the Thesis

In the next chapter we outline the background work related to the parsing of a programming language. We also describe the testing techniques used throughout this thesis. Chapter three presents a case study into the effectiveness of rule-coverage as a criterion in the reduction of test-suites for grammar-based software.

The fourth chapter is a description of the porting process used to port the back-end of keystone to Java. The automated processes used to test the Java back-end are also outlined.

The work involved in modifying an existing parser generator to produce GLR parsers is presented in chapter five. We also describe the test-suites and testing goals used to validate the correct operation of the GLR algorithm for many large grammars of popular programming languages.

Chapter six then completes the research contained within this thesis. The strategy for allowing user-defined semantic actions in a generated GLR parser is described. A working GLR parser front-end is utilised as the front-end of jKeystone and compared against the original keystone. Chapter seven offers some concluding remarks and describes potential of future work.

1.4 Publications based on this thesis

Chapter 3 was presented as a full paper at the 20^{th} IEEE/ACM Conference on Automated Software Engineering [HP05a].
Chapter 4 was presented at the IBM Centre for Advanced Studies Conference, Dublin symposium [HP06], where it received the best paper award.

Chapter 2

Related Work

In this chapter we provide an overview of the previous research that is related to the work presented in this thesis. We break this down into two major sections: parsing and testing. We first present an overview of grammars and parsing algorithms and discuss in detail the ISO C++ grammar and the difficulties associated with parsing ISO C++ programs. We then present an overview of the testing techniques used in the remainder of this thesis.

The ISO C++ grammar specification defines the grammar for the C++ programming language [ISO98, ISO03]. ISO C++ is an object-oriented programming language with full programming support for both low level access to hardware and backward compatibility with C, and high level concepts via generic programming techniques. The versatility of the language makes it a popular language for development, especially in an industrial context. Given the popularity of the language, it is vital that tool support is provided to aid not only the creation of C++ applications but also to aid in the automated reverse engineering and comprehension of C++ programs. Without the support of automated tools, maintaining and re-engineering C++ programs can be very difficult.

A major problem with creating automated tools for comprehending C++ programs is that the more popular tools to create parsers [Joh75, GNU06] are practically unchanged over the past 30 years and have yet to catch up with the technol-

ogy needed to parse grammars with ambiguities such as C++ and C#. The process of creating a parser for a language is based upon algorithms developed in the 1960's and with a few exceptions [PQ95, vdBvDH+01], development within the field has not kept pace with newer developments. Newer techniques exist that are in theory able to comprehend a complicated language such as C++ and it is a major goal of this thesis to demonstrate their feasibility in the construction of tools that analyse ISO C++ programs.

2.1 Parsing

Parsing, also known as syntax analysis, is the process of analysing an input sentence in order to determine its structure with respect to a given grammar [ASU86, GJ90]. The input sentence consists of individual units known as tokens. The parsing of an input sentence is therefore based around the transformation of the input tokens into some data representation, typically a tree. This tree is known as a parse tree and it is used to represent the hierarchy contained within the input sentence.

2.1.1 Grammars

Formally, a grammar is a four-tuple (N, T, S, P) where N and T are disjoint sets of symbols known as non-terminals and terminals respectively, S is a distinguished element of N known as the start symbol, and P is a relation between elements of N and the union and concatenation of symbols from $(N \cup T)$, known as the production rules. Some grammars further define another special symbol, ε, to represent the empty string. The ε symbol can only occur on the right-hand side of a rule.

The grammar's production rules may be read as rewrite rules, thus specifying alternative ways of re-writing the start symbol to a sequence of terminal symbols, known as the sentences of the language. In programming language terms, these sentences are programs that conform to the grammar of the language.

2.1.2 Parsers

To aid in the process of parsing, special tools known as parsers can be employed. Parsers typically use one of two distinct types of parsing algorithm, either top-down or bottom up. Top-down parsing works by starting at the distinguished start symbol and expanding all of the non-terminals from left to right until only terminals remain. Bottom-up parsing operates by matching an entire right-hand side of a rule and replacing it by the left-hand side non-terminal of that rule. This process is repeated until the distinguished start symbol replaces the input.

Constructing a top-down parser by hand is a conceptually easy task as each non-terminal can be mapped to a procedure in the code. Grammar operators such as choice and repetition map naturally to programming language constructs. However, it is the ability to predict which grammar-rule to use during a parse that is central to most parsing algorithms.

Predictive parsers use a special table, known as a *parse table* that can be consulted at any time to determine which rule to use. This table must determinise the choice of production rule, that is, there can only be a single entry in each index of the parse table. The most common classes of predictive parsing algorithms are the LL parsers for top-down parsing, and the LR parsers for bottom-up parsing.

LL Parsing

LL is defined as the left-to-right reading of the individual tokens of the sentence and the construction of a left-most derivation of the parse-tree. The parse table for an LL parser is a two dimensional array, indexed by non-terminal and terminal symbols. LL parsers have a major limitation in that they must choose which grammar rule to use, having only seen k tokens, typically 1, of the right hand side of a rule.

With the exception of Pascal [ISO83], no modern programming language specification can be parsed by a simple LL(1) parser, hence a more powerful technique to predict which rule to use is desirable. This can be achieved through

localised syntactic lookahead where the next *n* tokens are consulted in order to help determinise the parse. It is also possible to use another strategy known as semantic lookahead, which allows the user to program specific decision-making code into the rules. This strategy is employed by both top down parsers `ANTLR` [PQ95] and `JavaCC` [Jav].

LR Parsing

Bottom-up predictive parsing is typified by the LR parsing algorithm, which stands for left-to-right reading of the input using a right-most derivation. An LR parser postpones the decision of choosing a grammar rule until an entire right-hand side of a grammar rule has been matched. The parser uses a stack to store intermediate results when reading the input sentence. Based on the contents of the top of the stack and the current token in the input, it is possible to perform one of two kinds of actions:

- **Shift**: Push the current input symbol onto the top of the stack.
- **Reduce**: A rule of the form $A \rightarrow \alpha$ has been matched. Pop α off of the stack and push A onto the top of the stack.

Parsing is complete when the entire input has been reduced to the start symbol.

To determine when to shift and when to reduce during an LR parse, it is usual to construct a table that can be consulted during the parse. This table is typically known as a *parse table* and is indexed by an LR *state* number and a terminal index.

An LR state consists of one of more *items*. Each item is of the form ($A \rightarrow \alpha_{\triangle}\beta$, *l*), where α is at the top of the stack, the marker, $\triangle$, represents the current position in the right-hand side of a production, and *l* is the current lookahead which consists of a set of terminals. A new state is created by "moving the marker" one position to the right in an item, and this behaviour corresponds to a shift action in the case of a terminal or a goto action in the case of a non-terminal. If, in a given state, we have an item of the form ($A \rightarrow \alpha\beta_{\triangle}$, *l*), then we can perform a reduction for that item if the next input symbol is in the lookahead set *l* of that item.

$$A \rightarrow \beta \triangle \{S1\} \qquad A \rightarrow \beta \triangle \mathbf{a}\, \gamma \{S1\}$$
$$B \rightarrow \theta \triangle \{S2\} \qquad B \rightarrow \theta \triangle \{S2\}$$

(a) Reduce-Reduce Conflict (b) Shift-Reduce Conflict

Figure 2.1: Conflicts appearing in the item-sets. *The structure of conflicts that can appear in the states of an LR parser. A reduce-reduce conflict happens if* $S1 \cap S2 \neq \emptyset$*, and a shift-reduce conflict happens if* $\mathbf{a} \in S2$*.*

During the construction of the parse tables, it is possible to encounter item sets similar to those shown in Figure 2.1. In Figure 2.1(a), a state containing two distinct items is shown. Both items have their marker at the end of the item. When an LR state is in this configuration and $S1 \cap S2 \neq \emptyset$, both reduces are *equally* valid under the shared lookahead symbols. This scenario is known as a *reduce-reduce* conflict. Similarly in Figure 2.1(b) there are two items shown. The first item shows a marker in the middle of a right-hand side with the terminal **a** still to be pushed onto the stack and the other item has its marker at the end. In this configuration there is another type of conflict present if $\mathbf{a} \in S2$. This conflict is known as a *shift-reduce* conflict and it occurs when a decision cannot be made as to whether to shift **a** onto the stack as indicated by first item or to reduce the second item in its entirety and push *B* onto the stack.

The creation of an LR parse-table for a given grammar has a well established algorithm [Knu65] that is conceptually easy to follow but consists of many repetitious stages. Fortunately there exist automated tools known as *parser-generators*, that implement this algorithm to create parse-tables and parsers automatically.

Parser Generators

Parser generators such as yacc [Joh75] and GNU bison [GNU06] work by generating LALR(1) parse-tables along with a generic driver program. This driver program is responsible for the run-time operation of the parser such as consulting the parse tables, shifting symbols onto the stack and popping symbols off the stack

when a full right-hand side of a rule has been recognised. The driver code is also responsible for communicating with the *lexer* which splits up the input sentence into its individual terminal symbols as defined in the language specification.

Parser generators are a well established field of computer science with the most popular tools almost unchanged for almost 30 years. Simple parser generators typically expect the grammar to be free of conflicts. If a grammar does contain conflicts, the standard LR parsing algorithm does not allow for any form of non-determinism. This means that for each index into the parse table in a given state with any given input symbol there is exactly one parse action to execute.

To implement this approach, parser generators take steps during the processing of the grammar to ensure that the final tables are free of conflicts. For example in yacc, if a shift-reduce conflict is encountered, then the usual step taken is to always prefer the shift over the reduce. For a reduce-reduce conflict, the rule with the highest user-defined precedence or the rule appearing first will always be preferred. Thus if a grammar contains many conflicts, valid parse configurations will potentially be discarded during the automated creation of the parse tables. This method of overcoming ambiguity within a grammar specification is not so much of a problem for a language like C whose grammar contains a solitary conflict [ISO90]. However, for a modern programming language such as ISO C++ the construction of a parser using automated parser generation tools remains a significant challenge.

2.2 The ISO C++ Grammar

The C++ programming language was standardised by the International Standards Organization (ISO) in 1998 [ISO98], and further updated in 2003 [ISO03]. Appendix A of the ISO standard contains a grammar for the language, with 123 non-terminals, 184 terminals and explicitly specifying 399 grammar rules. The notation used for the rules permits optional symbols in the productions; when these are replaced systematically by expanding optional grammar rules, this rises

typedef-name	→	**IDENTIFIER**
original-namespace-name	→	**IDENTIFIER**
class-name	→	**IDENTIFIER**
	|	template-id
enum-name	→	**IDENTIFIER**
template-name	→	**IDENTIFIER**

Figure 2.2: The five context-sensitive identifiers within the ISO C++ grammar. *These grammar rules add to the number of conflicts present in the grammar, as they create a five way reduce-reduce conflict in any LR state that contains these items.*

---

to 479 rules using plain context-free notation.

This grammar is significantly more complex than that for other popular programming languages such as C and Java [PM04]. There are many features within the grammar that make parsing using a generated parser very difficult. The first major difficulty is the definition of five *context-sensitive* identifiers within the grammar, as illustrated in Figure 2.2. These context-sensitive identifiers add many reduce-reduce conflicts to the grammar as an identifier encountered during a parse may need to be reduced to any one of the five grammar rules shown. Deducing this reduction from purely syntactic information alone is impossible.

Another cause of conflicts within the grammar is the fact that the grammar contains *ambiguities*. An ambiguity occurs when a single input sentence can have two or more *valid* meanings. There are a number of ambiguities within the C++ grammar which add an extra level of complexity to parsing C++ [KLDM99]. Some of these ambiguities are legacies carried over from the fact that C++ was designed to be backwardly-compatible with C, others are due to the introduction of templates.

In Figure 2.3, an example of two types of ambiguity is shown. In Figure 2.3(a), there is an ambiguity between a pointer declaration and a multiplicative expression. The code on line 4 shows the declaration of a pointer to a type `C`, named `a`. The next line of code shows the multiplication of the integers `T` and `b`. As can be seen from this example, there is no syntactic difference in the code. In

```
1:  class C;
2:  int T;
3:  int b;
4:  C * a;//declaration
5:  T * b;//expression
```

(a)

```
template <typename T> A;
int B, X, C;
class Y;
B < X > C;//boolean expression
A < Y > D;//template instantiation
```

(b)

Figure 2.3: Some ambiguities within C++. *In Figure (a) there is an ambiguity between a pointer declaration and multiplication. In Figure (b) there is an ambiguity between a template instantiation and a boolean expression.*

Figure 2.3(b), line 4 shows a boolean expression involving a comparison between integers `B`, `X` and `C`. The code on line 5 then shows template `A` being instantiated as `D` with a template parameter of `Y`. In this case again, the code fragments are syntactically identical.

2.2.1 Parsing C++

There have been a number of attempts to construct a C++ parser over the past fifteen years but to date the construction of a fully ISO conformant *public-domain* parser front-end has been difficult.

The C++ grammar developed by Jim Roskind in 1989 was developed before a grammar for C had even been standardised [Ros89]. Consequently, this early effort lacks support for any of the advanced features of C++ such as namespaces, exceptions and templates. John Lilley attempted to construct a C++ parser using a top-down approach and the parser generator tool `ANTLR` [JL97]. This was a more complete effort than Roskind's but still did not offer full support for templates.

Other research efforts at developing a C++ front-end include CPPP, an early attempt to create an object-oriented front-end for C++ [RD95]. CPPP uses a C++ grammar that has limited support for templates, which makes in impractical for use with modern C++ programs. Another front-end developed was OpenC++ [Chi95]. This was developed with the goal of enabling meta-programming in C++ to easily

allow program transformations. Unfortunately OpenC++ has only limited support for templates, which are a crucial aspect of the language.

The aspect-oriented extension for C++, AspectC [SLU05] utilises Puma, a library of classes written in C++ for parsing and manipulating C++ source code [PUM] for its parser front-end. Puma uses a customised hand-written recursive descent parser to parse the language. While Puma works quite well for AspectC, it still cannot handle all C++ constructs yet.

In addition to these research systems, there are a number of commercial C++ front-ends available:

- **EDG C++ front-end**
  The Edison Design Group have developed a commercial C++ parser front-end that offers full support for all of the language features in the ISO standard [EDG]. It is a hand-written parser consisting of approx 440,000 lines of C code. The EDG front-end has been used successfully by many leading commercial software companies.

- **DMS**
  DMS is a commercial system offered by the *Semantic Designs* company [BPM04]. The DMS toolkit allows for the construction of parser front-ends for C++ and other languages. DMS also includes support for program transformations and clone detection within program source code.

- **Columbus**
  Columbus is a tool support the reverse engineering of C++ programs [RAMT02]. It is based upon a commercial C++ front-end, developed by *FrontEndArt*. Columbus allows for the extraction of facts from C++ to aid in visualisation or interoperability with language exchange formats such as GXL [Win01].

The open-source compiler gcc handles a variety of programming languages, among them C++. As of version 4.0 of gcc, the C++ compiler has been based upon a hand written top-down parser. This is in marked change to all of the previous releases which had relied upon a parser generated by the parser generator bison.

This latest parser uses syntactic lookahead and back-tracking to successfully parse the ambiguous constructs of C++. Furthermore this parser conforms very closely to the published ISO standard. It is however, difficult to de-couple the C++ parser front-end of gcc from the compiler internals. Hence constructing other analysis tools from the gcc front-end is a difficult task.

2.2.2 keystone

The keystone system is a parser and front-end for ISO C++. It is written entirely in C++, and consists of 40 classes and roughly 23,000 lines of code with many inter-class dependencies and inheritance hierarchies [PM00]. The front-end of keystone consists of a tightly coupled lexical analyser and *backtracking* bottom-up parser, generated by the parser generator btyacc [Sib03]. The back-end of keystone consists of a set of semantic routines called by the parser to maintain a symbol table, and to represent the scope and type information contained in the input program.

The backtracking algorithm allows conflicts to remain in the parse table. When a conflict is encountered during a parse, one of the possible conflict branches continues with the parse. If this branch results in an error, the parser backtracks to the conflict point to try the alternative parse. If the branch is valid then the parse continues as normal.

keystone uses a technique known as token-decorated parsing to help overcome the difficulties in parsing the ISO standard [GMP03a]. This technique allows an identifier to be marked up before it is passed to the parser. Thus the context-sensitive identifiers that are present in the standard can be identified by storing them in a buffer that allows their type to be *decorated* by changing the type of the token before being passed to the parser. This reduces the amount of ambiguity in the grammar and facilitates the parsing.

The keystone system was designed to have a well defined border between the parser front-end and the back-end used to perform the semantic analysis. Thus, in theory it becomes possible to de-couple the existing front-end and replace it with

```
int x;
int f (int);
typedef int g;
main(){
 f(x); // expression
 g(x); // declaration
}
```

Figure 2.4: An example of the declaration expression ambiguity in C. In this piece of C code, `f(x)` and `g(x)` are syntactically identical, yet one is an expression and the other a declaration.

---

another. However this facility has not been previously exploited.

2.3 Generalised Parsing

As the ISO C++ grammar specification shows, a grammar may give rise to many conflicts when using a predictive parsing algorithm. Furthermore there may be *ambiguities* in the grammar. This means that it is possible for a valid sentence of the grammar to have two or more possible legal parses for a single input.
A well known example of ambiguity is the declaration/expression ambiguity in C and C++ that is illustrated in Figure 2.4. In this example, the method call to `f(x)` is an expression while the code `g(x)` represents a declaration of `x` as a variable of type `g` with redundant parenthesis. A parse of this input without any access to type information results in the acceptance of two valid syntactic parses. A method of overcoming this ambiguity is to allow the lexer to access type information contained in the symbol-table during the parse. This allows the identification of `g` as a type definition and the parser then recognises `g(x)` as a declaration. Unfortunately the addition to C++ of features such as forward declarations, templates and context-sensitive keywords ensures that the use of the lexical-feedback mechanism results in a very tightly coupled lexer and parser. This design makes the

maintenance of the parser very difficult as the lexical and syntactic phases cannot be readily separated. This lexical feedback approach is slightly different to the notion of keystone's token decorated parsing. With token decoration, there is a separate subsystem responsible for identifying the token's types. The lexer has no access to the symbol table and is solely responsible for the lexical analysis.

Algorithms to parse grammars that contain ambiguities *do* exist but have yet to enter the mainstream of the compiler community, either through a lack of tool support or through the belief that the algorithms themselves incur a considerable overhead.

2.3.1 Generalised Parsing Algorithms

The earliest general parsing algorithms were devised by Earley [Ear70] and by Cocke, Younger and Kasami who devised the parsing algorithm known as CYK [GJ90]. Both of these algorithms are capable of parsing an arbitrary context-free grammar through the use of dynamic-programming techniques. However both algorithms have been proved to operate in $O(n^3)$ time, where n is the number of tokens in the input sentence. As this is far costlier than LL or LR predictive parsing and in the absence of complicated programming languages at the time, both of these algorithms have never gained a foothold in the mainstream software development process.

Tomita develops the idea of *generalised* LR (GLR) parsing [Tom85]. The GLR algorithm extends the notion of deterministic shift-reduce parsing by allowing multiple parsers to co-exist. The multiple parsers are maintained using a special data structure which Tomita named a *Graph-Structured Stack* (GSS) [Tom85]. This algorithm is an improvement on the original general parsing algorithms because the algorithm takes the large deterministic core of most grammars into account and only spawns multiple parses at specific points of ambiguity. This means that for deterministic sections of input, the GSS behaves identically to that of a typical stack in an LR parse, hence during deterministic operation, the algorithm has a similar time cost to the established bottom-up parsing algorithms.

The original algorithm contained a bug that saw it fail to terminate on ε-grammars in the presence of hidden recursion. Farshi describes a GLR recogniser that corrects the error of Tomita [NF91], although Farshi's solution results in a loss of efficiency over Tomita's algorithm. Farshi's algorithm is taken to be the first *correct* GLR implementation [SJ06].

There have been a number of implementations of the GLR algorithm for various parsing purposes. It has been used for natural language processing in [Lav96], which defines a variant called GLR*. The algorithm has been used in the development of an incremental parser [WG97] for C and C++. A modification to the basic algorithm to speed up the process of parsing is described in [AHJM01]. The authors separate the functionality of the nodes in the GSS into two distinct types; one for keeping track of the state and symbol, the other for pointing back to nodes in the GSS. Finally GLR has also been used in the creation of a parser generator for the functional programming language Haskell [dJKV99].

A more recent extension to GLR known as *right-nulled* generalised LR parsing (RNGLR) has been proposed [SJ06] following on from their earlier work [JS03, JSE04]. The main thrust of this proposal is to create reduction entries in the parse table at every position beyond the final non-terminal that does not derive an ε-rule, not just when the marker is at the end of the rule. For a given rule $\rho \rightarrow \alpha\beta\gamma$, where β and γ are non-terminals capable of deriving ε, reduce entries are made not only for $\rho \rightarrow \alpha\beta\gamma_{\triangle}$ but also for $\rho \rightarrow \alpha_{\triangle}$ and $\rho \rightarrow \alpha\beta_{\triangle}$. The authors claim a speed-up in parsing when using RNGLR over traditional GLR.

2.3.2 The GLR Parsing Algorithm

The GLR algorithm takes advantage of the fact that grammars often contain a large deterministic core with only a small amount of ambiguity. By extending the LR parsing algorithm to handle non-determinism within the tables, it is possible to parse grammars that contain genuine ambiguities.

The GLR algorithm starts as an ordinary LR parser, but upon encountering a shift-reduce or a reduce-reduce conflict within the parse tables it splits up into as

stmt	::=	expr **;**
	|	decl
expr	::=	**ID**
	|	**T (** expr **)**
decl	::=	**T** declarator **;**
declarator	::=	**ID**
	|	**(** declarator **)**

Figure 2.5: A vastly simplified C Grammar exhibiting the "expression/declaration" ambiguity.

---

many new parses as there are conflicts. To achieve this a GSS is utilised.

The GSS is broadly similar to the parse stack of a standard LR parser except that the GSS maintains a stack node that contains a state and a *set* of links to other stack nodes one level lower on the stack. Whenever a conflict is encountered in the parse table, the GSS is split into *n* branches if there are *n* conflicts present. For each token in the input sentence, every possible reduce is applied before all of the tops of the GSS are synchronised on the next shift of an input token. Stacks that are created due to a conflict naturally share the same common prefix, and the algorithm also avoids proliferation of parse states during a conflict by merging branches as soon as two tops enter the same state. If the input is ambiguous then multiple parse trees are represented by the GSS, otherwise the conflict is a *local ambiguity* and is usually resolved when more input has been read.

As an example of the operation of the GSS, consider the grammar shown in Figure 2.5, vastly simplified from the ISO C++ grammar that exhibits similar behaviour to the example in Figure 2.4. This grammar produces the LR(1) state machine that is shown in Figure 2.6. Each state contains a number of items with with the marker at varying locations on the right hand side of each item. Each transition between a state is uniquely labelled with either a terminal or a non-terminal and any state where the marker has moved past the end of file marker, $, is considered to be an accepting state. This state machine contains one reduce-reduce conflict in state 12, with a choice of reducing **ID** to either a *declarator* or

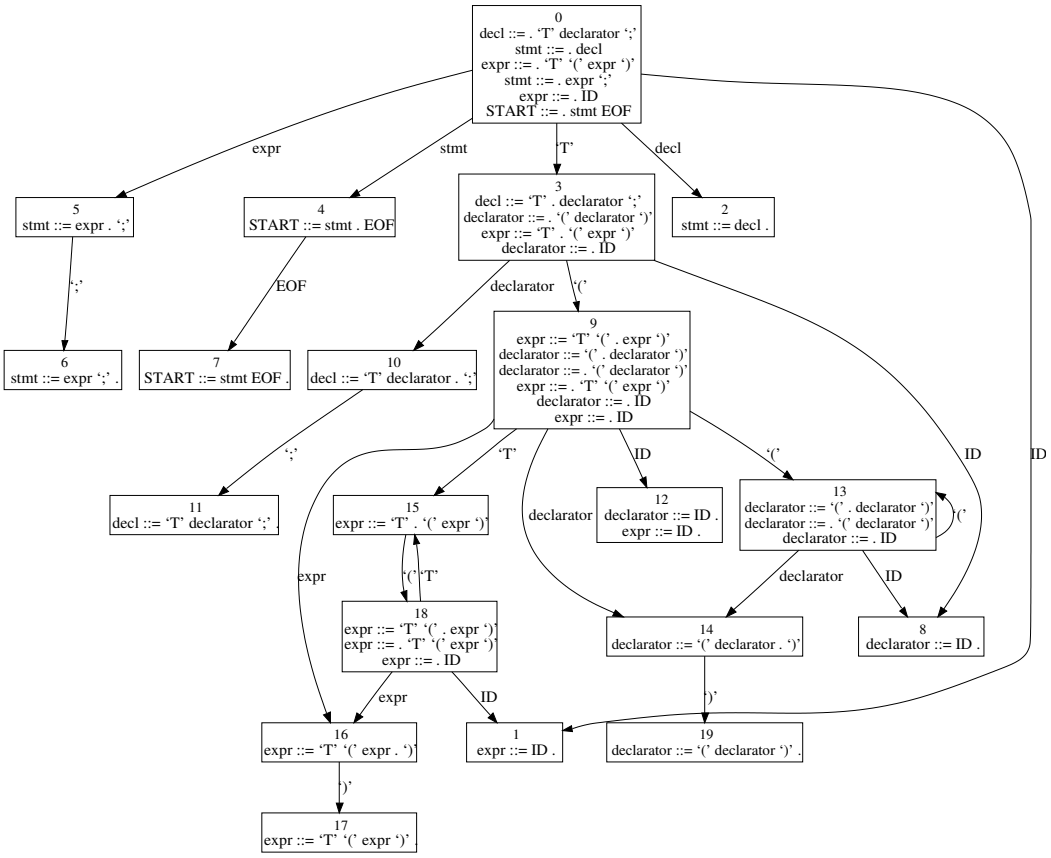


Figure 2.6: The LR(1) state machine for the grammar described in 2.5. *The start state of this machine is state 0, and state 7 is the accepting state.*

---

an *expr*.

A full GLR parse for the input "`T(x);`" is illustrated in Figure 2.7. The parse begins as for the usual shift-reduce parsing algorithm. The tokens `T`, `(` and `x` are all pushed onto the stack to give the stack configuration with states 0, 3, 9, 12 as shown in Figure 2.7(a).

In state 12, there is a reduce-reduce conflict between *declarator* and *expr*. In traditional table-based parsing, the conflict would have been resolved at the generation stage by choosing one of the reduces and marking a single reduce

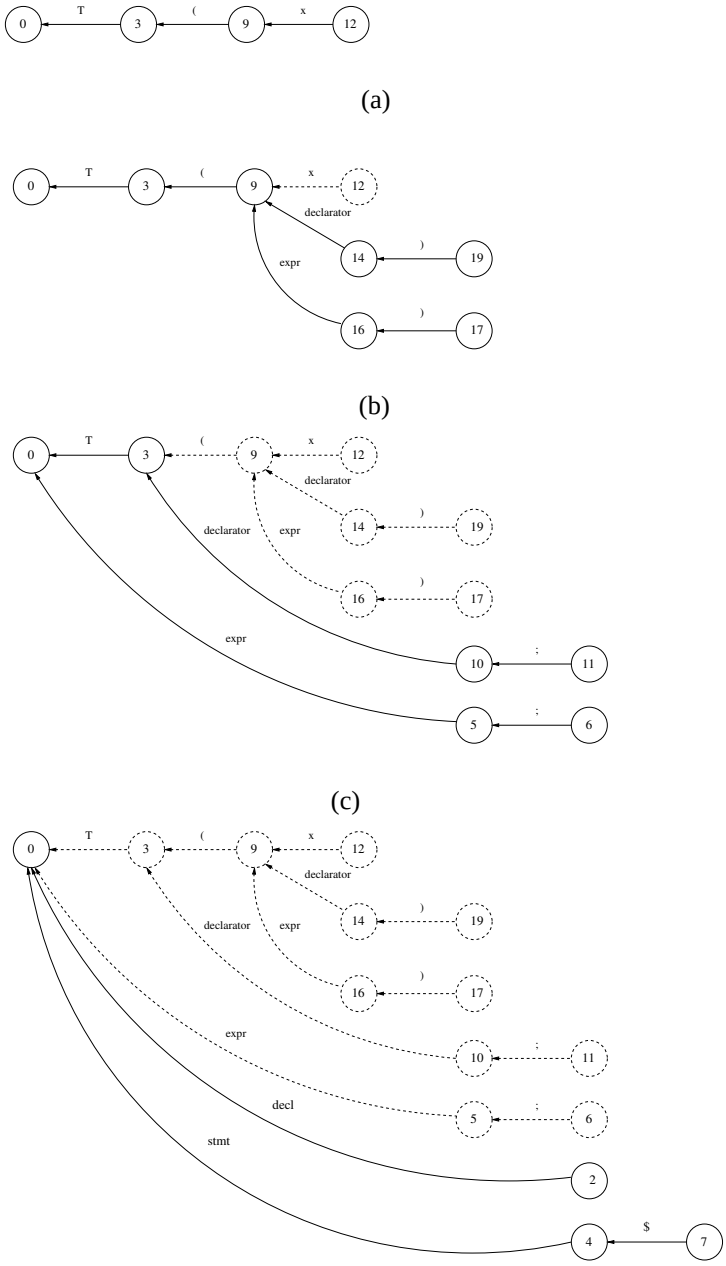


Figure 2.7: An illustration of a GSS corresponding to the grammar given in Figure 2.5. The GSS is shown at four successive stages of parsing the input sentence `T(x);`.

entry in the parse table for state 12. However the GLR algorithm now allows us to *fork* off different branches for each conflict that we encounter as we parse. Thus `x`, a terminal of type **ID** is reduced into both a *declarator* and an *expr* and two "parses" are created. The algorithm then calls for all possible reduces to be completed before the two parses are synchronised on the next input token. As both states, 14 and 16 are waiting on shifts, the algorithm proceeds to the configuration as outlined in Figure 2.7(b), to give stacks containing states 0, 3, 9, 14, 19 and 0, 3, 9, 16, 17.

The two parses continue in lockstep until the scenario illustrated in Figure 2.7(c) is reached. All of the input has been consumed and the current lookahead symbol is the end of file marker, $. The stack containing the states 0, 3, 10, 11 is reduced to give the stack containing 0, 2 with edge *decl* and the stack containing 0, 5, 6 is reduced to give the stack with states 0, 4 with an edge labelled *stmt*. The stack containing states 0, 4 is waiting to shift $ so the stack containing states 0, 2 is further reduced to give the stack with states 0, 4 with a *stmt* edge. However this edge already exists, so we can merge the stacks at node 4 to give the final accepting configuration of the GSS, which is displayed in Figure 2.7(d).

2.3.3 Existing GLR implementations

Due to the lack of mainstream penetration of generalised parsing algorithms there exist only a small number of publicly available GLR parser generators. The first of these is the **ASF+SDF** meta environment [vdBvDH$^+$01]. This is a large and well-established modular library that can be used to create GLR parsers via scannerless GLR parsing, where there is no separate lexical analysis tool. ASF+SDF also allows for the creation of tools to analyse and re-write any input via manipulation of the abstract syntax trees. The transformers project is an attempt to write a full C++ parser using ASF+SDF and attribute grammars [ADV04].

Another parser generator available is that of **Elkhound** [McP02]. Elkhound is a GLR parser generator written in C++ and has been used to create a parser for C++ known as Elsa. Elsa has been used to parse a large amount of "industrial-

strength" C++ code such as the Mozilla web-browser but as yet does not reject all invalid input.

From version 1.875 on, a GLR mode has been available for bison [Egg03]. This GLR mode does not use the GSS data structure and thus cannot represent a truly ambiguous parse. However there is a mechanism in place where the semantic actions that are traditionally written alongside the syntax definition of a grammar can be suspended during any phases of non-determinism during the parse. This allows for for a single parse of the input, with the semantic actions halted during ambiguities and only executed when the ambiguity is resolved at a merge point.

2.4 Testing

The work contained within this thesis seeks to develop a parser front-end for ISO C++ that utilises the GLR algorithm. As described earlier, the construction of a parser for C++ is a difficult task. Our work seeks to develop this front-end in the context of well established software-engineering techniques for parser design [PM01, MPW02]. One of the most important and fundamental of those techniques is software testing. We seek to apply some well known software testing techniques to aid in the construction of our grammar-based software for C++.

Testing constitutes an informal method of ensuring that a program behaves as per its specification. As Dijkstra observed, the process of testing a system only shows the presence of bugs, it can never show that a system is free of bugs [Dij70]. Hence it is essential that all areas of code within a system are covered by a test-suite. There are a number a different testing strategies available [Rop94, Bin99] and these strategies are outlined below.

2.4.1 Test-Suites

Constructing a test-suite is an integral part of the testing process. A test-suite is a collection of individual test-cases that are designed to exercise specific parts of

the system under test. A test-suite may be an *implementation*-based test-suite or a *specification*-based suite. An implementation based suite is one that is built to test features of a system. As new features are added over time or components of the system are changed, the test-suite is augmented with new test-cases designed to test the new features. An alternative approach is to design a test-suite directly from the specification of the system. For testing grammar-based software, this involves designing test-suites directly from the language specification.

2.4.2 Testing Techniques

Once a test-suite has been created there are a number of testing techniques utilising the test suite that are applied to the system throughout its development lifecycle. These techniques are summarised below.

Black-box Testing

Black-box testing is concerned with testing the system with respect solely to inputs and expected outputs. A specified output is expected for each test-case that is provided as input. The result of each test is a simple binary choice between pass or fail. This type of testing is referred to as black-box testing because there is no access to the source code of the system during the tests.

White-box Testing

White-box testing extends the notion of black-box testing. During each test, access to the source code is allowed. This allows bugs identified by the execution of the test to be fixed straight away. Furthermore, it allows for the measurement of the test-suite's code coverage.

Code coverage is a metric for defining how much code in a system has been executed during the running of a test-case. There are many measurements that can applied to code coverage. Three well-used examples are:

- **Statement Coverage**: This measures how many individual lines of code in the system have been covered during the test [JCMCJM63].
- **Branch Coverage**: This is a measure of how many conditional points in the system have been executed during testing [Rop94].
- **Path Coverage**: This checks that every route through a section of code has been covered [Nta88].

Unit Testing

Unit testing [SR92] forms part of the recent move towards the eXtreme Programming philosophy [Bec00] which is a cornerstone of test-driven development [Ast03]. This form of testing calls for test-cases designed to test individual modules, typically classes, to be created before any code is written. Unit-testing works by testing all of the methods in a class and each unit-test should be capable of being executed in isolation.

Regression Testing

Regression testing is a technique used to ensure that changes to a code base do not affect the correct operation of the system. If a change is made to a code base that is fully functional, then the changed system must be rigorously tested to ensure that the system still operates correctly. The entire test-suite for the system is applied to ensure that the changed code is exercised and the system still operates correctly.

Record and Replay

When testing a user-interface or some other system that relies upon human activity to drive the system events, it is often desirable to record the event-driven actions once so that they can be replayed repeatedly. This strategy is known as *record and playback* and is used heavily in testing systems such as GUIs [Mem02] and

network firewalls [HY05]. It operates by recording the correct operation of a test-case in a manner that can then be used automatically at some point in the future, without any human input.

The notion of a clear distinction between the front- and back-end in a GUI is standard. The actions from the user of the GUI drive the back-end logic. In theory the entire GUI could be replaced with an entirely new GUI without affecting any of the back-end logic. The distinction between the front- and back-ends of keystone is similar. The front-end is responsible for driving all of the back-end routines. In theory it becomes possible to replace the front-end without changing the back-end as only the *interfaces* between the front- and back-end need to be preserved.

2.4.3 Test-Suite Minimisation

When a test-suite becomes too large, then the cost of performing regression testing can become prohibitive and it may become desirable to reduce the test-suite size based upon some criterion where the test-cases overlap.

There are two central issues when performing test suite reduction. First, some criteria must be used to decide if a test case is redundant with respect to others in the suite. Typically, the criteria used are coverage based, although there are many different types of coverage criteria. Second, it is desirable that the reduced test suite have the same fault detection capability as the original.

Harrold *et al.* use coverage of definition-use pairs as their reduction criterion, and apply it to a set of seventeen C programs each containing less than 100 lines of source code [HGS93]. The test suites for these programs range in size from 4 to 80 test cases, and a reduction of up to 60% in the size of the test suite is reported. They do not report on the fault detection capability of the reduced suites.

Wong et *et al.* investigate the fault-detection effectiveness of reduced test suites for ten C programs, ranging in size from 90 to 842 executable lines of code [WHLM98]. They use block, decision and all-uses coverage as the reduction criteria, and with test suites for each application ranging in size from 156 to 997 test

cases they achieve reductions in size in excess of 94%. To measure the effectiveness of the reduced test suite, between 12 and 30 faults were manually injected into each program, with an average reduction in effectiveness ranging from 4.44% to 9.20%.

In contrast, a more recent study by Rothermel *et al.* finds a significant decrease in the fault-detection capability of reduced test suites [RHOH98]. This study uses seven C programs, ranging in size from 138 to 516 lines of code, with substantial test suites, ranging in size from 1052 to 5542 test cases. Using edge-coverage as their criterion for reduction, and starting with randomly selected subsets of the test suites, they achieve a reduction in test suite size of between 87% and 95%. However, after manually injecting between 7 and 41 faults into the programs, they report a significant decrease in the fault detection capability of the reduced suites, in many cases by up to 100%.

Jones *et al.* use modified condition/decision coverage as the reduction criterion, and apply it to two software systems written in C [JH03]. The first system, TCAS consists of 138 executable lines of C code, and its test suite is reduced in size from 1608 test cases to 10 test cases. The second system, Space, consists of 6,218 executable lines of C code and its test suite of 13,585 test cases is reduced to 11 test cases. To evaluate the fault detection capability of the reduced suites, 41 faulty versions of TCAS and 35 faulty versions of Space were employed, with the average loss in fault detection being 44.4% and 10.2% respectively. The figures for coverage and fault detection are average figures, since 1,000 randomly sized selections of test cases were used as the starting point for the reduction process.

Heimdahl *et al.* apply test suite reduction to specification-based tests for a flight system consisting of 2564 lines of code in RSML^{-e} [HG04]. They generate and then reduce test suites using six different coverage criteria. With the original test suite sizes ranging in size from 115 to 537 test cases, they report an average reduction in test suite size of 80%. Using a random fault seeder they create 100 faulty versions of the program, and report a decrease of between 7% and 16% in fault detection capability for the reduced suites, which they deem unacceptable

for their domain of interest.

2.4.4 Mutation Testing

Mutation testing is a testing strategy that attempts to introduce faults into a software system [DLS78]. These faults can be introduced by hand or by automatically generating mutants directly from a line of code. The automated generation of a mutant is achieved by applying what is known as a mutation operator to the code. The result of this is a faulty version of the software.

A mutant is said to be "caught" if there is a test-case within a test-suite that uncovers the fault, i.e. the program behaves in an unexpected manner for the test-case. Failure to uncover the mutant is a sign of a deficiency in the test-suite and results in the addition of new test-cases to the suite. The mutant can be caught by either by one of two types of mutation testing:

Strong Mutation Testing If the external output of the program differs from that expected output, then the mutant can be considered caught.

Weak Mutation Testing If the state of the program is different than expected immediately after the execution of the mutated statement, then the mutant is labelled as caught.

There are numerous ways of mutating a system but a study by Offutt *et al.* analyzed 22 different types of mutation, and identified a core set of five mutation types that were almost as effective as the entire set [OLR+96]. The mutation types, listed in Table 2.1, facilitate the automated mutation of a statement within a program. This is in contrast to more advanced mutation techniques, such as a mutation based upon class properties, which need full semantic information in order to apply the mutation.

Mutation testing allows the fault detection capability of a given test-suite to be analysed. Given a number of test-suites, it is possible to compare them side-by-side by counting how many mutants they catch when the mutant operators are applied to a system.

Operator	Description
ABS	Absolute value insertion Replace an expression by 0, a positive value and a negative value
AOR	Arithmetic operator replacement Replace one of the binary arithmetic operators by each of the others
LCR	Logical connector replacement Replace one of the binary logical operators by each of the others
ROR	Relational operator replacement Replace one of the binary relational operators by each of the others
UOI	Unary operator insertion Insert a unary operator before the expression

Table 2.1: The five kinds of mutation operator applied to the SUTs. *All mutation types apply to expressions, and, when applied recursively to a single expression, can give rise to many mutated versions.*

2.4.5 Defining a Test-order

When two or more classes that have an interaction with each other are implemented, it is desirable to test their interaction through integration testing. Where a class interacts with another, as yet, unimplemented class, then class *stubs* need to be used to simulate the functionality of the unimplemented class [Mar03]. Stubs usually implement a subset of the class functionality but may have to replicate the complete class functionality in extreme cases. Thus, where possible, the use of stubs should be kept to a minimum due to the costs involved in their construction.

There are many examples of work in the area of defining an order for testing object-oriented systems that make use of a type of graph known as an *Object Relation Diagram* (ORD) [KGH+95, TD97, BLW01, MRR02]. The ORD can be loosely compared to a UML class diagram as it features classes as nodes and the relationship between classes as edges. The three edges typically represented

within an ORD are *Associations*, *Composition* and *Inheritance*. Compositions represent classes that are composed of other classes. The life-times of the enclosed classes are bound to that of the containing class. *Associations* themselves can be further decomposed into simple aggregations, dependencies and associations.

The ORD may be constructed manually or through the use of an automated process. The guidelines for transforming C++ source code into an ORD were first described without any explicit reference to automated tool support [KGH+93]. However it is possible to use a reverse engineering tool to construct an ORD automatically [MCL03]. Once the ORD has been constructed, it is possible to assign costs to the edges and nodes, and use these to define a testing order, so that the class with the smallest number of dependencies can be tested first. To this end, it is usual to define an ORD *cost model* that determines the order in which classes can be tested so as to minimise the need for class stubs. We observe that the same problem exists in porting, specifically in regard to the order in which to port the classes within the system. By porting the classes according to a costed ORD, we can dramatically reduce the amount of class stubs needed during the unit testing phase.

Malloy *et al.* have enhanced the basic ORD cost assignment by developing a *parameterised* cost model [MCL03]. Such a cost model allows the ORD to be configured and changed easily, so that it can be fine tuned for a specific application. With the parameterised cost model, each type of edge within the ORD is given a specific weight and all the edges between two classes are merged and their weights are summed. The ORD is divided into strongly connected components (SCC) and the edge with the smallest weight is removed. This process is repeated until all the SCCs contain only 1 node. Finally the ORD is reverse topologically sorted to give an inter-class test order.

Milanova *et al.* have extended the idea of the ORD further by developing the ExtORD which is an ORD created at the precision of the statement level [MRR02]. Multiple edges of each kind of dependence between the class nodes are utilised to give this precision. The level of precision in this strategy is unnec-

essary for our porting strategy but this approach is useful in identifying coverage of statements that trigger inter-class dependencies.

2.5 Testing Grammar-based Software

The use of *rule coverage* as a criterion for testing grammars was introduced by Purdom [Pur72]. A test case is said to cover a grammar rule if that rule is used at least once in deriving that test case. Since a non-terminal may have many alternative rules, rule coverage is similar to decision coverage at the code level in a traditional software testing context [Rop94]. Purdom described an algorithm that systematically uses the grammar rules to generate valid sentences, so that each grammar rule is used at least once. Thus, the output of Purdom's algorithm is a test suite of grammatically correct programs that achieves 100% rule coverage. Purdom applied the technique to several small grammars, as well as a grammar for ALGOL, and it has since been applied to other languages including PL/1 and Pascal [BS82, CCRV+80].

However, there are at least three main difficulties in applying this technique to grammars for modern programming languages [MP01]. First, many grammars over-specify the language, in that they admit constructs that are not syntactically valid. This approach can often make the grammar easier to understand, but means that extra constraints must be applied to the generation algorithm to weed out spurious programs. Second, context-sensitive information, such as the scope and type of variables, is not represented in the grammar, and thus has to be added to the programs using some other technique. While it is possible to define these extra constraints using multi-level grammars [CCRV+80] or attribute grammars [HL00], it would be extremely difficult to apply this in full to a programming language like C++. Finally, if the grammar contains ambiguities, such as the C++ grammar, there is no guarantee that the rules used in generating a sentence will be the same as those used in parsing that sentence.

2.5.1 Existing test suites for ISO C++

The popular, open-source GNU compiler collection *gcc* includes a large test suite for the various languages accepted by the compiler. The C++-specific part of the test suite distributed with *gcc* version 4.0.0 contains 5067 C++ programs. This is an *implementation-based* test suite, in that it was assembled to test various compiler features, and augmented as bugs were discovered or new features were added. Indeed, the four most recent versions of *gcc*, 3.2, 3.3, 3.4.0 and 4.0.0, released roughly at annual intervals, show an increase in the size of the C++ test suite of 8%, 10%, 18% and 12% respectively on the previous version.

An alternative approach to gathering a test suite is to consult the language specification, and to attempt to create test cases that cover all aspects of the language. This *specification-based* approach is commonly used to test for compliance with the standard, to ensure that a compiler implements all features of the language. Examples for C++ include the CppETS suite developed as a benchmark suite for reverse engineering tools [SHE02], the *DDJ* suite, developed to test compliance of different compilers to the ISO standard [GMP03b, MLDP02] and the commercial test-suites from Plum-Hall and Perennial [PHts, PtsfIC++].

2.6 Synopsis

We have outlined the testing methodologies that we used throughout this research to deliver a new parser front-end that utilises the GLR algorithm. The next chapter describes the work used to create standardised test-suite for front-ends that receive ISO C++ programs as input by applying test-suite minimisation techniques to existing test-suites. The fourth chapter expands on the porting strategy for porting the keystone back-end from C++ to Java by exploiting object relation diagrams. The fifth chapter describes the construction and testing of a GLR parser generator from which the new front-end is generated. Finally, the sixth chapter describes the testing of the generated front-end and the integration of the new front-end and ported back-end.

Chapter 3

A study of test-suite reduction techniques for grammar-based software

In this chapter we conduct a study into the development of a minimised test-suite for grammar-based applications that receive ISO C++ as input. We choose to reduce two existing test-suites based upon grammar-rule coverage. We then evaluate the minimised test-suites based on two criteria: that of code coverage and fault detection.

3.1 Introduction

A grammar may occur either explicitly or implicitly in grammar-based software. An *explicit* occurrence typically takes the form of input to a parser-generation tool such as yacc and, in this case, a direct correlation can often be achieved with the rules of the programming language grammar. An *implicit* occurrence may be in the form of a hand-written parser, where it is not easy to distinguish parsing code from the remainder of the tool. Further, many tools that require only partial information from the input make use of a *fuzzy* parser, where irrelevant

parts of the input are ignored by the parsing routines [Kop97]. However, whether the grammar is explicitly defined or not, we expect to find a commonality that pervades all grammar-based systems: the acceptable input can be defined by a context-free grammar.

Since a grammar constitutes a formal specification of the input to grammar-based software, it is possible to utilise formal approaches to verifying such software [RL01]. However, in the case of implicit grammar occurrences, less formal techniques such as testing become important. Even in the case of software based on explicit grammars, the scale and complexity of modern programming languages can cause considerable difficulties for theoretical approaches, such as those based on attribute grammars. Thus, in such situations, issues associated with software testing, such as coverage, fault detection capability and test suite size come to the fore.

Test suites typically evolve in tandem with the software they test: as new features are added to the software, and new bugs are uncovered and fixed, relevant test cases are added to the suite. Since large test suites can impose a considerable overhead on regression testing, it is desirable to reduce the test suite size if overlaps or redundancies exist. The reduction is typically based on a *code* coverage criterion within the system under test [HGS93]. For grammar-based software however, we choose to use the *rule* coverage of the inputs to the system as the reduction criterion.

In this chapter we describe an approach to the testing of grammar-based software, using the ISO C++ grammar as a case study.

3.2 Goals of this Chapter

Our study involves taking two existing test suites for ISO C++ and analysing test suite reduction techniques based solely on grammar coverage. In the remainder of this chapter we refer to these test suites as:

- T_{gcc}, the C++ programs from the test suite distributed as part of *gcc* version 4.0.0

- T_{ddj}, a test suite derived from the ISO standard used to test conformance to the standard, described in [MLDP02]

We examine whether reduced versions of these test suites will be as effective as their larger counterparts; specifically, we investigate the following hypotheses:

Hypothesis 1: Reducing test suites based on rule coverage will not adversely affect *code coverage* when used to test grammar-based software.

Hypothesis 2: Reducing test suites based on rule coverage will not adversely affect the *fault detection capability* when used to test grammar-based software.

Hypothesis 2a: Reduced test suites that maintain rule coverage catch more faults than randomly-generated test suites of the same size. Showing that randomly-generated suites catch less faults while covering less rules than the reduced suites demonstrates that rule coverage is an essential determinant of fault detection capability.

3.3 A reduced test suite for ISO C++

In this section we describe the construction of two reduced test suites for ISO C++. We discuss the implementation of rule coverage measurement using *gcc*, and we present the results of applying test suite reduction to the T_{gcc} and T_{ddj} test suites.

There are two main phases in reducing a test suite based on rule coverage. First a system capable of determining which grammar rules from the grammar were used during a parse must be constructed. Second, a test suite reduction algorithm must be implemented and applied to the test suite.

Test suite	Test Cases	Positive Test-Cases	LOC	Rule Coverage
T_{gcc}	5067	4195	95946	95.3 %
T_{ddj}	468	449	6003	89.6 %

Table 3.1: Results of profiling the two test suites. *For each of the original test suites we show the size in terms of the number of test cases and lines of executable code, along with the percentage grammar rule coverage achieved by each. The positive test-cases are the number of test-cases that are positive with respect to keystone.*

3.3.1 Measuring rule coverage

Since our reduction strategy is based on grammar rule coverage, it is essential to be able to determine which rules are used by each test case. Given that the C++ grammar is heavily context-sensitive, it is essential to use a fully-functional parser and front-end in order to correctly determine the rules that are used. Previous work had developed an instrumented version of GNU bison, and had used this with the parser in the version 3.0 of the *gcc* C++ compiler to produce an XML trace of the grammar rules used [HPM03, PM02]. However, while harnessing this explicit grammar facilitated profiling, the grammar in question had undergone considerable evolution, and it proved difficult to reconcile its rules directly with the ISO standard.

Fortunately, the C++ parser in *gcc* has been completely re-written as a hand-coded recursive descent parser, which corresponds closely to the grammar in the ISO standard. To track rule coverage, the parsing code in *gcc* version 4.0.0 was identified and profiling code was added to generate a log of grammar rules that were used as each input program was processed. Each test case in our two test suites was then profiled in this way using our modified *gcc*.

The results of profiling the two test suites is given in Table 3.1. As can be seen from column 5 of this table, neither test suite achieves 100% rule coverage, though both come close. The second column in Table 3.1 lists the number of *positive* test-

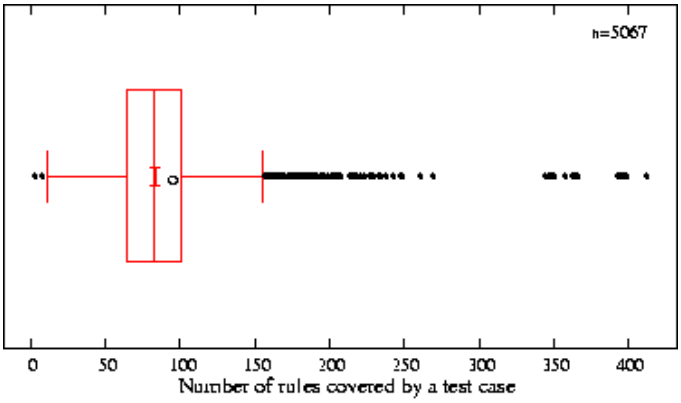
cases in each test-suite. It is important that only positive test-cases be considered as our later experiments involve the use of mutation testing, and it is important that all negative test-cases are removed lest they give a false impression of the mutation detection ability of the test-suites. Based on the rule-coverage analysis, both suites were augmented with extra test cases in order to achieve 100% rule coverage. These test cases were generated by slightly modifying Purdom's sentence generation algorithm so that it produced sentences guaranteeing coverage of just a single rule at a time. These generated test cases were simple enough so that they could then be modified by hand to ensure that they were correct C++ programs. In the remainder of this chapter, we use T^+_{gcc} and T^+_{ddj} to denote the set of positive test cases augmented to bring them to 100% rule coverage.

Since we are not aware of any existing work analysing rule coverage for test suites, we present a summary of the rule coverage data for the T^+_{gcc} and T^+_{ddj} test suites in Figure 3.1. Figure 3.1 is a box plot showing the distribution of rule coverage among the 4195 test cases in T^+_{gcc} and the 449 test cases in T^+_{ddj}. As can be seen from this figure, the T^+_{gcc} suite has a slightly higher mean rule coverage, but also a larger spread of coverage. In fact, the mean coverage for T^+_{gcc} is 92 rules, against 70 rules for T^+_{ddj}, but the standard deviation is 55.6 for T^+_{gcc}, against 27.4 for T^+_{ddj}. The 25th and 75th quartile are 65 and 101 for T^+_{gcc}, and 53 and 86 for T^+_{ddj}. The narrower coverage range for T^+_{ddj} reflects on the specific nature of its test-cases, which are designed to test very specific features of the C++ language.

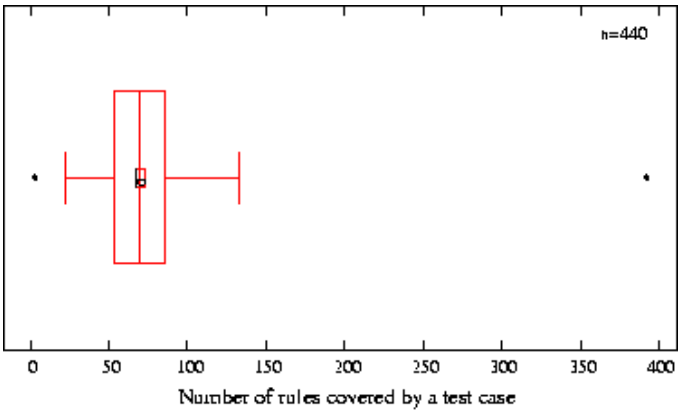
3.3.2 Test suite reduction

The test suite reduction algorithm follows that of Jones *et al.* [JH03], and operates as follows:

1. Taking each positive test case in turn we compile a vector of length 479, with one entry corresponding to each C++ grammar rule, holding a 1 or 0, depending on whether or not that rule was used as the test case was parsed.
2. The vectors for all the test cases are placed together in a 2D array whose



(a) T^{+}_{gcc} test suite



(b) T^{+}_{ddj} test suite

Figure 3.1: Distribution of rule coverage among the T^{+}_{gcc} and T^{+}_{ddj} test suites. *For each graph, the horizontal axis represents a count of the number of rules, and the box plot shows the distribution of rules covered.*

---

rows are indexed by the test cases and whose columns are indexed by grammar rule number.

3. If any *column* sums to one, then only one test-case covers the corresponding rule, and these test-cases are deemed essential and added to the reduced test suite. Whenever a test-case is added to the reduced suite, all of the vector entries corresponding to rules that are covered by this test-case are set to

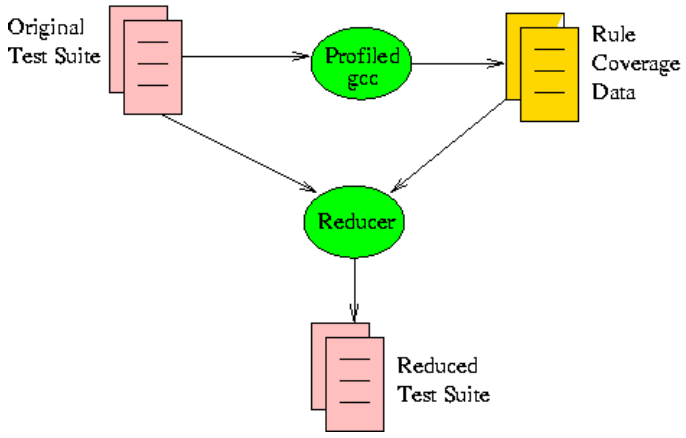


Figure 3.2: Overview of the test-suite reduction process. *An instrumented version of* gcc *is used to gather rule coverage data for each program in the test suite, and this data then used to reduce the test suite.*

Test Suite	Test Cases	Reduction	LOC	Reduction
R_{gcc}	40	99.05%	1210	98.74%
R_{ddj}	44	90.20%	638	89.38%

Table 3.2: Percentage reduction achieved by the test suite reduction algorithm. *For each of the reduced test suites we show its size, in terms the number of test cases and total lines of executable code (LOC), and the percentage reduction compared to the corresponding original test suites.*

---

zero.

4. The *rows* are then summed to identify the test-case that contributes the most to rule coverage. This is added to the reduced set, the vector entries corresponding to the rules it covers are set to zero, and the process is repeated.

The test-suite reduction process is summarised in Figure 3.2.

It is worth noting that once all the essential test-cases have been removed, the problem of choosing the minimum test-set that covers the remaining rules is equivalent to the minimum cardinality hitting set, which is an intractable problem [GJ79]. Hence the process will always be heuristic and in our case we choose to

System	Version	Source Files	LOC (≈)
Doc++	3.4.10	17	5,561
Keystone	0.2.3	52	6,879
Puma	1.0pre3	136	14,438

Table 3.3: Systems Under Test. *For each of the three grammar-based applications used in our case study we show the version number used, the number of C++ source files, and the number of executable lines of code (LOC).*

---

always use the test-case that contributes the most coverage even though this will not guarantee the smallest test suite.

The test suite reduction algorithm was applied to both of the existing test suites, generating two new suites which we refer to as R_{gcc} and R_{ddj}. By design, each of these suites achieves 100% rule coverage. The results of applying this algorithm are summarised in Table 3.2. For R_{gcc} there are 4155 *less* test-cases, a reduction of 99%. For R_{ddj}, there are 404 less test-cases, a reduction of 90%. Both of these suites represent a dramatic reduction in size from the originals, and are comparable to the size of the test suites generated for C++ using Purdom's algorithm. However, all the test cases in these reduced suites are semantically correct C++ programs, unlike the test cases generated using Purdom's algorithm.

3.4 Empirical Study: Code Coverage

In this section we investigate our first hypothesis, that reduction under rule coverage does not adversely affect code coverage. In order to do this we use three examples of grammar-based software that accept C++ programs as input. We refer to these as the systems under test (SUT), and they contain a mixture of implicit and explicit grammars.

Doc++ is an automatic documentation generator for C++ files [Aco00]. There is no explicit grammar file and it must rely on code landmarks within an input

C++ program to complete a fuzzy parse.

Keystone is a complete front-end to aid in the static analysis of ISO C++ programs [GMP03a]. It has an explicit grammar, modelled on the grammar in the ISO standard, which is used as input for the btyacc parser generator.

Puma is a library for parsing C++ that is used as the front-end for AspectC, an Aspect Oriented extension for C++ [SGSP02]. The parser code is hand written and thus has no explicit grammar.

Table 3.3 gives the version numbers and some basic size measures of these programs. In this and subsequent sections, all measurements in terms of lines of code (LOC) refer to *executable* lines of code, as reported by version 4.0.0 of the gcov utility.

3.4.1 Calculating code coverage

The first experiment was conducted in a highly structured manner and where possible automated scripts were used. The steps involved are outlined below.

1. Each of the three SUTs was built with compiler flags set to profile using gcov, a profiling tool that is part of the *gcc*.

2. Each of the three SUTs were run using the full and reduced test suites as input. The output from each SUT for each test case was stored for use later in the mutation testing phase.

3. The coverage figures for each test suite were measured twice. For the first run, all test cases were passed through to give cumulative coverage figures for each of the test suites. For the second run, all individual test cases had their code coverage figures measured to determine if there was a correlation between rule coverage and code coverage.

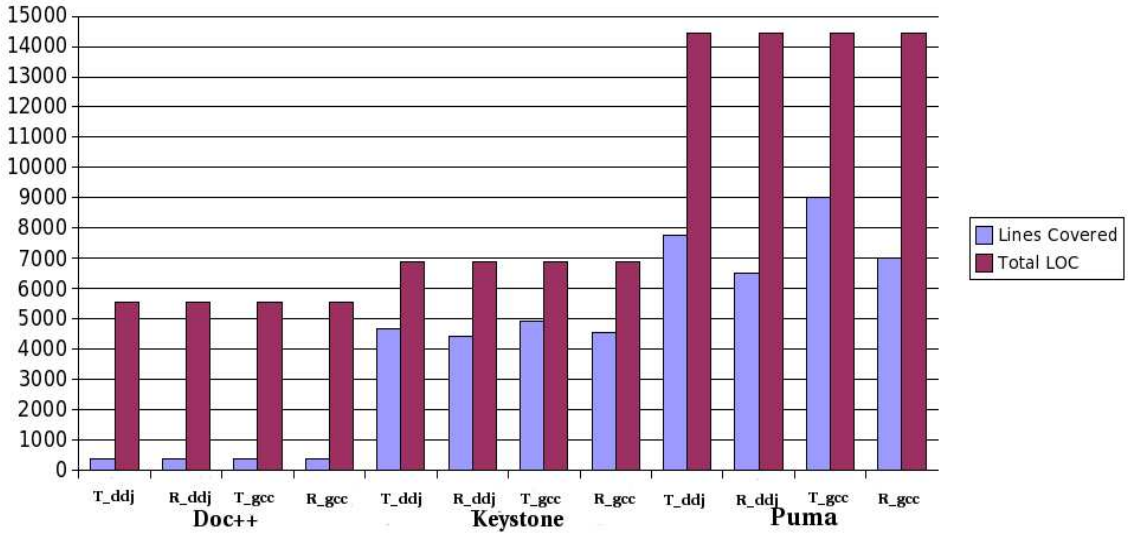


Figure 3.3: Code coverage results for each of the SUTs. *For each SUT we show the code covered by the large and reduced test-suite alongside the total amount of code in the SUT.*

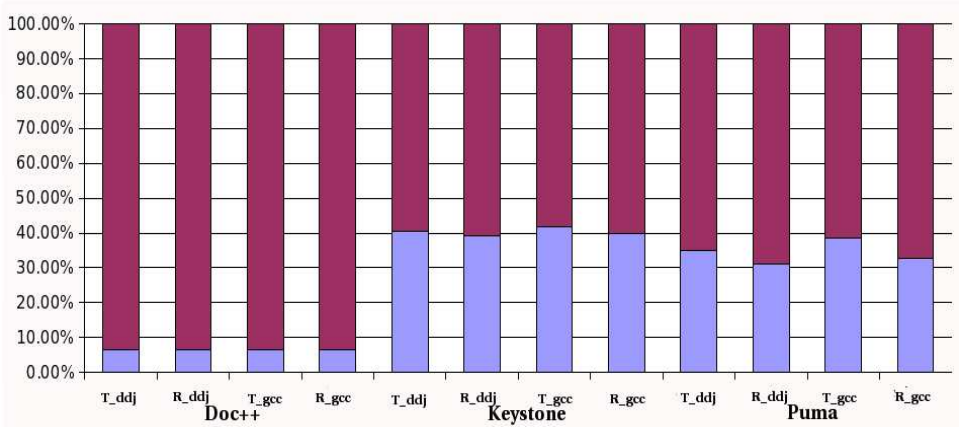
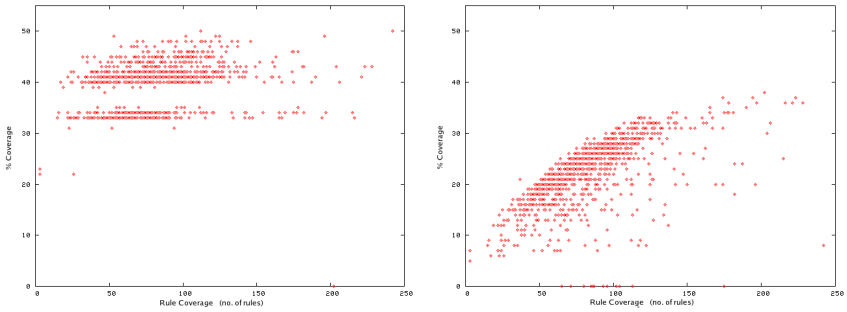
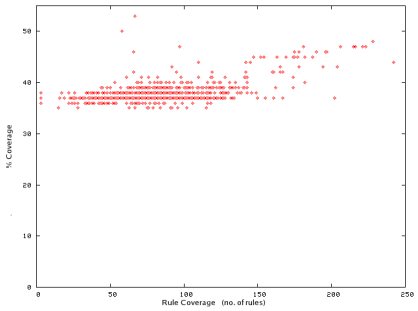


Figure 3.4: Code coverage results for each of the SUTs. *For each SUT we show the percentage of code covered for each test-suite.*



(a) Doc++ (b) Keystone



(c) Puma

Figure 3.5: Rule coverage versus percentage line coverage for the three SUTs. *In each graph the horizontal axis measures the number of grammar rules covered, and the vertical axis represents percentage line coverage. Each point on the graph represents a single test case, with outliers covering more than 250 rules removed for clarity.*

3.4.2 Results

Figure 3.3 displays the summarised code coverage figures for each of the test suites. Each SUT is represented by eight bars, with red bars representing the total number of lines of code in the SUT and the blue bars representing the number of lines of code covered for the T^{+}_{ddj}, R_{ddj}, T^{+}_{gcc} and R_{gcc} suites. Figure 3.4 shows

the coverage as a percentage of the overall coverage of the system. The first point to note in Figure 3.4 is the relatively low overall code coverage, even for the two larger suites, T^{+}_{ddj} and T^{+}_{gcc}. This is due to neither test suite being developed specifically for the SUT in question, and thus some of the back-end functions of each SUT remain untested.

However, the main result shown in Figure 3.4 is the relatively low decrease in the degree of code coverage between the total and reduced versions of the test suite, despite the considerable reduction in test suite size. The largest reduction in coverage for any SUT is that shown for keystone, where moving from T^{+}_{gcc} to R_{gcc} reduces the code coverage from 69.6% to 60.4%.

Figure 3.5 contains a scatter plot for each SUT, showing the relationship between rule coverage and code coverage. Here, each point on the scatter plot represents a single test case from the T^{+}_{gcc} test suite. As might be expected from the visual data in Figure 3.5, no strong linear correlation exists between rule coverage and code coverage. For *doc++* and *puma*, the graphs show that code coverage is largely invariant in the range 30%-50% for many of the test cases. keystone exhibits a very weak linear relationship, but again code coverage for individual test cases lie predominantly in the range 15%-35% These results demonstrate that the results shown in Figure 3.4 are not simply due to rule coverage acting as a surrogate measure for code coverage.

3.5 Empirical Study: Fault Detection

In this section we investigate the usefulness of the reduced test suites in terms of detecting faults within a grammar-based system. Fault detection is the central focus of the testing process, and provides an external measure of the effectiveness of that process. Our second hypothesis under investigation aims to determine whether the reduced test suites can detect as many faults as their larger counterparts.

3.5.1 Mutation Testing

To investigate the fault detection capability of the reduced test suites, we seed the three SUTs from Section 3.4 with faults, and compare the effectiveness of the full and reduced test suites in detecting these faults. This approach is broadly similar to mutation testing, except that our goal here is to compare test suites, rather than to ensure full fault detection capability. In mutation testing, the source code of the SUT is mutated to introduce an error, and a test suite is evaluated on its ability to detect this error. If the test suite produces different output or behaviour for the mutated version of the SUT then it has detected the error, and the mutant is said to be *killed*. Failure to kill all mutants indicates a deficiency in the test suite and, typically, new test cases are added to address this.

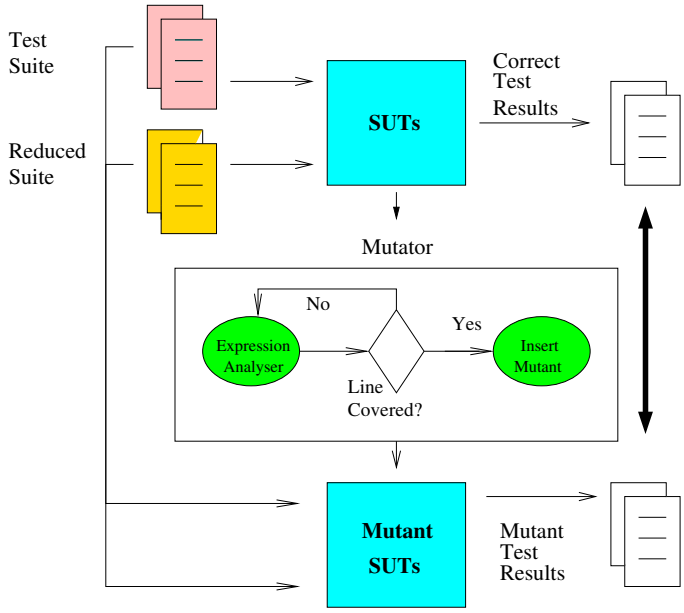


Figure 3.6: Overview of the fault-insertion process. *The mutator parses the source code of the SUT, identifies expressions, and outputs the relevant mutation operations. These are then applied one at a time, and the results are compared with the original test results.*

We applied the five kinds of mutations listed in Section 2.4.4 to our three SUTs

automatically, using the following process:

1. Each SUT is run with each test case in the test suite as input, and the output for that test case is recorded.

2. The code coverage of the R_{gcc} suite was recorded using the gcov utility for each SUT.

3. The C++ code for each SUT is analysed, and relevant expressions in the code are identified automatically as candidates for mutation.

4. The mutation operators are applied to each expression in turn if and only if gcov reports that the R_{gcc} suite has covered the current line, and the mutated expression, along with the position where it occurs in the program is output.

5. A simple script then applies each mutation to the relevant source file in the SUT, which is then re-built. The mutant SUT is first tested using the reduced test suite and, if the mutant is not killed, it is tested using the full version of the test suite.

The mutation generator consists of a scanner and fuzzy parser for C++ and is written in just under 1,000 lines of Python. For simplicity, the parser does not use a symbol table, and thus over-recognises expressions in the code. While this does not impact the findings of the experiment, it does result in a high number of mutant programs being invalid, since they fail to compile. The effect of over-recognition of expressions resulted in between 60% and 90% of the generated mutations being invalid. Since the process of discarding an invalid mutant is simpler that writing an accurate parser for C++, this was deemed to be an acceptable level of invalidity.

Applying a single mutant involves checking to see if the current line in the current source file is covered and then changing that single source file by inserting the mutant, rebuilding the SUT, and, if the SUT compiles, running the SUT with each program from the test suite as input. If the output for any one of the test cases is different from the non-mutated version of the SUT, then that mutant is killed.

	Mutants						
SUT	Total Applied	Killed *gcc*	Missed *gcc*	Reduction *gcc*	Killed *drdobbs*	Missed *drdobbs*	Reduction *drdobbs*
Doc++	893	509	0	0%	509	12	2.30%
Keystone	5434	3376	0	0%	3276	268	7.56%
Puma	15308	4575	0	0%	4333	590	11.98%

Table 3.4: Results for fault detection within the SUTs. *For each SUT we show the total number of mutant programs generated, the number of mutants killed and missed by the reduced suite, and the percentage reduction in fault-detection effectiveness.*

3.5.2 Results

Table 3.4 summarises the results of the mutation process for both test-suites. The first data column shows the total number of programs containing mutants generated by the mutation process for lines of code covered by R_{gcc}. The second data column shows the number of mutants killed by the reduced test suite, and the third data column shows the number of mutants missed by the reduced test suite, but killed by the total test suite. The final column of Table 3.4 gives the reduction in fault detection effectiveness, expressed as a percentage of the number of faults detected by the whole suite; that is:

$$Reduction = \frac{Missed}{(Killed + Missed)} * \frac{100}{1}$$

As can be seen from Table 3.4, the minimum test-suites appear to be just as effective as their larger counterparts. The R_{gcc} suite does not miss any seeded mutants while the R_{ddj} suite shows a reduction in effectiveness over a range from 2.5% to approx 12%. The reduction in the effectiveness of the R_{ddj} suite is due to the fact that the coverage figures for this experiment were only measured with respect to the coverage of the R_{gcc} suite. As can be seen from Figure 3.4, the R_{gcc} suite covers marginally more code than the R_{ddj} suite. Hence when the mutated programs are being tested, the R_{ddj} suite does not cover every mutant applied and

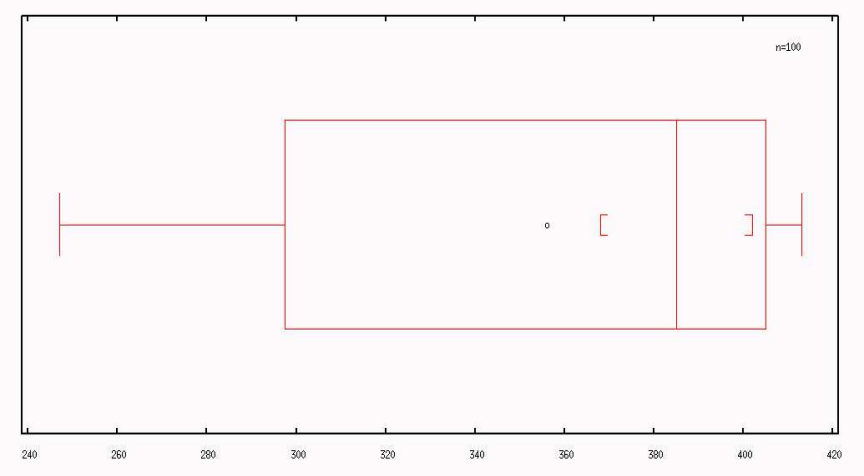


Figure 3.7: A box plot showing the distribution of the rule coverage of randomly created test-suites. *The number of rules covered range from 247 to 413 with 356 being the mean. All of the random suites cover less rules than the R_{gcc} and R_{ddj} suites.*

SUT	Total # of Mutants Applied	Average # of Mutants Killed	Max # of Mutants Killed
Doc++	55681	499.19	518
Keystone	9194	3223.75	5077
Puma	44474	8193.85	9739

Table 3.5: The total number of mutants applied and caught during the investigation of Hypothesis *2a.*

thus the mutated SUT is tested with the T^{+}_{ddj} suite.

The results in Table 3.4 indicate that the reduced test-suite is just as effective as the larger suite at catching mutants applied with respect to the coverage of the minimum suite. These results seem to indicate that suites reduced by rule coverage can maintain the fault detection capabilities of their larger counterparts, hence we might provisionally accept that Hypothesis 2 is true.

3.5.3 Comparison with randomly generated reduced suites

To validate how effective rule coverage is as a reduction criterion, a further experiment, summarised by Hypothesis 2a, was conducted. 100 new test-suites were randomly created by sampling with replacement from the T^{+}_{gcc} suite. Each new test-suite contained 40 test-cases chosen randomly without any regard to their rule coverage. The number chosen was exactly the same as the number of test-cases in the R_{gcc} suite and test-cases were available for selection repeatedly due to the sampling with replacement. The box plot in Figure 3.7 shows the distribution of the rules throughout the randomly created suites. The number of rules covered ranges from 247 to 413 with more than 50% of the random suites having a coverage figure of 380 rules. All of the randomly created suites covered less rules than the R_{gcc} and the R_{ddj} suites.

The main mutation experiment was repeated with a slight elaboration for each of the 100 random suites. The new experiment applied all possible mutants for each SUT and only executed one of the random suites if that random suite covered the line with the inserted mutant. As each new random suite covers less grammar rules than R_{gcc} and R_{ddj}, we expect all of the random suites to uncover less mutants than R_{gcc} and R_{ddj}. If the random suites find more inserted faults than the minimum suites then we can reject rule coverage as an effective criterion is reducing test-suites of grammar-based software.

Table 3.5 gives a summary of the total number of mutants applied and caught during the investigation of *Hypothesis 2a*. As the third column in Table 3.5 shows, the *average* number of mutants caught is roughly similar to the *total* number of mutants caught by R_{gcc} and R_{ddj} for *doc++* and keystone and *twice* that of the mutants caught for *puma*. The fourth column in Table 3.5 shows the maximum amount of mutants caught by a single random test-suite during the experiment.

Our study in section 3.5.2 showed that the minimised test suites R_{gcc} and R_{ddj} were reasonably good at maintaining fault detection. However, the results displayed in Table 3.5 show that similarly-sized test-suites that cover less grammar-rules are equally good, if not better than the minimised suites at killing mutants.

Hence we can conclude that the level of fault detection is a factor of the number of statements covered rather than the level of rule coverage. As this is a direct contradiction of hypothesis 2a, which expects test-suites with lower rule coverage to kill less mutants, we can then conclude that rule coverage *alone* is an ineffective criterion for reducing test-suites for grammar-based software. This allows us then, to also reject hypothesis 2.

3.6 Threats to validity

In this section we discuss the threats to the internal and external validity of this study.

Threats to internal validity

The two test suites used, T_{gcc} and T_{ddj}, may not be representative of test suites for C++ programs. While T_{gcc} is certainly among the most comprehensive implementation-based test suites available, it should be noted that commercial test suites for ISO compliance, such as those produced by Perennial [PtsfIC++] or Plum Hall [PHts] , can be an order of magnitude larger than either T_{gcc} or T_{ddj}.

The reduction strategy was based solely on rule coverage, and it is possible that a combination of rule coverage with other kinds of coverage might yield better results. For example, one stronger form of rule coverage is *context-dependent* rule coverage [RL01], although our analysis of context-dependent rule coverage [HP05b] suggests that there is little practical benefit to be gained from context-dependent coverage, at least in the context of the ISO C++ grammar.

The mutations applied to each SUT are only a selection of those possible. For example, mutation operators can be defined that test object-oriented features (such as those defined by Ma *et al.* for Java [MOK05]), that could yield different results.

Threats to external validity

Threats to external validity centre on the choice of grammar used and the choice of SUTs. ISO C++ was chosen for our study as it represents a particularly challenging grammar for analysis purposes. It is thus possible that grammars for less complex languages may yield better results, although in the absence of a formal quantification of the link between grammars and the back-end code this is difficult to judge.

The three SUTs used in our experiments in Sections 3.4 and 3.5 were chosen as examples of medium-sized applications that took C++ code as input. As discussed in the introduction, grammar-based software includes many other kinds of application, and it would be useful to add examples of these to our study. Using larger applications as SUTs might also be useful, but it seems unlikely that they would yield better back-end coverage results than the ones presented here.

3.7 Synopsis

In this chapter the feasibility of using rule coverage as a criterion in the reduction of test suites for grammar-based software has been tested. We have taken two existing test suites for ISO C++, and applied a reduction strategy based on rule coverage. To estimate the effect of this reduction we have studied three grammar-based applications, and investigated the code coverage and fault detection capabilities of the reduced test suites.

The main findings of the work are:

1. Test suite reduction based on rule coverage provides a significant reduction in the number of test-cases, and thus in the testing overhead. The size of the reduction is comparable to strategies that use other coverage criteria, and produces a test suite that is comparable in size to that generated by Purdom's algorithm, with the added advantage of semantic correctness.

2. We have demonstrated for three grammar-based applications that, while there is no formal correlation between rule coverage and code coverage, the reduced test suites do not significantly reduce the level of code coverage. While this was to be expected for the code purely relating to parsing, it is notable that it also holds for other parts of the applications that were tested.

3. However, our mutation testing results indicate that the reduced test suite does not adequately preserve fault detection capability. In this, the results are consistent with those of Rothermel *et al.* [RHOH98], in that there is a notable decrease in the fault detection capability of the smaller test-suite. Indeed we have also shown in rejecting Hypothesis 2a, that rule coverage is a subset of statement coverage with respect to reduction criteria.

Despite the encouraging results in relation to the preservation of code coverage for the reduced suites, the failure in the rate of fault detection must be considered a significantly negative finding.

We identify the novel contributions of this chapter as:

- The use of standardised test suited for grammar-based applications. This differs from standard testing techniques where, typically, test suites are designed anew for each individual application.

- A rule coverage analysis of two significant test suites for ISO C++, based on results from profiling the parser from the *gcc* C++ compiler.

- The implementation and analysis of automated test suite reduction using rule coverage as the criterion.

- An analysis of the reduced test suite in terms of code coverage and fault detection, and its application to three instances of real-world grammar-based software.

Chapter 4

Test-driven porting of the keystone back-end

The results of the study presented in Chapter 3 illustrate that test-suites which provide coverage of the front-end do not cover an equivalent amount in the back-end. It is desirable then, to ensure that the back-end is tested adequately, in isolation from the front-end. As there is a clear separation between the front- and back-ends of keystone, the back-end can be decoupled, ported and tested in isolation from the front-end. This chapter outlines the process behind test-driven porting of the keystone back-end to Java and the subsequent testing to ensure behavioural equivalence with the original C++ back-end.

4.1 Porting Overview

The porting strategy described in this chapter is based upon experiences of porting the back-end of the keystone system, written in C++, to a functionally-equivalent version written in Java, which is referred to as jKeystone. An important facet of the porting process was that it was *test-based* and *test-driven* in order to ensure that the porting was performed correctly.

Porting is defined as the meaning-preserving transformation from one pro-

gramming language to another. When programming language features become obsolete or another programming paradigm becomes widely used then it is often desirable to port useful programs and systems to a new programming language. Some examples of this include the original lexical analyser generator for C, *lex*, which has been ported to many other programming languages including Java, Python and Perl. Another example is the unit testing tool for Java, JUnit [BG98], which has been ported to C++ as CppUnit.

The porting of code does not automatically guarantee that the ported system will work exactly as the original. Errors may enter the ported code through a lack of comprehension of the features of the system to be ported, through subtle differences between the programming languages or through errors committed by the programmer. The ported system must be tested rigorously to ensure that errors have not entered the ported code.

The work in this chapter extends the *eXtreme Porting* approach outlined by Varma *et al.* for porting C/C++ applications [VAPT05]. This process operates by porting one class at a time and then unit testing to ensure that the ported class behaves as expected. Related clusters that have been ported are then integration tested. Once all of the classes have been ported, the entire system must pass some system level tests before the testing effort is complete.

When porting from one object-oriented language to another, we discovered that it is not altogether obvious as to what order the individual classes should be ported so that the use of stubs when unit testing the ported classes is kept to a minimum. In this chapter we define an *order* for the porting of classes from one object-oriented language to another using an *Object Relation Diagram* (ORD). Previous work on ORDs uses them to define an order for inter-class testing, and our contribution is to extend this to test-driven porting.

One feature of the keystone system is that it has a well-defined separation between its front-end, which parses C++, and its back-end which performs semantic analysis and outputs a summary of the program. We exploited this separation by using a record-and-replay testing harness [Mem02, HY05]. This technique

is commonly used in problem domains that share a well defined border between front-end and back-end, such as migration in back-end web services or the transformation of the underlying code of a GUI system.

Since keystone's output is a summary of the symbol table, we took the view that black-box testing would not provide satisfactory assurance that the ported version of keystone was working correctly. In order to verify the internal behaviour of the ported system we conduct white-box tests on crucial elements. The white-box tests are implemented using Aspect Oriented Programming (AOP) [KLM+97]. We exploit AOP by weaving similar aspects through both systems which have the effect of dynamically generating a UML sequence diagram for each test case. A comparison of sequence diagrams using Hirschberg's algorithm [Hir77] on a case-by-case basis then provides assurance that the behaviour is identical in each case.

4.1.1 Porting Strategy

The overall structure of the porting process was as follows:

Step 1 The keystone front- and back-end were decoupled. Fortunately, there is a clear distinction between these keystone phases, and the classes in the back-end are easily identified.

Step 2 The back-end keystone classes were then ported to Java. This consisted of three steps:

Step 2a Dependencies on external libraries were identified. keystone has relatively few library dependencies, mostly classes in the C++ Standard Template Library, and it was not difficult to identify classes with similar functionality in the Java class library.

Step 2b The jKeystone class hierarchy was generated. The class hierarchy of keystone was reverse engineered using an in-house tool, and the corresponding class hierarchy of jKeystone was generated. This was

then edited by hand to ensure that parameter and attribute types were correctly represented. No method-body code was ported automatically in this step.

Step 2c Each class in turn was then ported and unit-tested, in the vein of the *eXtreme Porting* approach.

Step 3 keystone's front-end was modified so that when run over a test case, it generated a Java test harness for that test case. These were then used to perform black-box system tests on jKeystone.

Step 4 Aspect Oriented Programming was used to weave similar tracing aspects across both keystone and jKeystone. For each test case, the aspected version of keystone and jKeystone produced a sequence diagram, containing the objects and interactions involved in processing that test case. These were then compared to verify the behaviour of jKeystone.

Steps 1, 2a and 2b are mostly routine, particularly when porting between relatively similar languages such as C++ and Java. While Step 2c is based on the approach outlined by Varma *et al.* [VAPT05], they do not address the important issue of what order the classes should be ported in; our solution to this is described in Section 4.2. The system testing using a test harness is similar to that used in GUI testing, but we extend this to apply to white-box testing using sequence diagrams. Both kinds of system testing are described in Section 4.3.

4.2 ORD-based porting

In this section we outline our algorithm for ORD based porting of a system. Our method for deriving the port order is outlined along with an example.

Since C++ is acknowledged as a difficult language to parse and analyse, constructing tools to aid in porting requires a significant effort. Hence there is a scarcity of freely-available tools that support the automated porting of C++ applications to other programming languages.

When moving from C++ to Java, the class and method signatures can be generated automatically, but the overwhelming majority of the code must be ported by hand. To ensure that any bugs that inadvertently entered the system during porting were caught and eliminated, the principles of eXtreme programming "test often and test early", were applied [Bec00].

4.2.1 A cost model for porting

We have devised an algorithm to cost an ORD specifically for the purpose of porting. Our algorithm is outlined in Figure 4.1.

The algorithm works by applying the cost model to all root classes of inheritance groups initially, as outlined in Step 1 of Figure 4.1. Root classes are defined as those that have an incoming inheritance edge but no outgoing inheritance edge. The weight calculation, described in Step 3 of Figure 4.1, then calculates the weight of the current node. It adds the current weight of the node to the *sum* of all the incoming composition and all outgoing association and inheritance weights. The sum function then recursively calculates the cost of every direct child node from the current node if an incoming Inheritance edge exists. Finally, Step 2 of the algorithm calculates the cost of all of the remaining edges within the ORD.

In order to demonstrate the operation of the cost assignment algorithm described in Figure 4.1, we apply it to the ORD shown in Figure 4.2(a). This ORD was originally presented by Briand *et al.* [BLW01]; we use it here to facilitate comparison of our approach with theirs. The ORD in Figure 4.2(a) has 8 classes, labelled *A* through *G*, with inheritance, association and aggregation edges labelled *I*, *As* and *Ag* respectively. In this example there are 3 inheritance edges, demonstrating both single and multiple inheritance, 3 aggregation edges and 11 association edges.

By following the algorithm outlined in Figure 4.1 we can cost our ORD as follows:

- We select a weighting for each of the edge types of $I = 5$, $As = 35$ and

Initialisation: Build the ORD labelling edges as inheritance, association and aggregation.
Assign weights to each of the *edges* according to our cost model.
Then propagate weights to the nodes by performing Steps 1 and 2 below.

Step 1: Identify the root nodes of inheritance hierarchies by choosing each node with with one or more *incoming* Inheritance edge and no *outgoing* Inheritance edges. Perform *Step 3* for each Node identified.

Step 2: Identify the nodes that have no *incoming* or *outgoing* inheritance hierarchies. Perform *Step 3* for each node identified.

Step 3: Calculate the weight for the current node as follows:

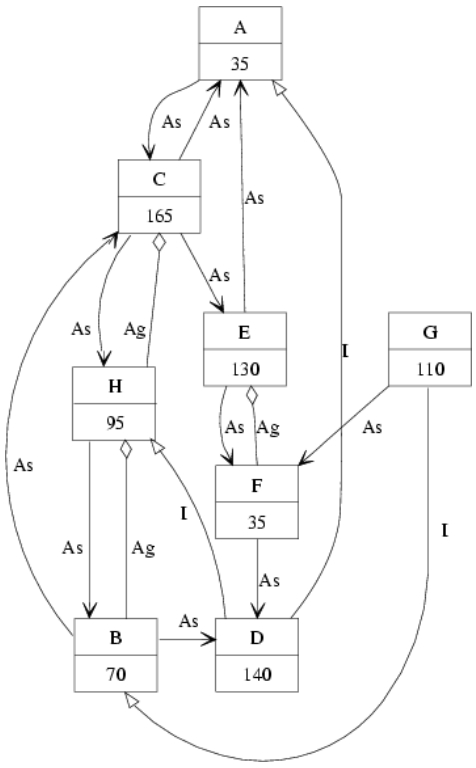
Step 3a: The weight is calculated as the current weight added to the sum of the *incoming* aggregations, the sum of the *outgoing* associations, the sum of the *outgoing* inheritances and the value of any weight parameters passed.

Step 3b: For each child of the current node, identified as an *incoming* Inheritance edge of the current node, perform *Step 3a* passing the child node and weight of the current node as parameters.

Figure 4.1: The algorithm to determine the porting order for classes. *When run over an ORD this algorithm assigns a weight to each class, and the class with the lowest weight is ported first.*

$Ag = 60$. These weights are based on Malloy *et al.* [MCL03] and adjusted heuristically, noting that the algorithm we use automatically assigns a relatively high priority to inheritance.

- Following Step 1, the root classes in the inheritance hierarchy are *A*, *B* and *H*; by applying step 3a we assign a weight to these classes based on their edges.
- We then perform Step 3b, propagating these weightings down the inheritance hierarchy in a depth-first manner, assigning weights to nodes *D* and *G*.



(a) An ORD with eight classes.

	Class	Weight	Stubs
1	A	35	C
2	F	35	D
3	B	70	C,D
4	H	95	None
5	G	110	None
6	E	130	None
7	D	140	None
8	C	165	None

(b) The porting order for the ORD

Figure 4.2: An example of using an ORD to define porting order. *Here, the edges have been weighted using the values I=5, As=35 and Ag=60. The table shows the order in which the classes would be ported and the stubs on which they depend.*

- Finally we apply Step 2 to those classes not so far covered, and assign weights to classes C, E and F.

The final values assigned to each class are given in Figure 4.2(b). This defines a testing/porting order where we start with the smallest weight and work upwards. In this example, just two classes, C and D need to be stubbed, and only the first three classes to be ported have direct dependencies on these stubs.

4.2.2 Porting keystone classes

Applying our algorithm to the ORD for keystone we can derive a porting order that involves the creation of only three stubs for unit testing during the porting. The ORD for keystone can be seen in Appendix A. In the ORD displayed in Figure A.1, the classes are numbered in the order that they were ported. Table A.1 gives a listing of the classes within the keystone ORD with each index mapping to the same numbered node within the ORD. The three stubs that are needed are the classes Scope, Type and ActionFacade respectively. The next section describes our system testing strategy for the newly ported system.

4.3 System Testing

The use of the ORD in our porting strategy ensures that as we port, the *architecture* of keystone is preserved. However it is also essential that the *behaviour* of the original system remains identical in the ported version. The unit tests provide some guarantee, but further testing is necessary to ensure that the system as a whole behaves as expected. In this section we describe the testing of the ported jKeystone system using back-box and white-box tests.

4.3.1 The System Test Suites

To test the system in order to ensure that jKeystone was a correct port of keystone, two separate test suites are used. The first is the implementation-based test-suite T_{gcc} and the second is the specification-based test-suite T_{ddj}, both described in the previous chapter. As the work in the previous chapter has highlighted, the creation of an effective minimal test-suite for applications receiving ISO C++ is difficult. The synthesis of test-cases via Purdom's algorithm produces test-cases that are semantically incorrect while test-suite minimisation via rule-coverage reduces the fault detection capability of the test-suites. Hence we use the full T_{ddj} and the full T_{gcc} suite for our system tests.

The *negative* test-cases in the T_{gcc}suite were not used for our system tests, reducing the total size of the test-suite to 1321 test cases. Negative test cases are not valid C++ programs, and are designed to ensure that the error handling of gcc is robust and correct. However, the use of the record and playback approach described below meant that a negative test case would not trigger the generation of driver code for jKeystone, and hence could not be used to determine the correctness of the ported code. This was not of concern in this project, but may be an issue for different kinds of application.

4.3.2 Record and playback

As outlined in Section 4.2, only the keystone back-end was ported in this phase. Thus, in order to complete the system tests for the ported system it was necessary to emulate the operation of the keystone front-end, a parser generated by the tool btyacc. The parser front-end is responsible for parsing the input file and calling the corresponding semantic actions in the back-end.

Record and playback was achieved by instrumenting the keystone parser front-end to generate Java code as it parsed its C++ input. Since the keystone parser was automatically generated using the parser generator btyacc it was contained in a monolithic file. While this usually causes problems for modular development, in this case it proved an advantage, facilitating the instrumentation process. The generated Java code produced from the instrumentation then acts as a driver for the test-case, calling the back-end routines in jKeystone.

Figure 4.3 presents an overview of the system testing process. An input test case, a C++ program from one of the test suites, is depicted on the left of the figure. When used as input to keystone this generates the normal test results, which are stored for comparison later. It also causes the instrumented keystone front-end to generate Java driver code that is specific to that test case. This front end is then run with the ported jKeystone, causing jKeystone to output a result that should correspond to the result produced by keystone. After this black-box test, aspects are woven through both systems to generate two sequence diagrams specific to the

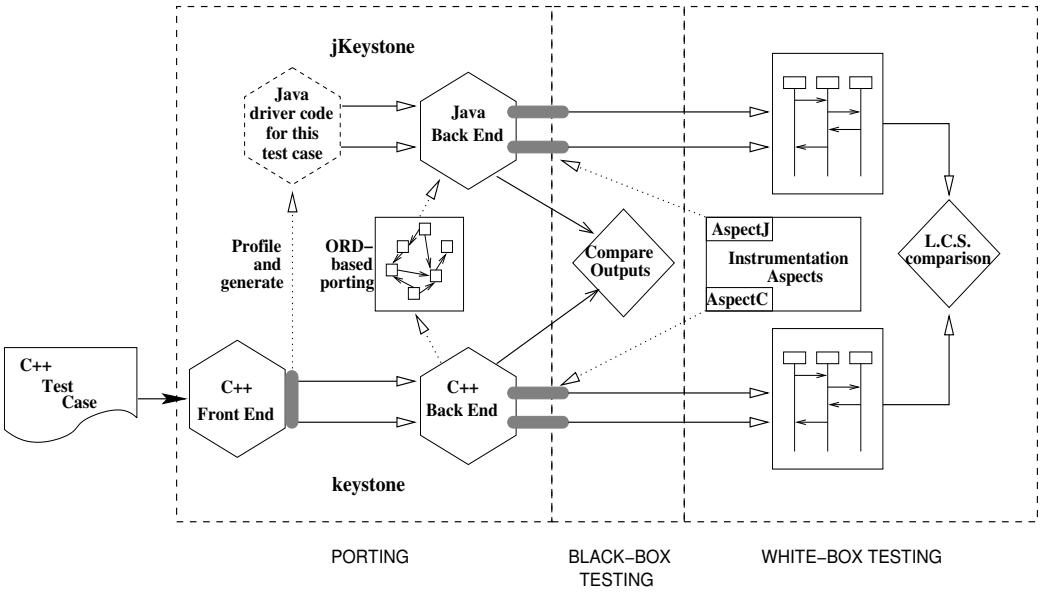


Figure 4.3: An overview of the porting process. *The lower half of the diagram represents the original* keystone *system in C++, and the upper half represents the ported* jKeystone *system in Java. The right part of the diagram shows the use of AOP to generate sequence diagrams, which are then compared using the LCS algorithm.*

test case, and these are then compared as a white-box test. Both of these processes are described in more detail below.

4.3.3 Black box testing

Black box testing is a method of testing where only the input and output by the subject under test are analysed. The black box testing of the ported jKeystone back-end was achieved using the T_{gcc} and T_{ddj} test-suites described previously.

We choose to use two outputs from keystone's static analysis as the test-case result criteria. The first of these is a summary of the tokens passed to the parser by the lexical analysis phase. keystone overcomes some of the ambiguity of ISO C++ by using a system known as *token decoration*, which, among other things,

generates context-sensitive identifiers. Since the correct assignment of context is crucial to the parse, the output of the lexical analysis phase was used as test output. Since this output can be generated using a command-line switch in keystone, this still technically constitutes a black-box test.

The second output used as a criterion in the black-box tests is the main output of keystone, which consists of a detailed summary of the symbol table information, including a summary of all of the scopes and their members. This information is solely dependent on how the program is parsed and the output constitutes a relatively detailed summary of the input test-case. If the output of a test-case is identical for *both* criteria during black box testing then the test-case is considered to have passed. If the output is not identical, then the white box testing allows the location of the difference in execution traces to be readily identified for bug-fixing.

4.3.4 Dynamic Sequence Diagrams

Since keystone is also used as an API, it is important that the internal behaviour of the ported system corresponds to the original. This cannot be readily verified by the end-to-end style of testing used in the black-box tests. White box testing involves the examination of the internals of the system under test.

Modern object-oriented features such as inheritance, polymorphism and dynamic binding mean that the effort in comprehending and reverse engineering a system from source code alone is high. Furthermore without access to an object's runtime type, it can become impossible to predict the flow of control through a system. The use of a UML sequence diagram can greatly aid in the comprehension of the behaviour of an object-oriented system [TOMG03].

Sequence diagrams are typically used in the design stage of a system, and depict the objects and interactions involved in a particular scenario of usage. However, given a system implementation, it is also possible to reverse engineer sequence diagrams. These diagrams can be reverse engineered either statically [RC05] through an analysis of the program source or dynamically from program execution traces [MP05, BLM03] or more recently by utilising aspect-oriented

programming [BLL05].

4.3.5 Using Aspects to generate sequence diagrams

Upon completion of a black box test, each test-case was investigated further to ensure that the corresponding sequence diagrams for keystone and jKeystone were equivalent. To achieve this, it is necessary to generate a dynamic sequence diagram from both systems for each test case. This entailed instrumenting object creations as well as method calls and returns in both systems. As such, it is an instance of one of the canonical application areas for aspect oriented programming.

Through the use of AOP, it is possible to weave an aspect through a program that captures a trace of its execution. Following the standard AOP approach, a set of *pointcuts* are defined to trace method calls and returns, as well as constructor invocations. The *advice* executed at each of these pointcuts printed a relevant message to a logging file, maintaining the sequence of the method calls along with the current level of nesting.

Another feature of AOP is the use of *introductions*, which allow the aspect to make static changes to a class, such as introducing an extra attribute. This was used to assign a unique numeric identifier to each object being profiled, so that it could be explicitly identified. By using the same introduction in the C++ and Java versions of the aspect, we ensured that objects were being created in the same order, at the same point in the program, and could thus assign a specific owner-object to each method call.

In general, one advantage of AOP is that the aspect can be coded separately from the main system, and added and withdrawn as necessary. In the context of porting, AOP offers the possibility that the aspect can be written once, and then woven into both the original and ported system. As well as economy of effort, this also helps ensure that the behaviour of the resulting code is identical in each case, and any differences result from differences in behaviour of the systems themselves.

In practice, it was not possible to share identical aspects between both sys-

tems. The original keystone system was profiled using aspects woven by *AspectC* [SLU05], which has almost reached maturity as a project since the preview release of version 1.0 of the program. The ported jKeystone system was profiled using aspects woven by *AspectJ* [KHH+01], a mature project that provides a compiler to weave aspects across Java source code. While the notation used by each aspect compiler is slightly different, the use of aspects did facilitate ensuring that the woven code performed similar actions in each case.

4.3.6 Comparison of Sequence Diagrams

The traces recorded by the aspect woven across keystone and jKeystone recorded the method currently executing, the owner-object and its unique identifier, and the current nesting level of the method. Figure 4.4(a) illustrates how our aspect captures the structure of a sequence diagram and Figure 4.4(b) shows its representation as a UML sequence diagram. Representing both traces textually allows them to be compared side by side to identify points at which program execution or object construction differ. However, the size of the diagrams involved ensures that manual comparison of the diagrams is infeasible for each test-case. A method for automating the comparison of different diagrams for each trace is desirable. An overly simplistic approach would be to use a utility tool such as `diff` and report on any differences. However there are a number of important factors that mitigate against this approach.

The first of these is that the convention for the evaluation of arguments differs between Java and C++. According to the Java language specification [GJSB05], the arguments must be evaluated from left to right. This is not strictly the case in C++ which does not explicitly specify an order. However as a legacy carried over from C compilers, and especially on `x86` architectures, arguments are evaluated from right to left [ISO98]. This small detail does not obviously affect the external behaviour of the systems but can change the generated sequence diagram if argument evaluation causes further method calls.

The second factor that must be accounted for is a potential difference in method

names across both systems. Method names may not remain the same across both systems, *e.g.* the method name *clone* could not be ported to Java due a clash with the *clone* method belonging to `java.lang.Object`. A rudimentary comparison of two diagrams containing the same execution but with different method names would flag this as a difference. Another factor is a slight difference in coding idioms between C++ and Java. In the C++ version of keystone, an iteration over a list data structure results in two calls, one a pointer to the beginning of the list, another to the end. The Java version makes just one call via the new *for-each* loop present in Java version 1.5.

The final factor relates to the tools used for the tracing itself. *AspectC* is only moving towards maturity as a project and thus is incapable of parsing all C++ constructs fully. To overcome this, it is possible to supply empty methods via macro definitions to *AspectC* to allow it to bypass difficult constructs. Thus for a very small percentage of the traces, the dummy code supplied to *AspectC* will cause its output to vary from that of *AspectJ*.

The issues outlined above mean that a comparison of the diagrams involves more than a simple use of `diff`. We have devised an automated approach to comparing the diagrams via a 5-step process:

1. The first step involves post-processing the Java version of the sequence diagram to fix the issue with the arguments being evaluated in opposite order to the C++ version.

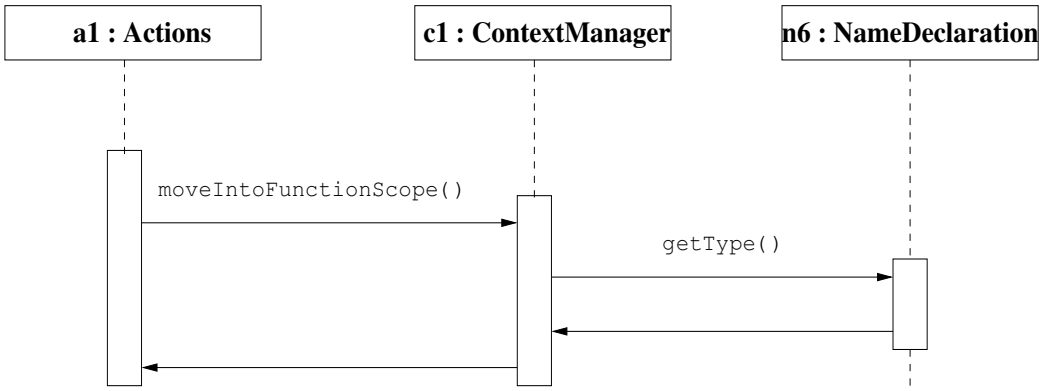
2. The percentage similarity of the diagrams is calculated through the use of Hirschberg's algorithm for identifying the longest common subsequence (LCS) [Hir77].

3. The inverse of the longest common subsequence is calculated from the output of Step 2.

4. The output from Step 3 is compared to all known differences that can occur through different method names or through a limitation of *AspectC*.

Actions	1	function_definition_1()	1
ContextManager	1	moveIntoFunctionScope()	2
NameDeclaration	6	getType()	3

(a) A sample of the execution data logged for a method call.



(b) A UML sequence diagram derived from program trace.

Figure 4.4: An example of the execution data and corresponding UML sequence diagram for a method call. *For each call we record the owning class and object identifier, the function name and parameters, and the nesting level of the call.*

---

5. If there are any remaining elements in the inverse that cannot be accounted for then the test-case had failed the white-box test and a bug fix must take place.

Hirschberg's LCS algorithm compares two sequences of characters and computes the longest common subsequence of (not necessarily contiguous) characters that they have in common. Using a full line of the sequence diagram as seen in Figure 4.4(a) for the characters, applying the LCS algorithm to a list of method calls, and taking the inverse of the LCS with the jKeystone sequence diagram, yields those method calls in jKeystone that did not occur in the original keystone. These are evaluated, a patch is applied, and the process is iterated until the se-

	Objects created		Methods executed	
	jKeystone	keystone	jKeystone	keystone
Min	3	3	8	8
1st Qu.	15	15	638	640
Median	23	23	1087	1091
3rd Qu.	34	34	1792	1800
Mean	31.50	31.50	2152	2152
Max	1150	1150	216141	216350

Table 4.1: A summary of sequence diagram size for the 1761 test-cases. *The results are partitioned into objects created and methods executed during the running of the test-cases. As well as the minimum and maximum encountered, the first, second and third quartiles and the mean are also given.*

quences are identical.

4.4 Results

In this section we outline the results of our test-driven porting strategy for the jKeystone back-end. We present a summary of the sequence diagram size and the overhead involved in their generation. We also present a classification of the types of bugs we have discovered through our testing.

4.4.1 Sequence diagrams

In order to demonstrate the scale of the activity involved in generating sequence diagrams, Table 4.1 gives a statistical summary of the UML sequence diagrams generated by both keystone and jKeystone. The results are partitioned according to the number of distinct objects present and the number of method calls in a sequence diagram. Within each partition, we further sub-divide the results between jKeystone and keystone. The first row in Table 4.1 gives the minimum number of objects created and methods executed in any single sequence diagram. We present

the same results for the first, second and third quartile. The average number of objects created and methods called is given in the fifth row while the maximum values are provided in the sixth row.

As well as demonstrating the infeasibility of manually comparing the sequence diagrams, the descriptive statistics in Table 4.1 provide a broad indication that the two systems are performing similarly. Object creation is a crucial operation of any object-oriented system and as can be seen from the average column, the number of object creations in both systems is identical across the 1761 test-cases. Furthermore the number of method calls is, on average, the same across both systems. However this number can fluctuate between test-cases, as outlined in Section 4.3.6 above, and further analysis on a case-by-case basis is crucial to ensuring the correctness of the port.

A more precise quantification of the discrepancies between keystone and jKeystone is given by applying the LCS algorithm and examining the differences. These differences in the sequence diagram contents, exposed by the LCS algorithm and expressed as a percentage of the size of the keystone sequence diagram, served as a metric of progress of the white-box testing. That is, the LCS algorithm was used not only to isolate bugs during testing and to ensure ultimate 100% identity between generated sequence diagrams, but also as a measure of the rate of progress of the white-box testing phase.

An analysis of using the LCS to quantify similarity between keystone and jKeystone at an early stage of white-box testing is given in Figure 4.5. This figure plots the percentage similarity between the two sequence diagrams before and after the fix for the method argument ordering is applied. The lower line plotted in Figure 4.5 shows the jKeystone sequence diagrams for approximately 1700 of test-cases exhibiting similarity of over 90% with keystone before any fix is applied. The upper line plotted in Figure 4.5 shows this increasing to over 97% for most of the test-cases once the fix is applied. Ultimately bringing this figure to 100% for all test cases allows us to determine with certainty when the white-box testing phase is complete.

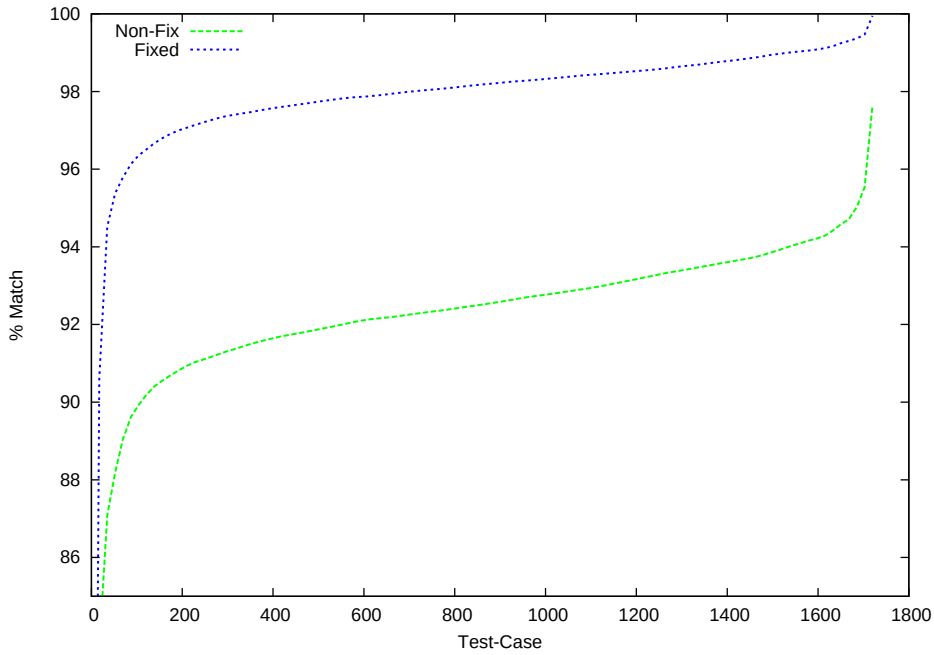


Figure 4.5: An analysis comparing sequence diagrams across all of the test-cases. *The test cases are listed along the x-axis, and y-axis represents the percentage match between the sequence diagrams generated from* jKeystone *and* keystone. *The lower and upper lines plot the percentage match for each test case before and after the fix for method argument evaluation is applied.*

4.4.2 Instrumentation overhead

In order for a testing strategy to be used it must be practicable; in particular, it must not impose an unreasonable burden on the tester. While the strategy of using UML sequence diagrams is based on an existing test suite, it is possible that the overhead of generating the diagrams would impose a significant overhead on the testing process. One immediate advantage of using AOP is that we can turn sequence diagram generation on or off very easily.

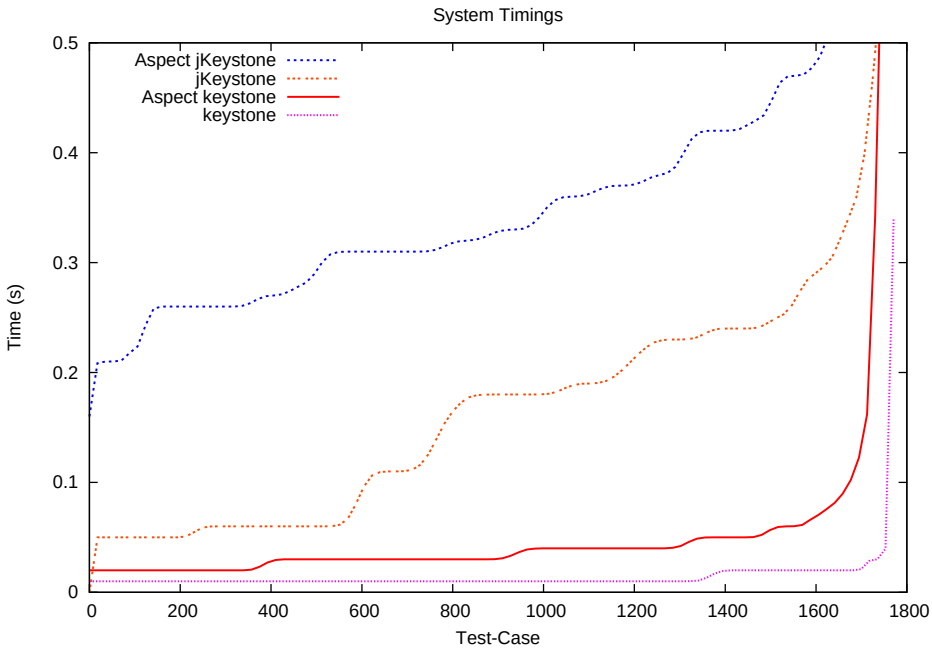


Figure 4.6: The timing results for each test case. keystone *and the aspected version of* keystone *are the two bottom plots whilst* jKeystone *and the aspected* jKeystone *form the top two respectively. Test-cases with timings greater than 500ms have been removed for clarity.*

To examine the overhead, we tracked the timing of generating sequence diagrams for both keystone and jKeystone, and the results are summarised in Figure 4.6. The timings were performed on a *Dell* Optiplex GX280 PC, with a 3.06Ghz Intel processor, 1Gb DDR RAM running the Fedora Core 4 distribution of GNU/Linux.

The graph in Figure 4.6 shows four sets of timings, one each for keystone with and without sequence diagram generation, and one each for jKeystone with and without sequence diagram generation. Each point on the graph represents a single

Anomaly	Freq.	Actual Occurrence
IV310	3	Assertions incorrect
IV315	4	Code ported incorrectly
IV316	9	Guard clause inadequate
IV321	1	Guard clause too strong
IV317	3	Wrong variable checked
IV321.4	2	Scope of statement incorrect
IV341	3	Object initialised incorrectly
IV342	2	Accessed incorrect data
IV342.1	1	Return type hard-coded
IV342.4	2	Data accessed out of bounds

Table 4.2: A classification of the bugs identified during out system tests. *The IEEE anomaly index is given along with the frequency of the bug and the actual manifestation of the bug.*

test case, and the test-cases have been arranged in increasing order of size along the horizontal axis.

The first point to note in Figure 4.6 is the C++ / Java divide in the timings. The C++ based keystone gives an almost constant 10 milliseconds for almost 1400 of the test-cases whereas the aspected keystone takes twice the time to generate the sequence diagrams. The timings for jKeystone range between 300 and 400 milliseconds for approximately the first 1400 test-cases. When sequence diagram generation is turned on, the timings rise by approximately 50% across the test-cases.

This divide between the systems is not unexpected. Many of the test cases are quite small, and the overhead of JVM startup imposes a significant penalty on jKeystone. It is positive to note, however, that the aspects do not place an unfeasible bound on the time taken to analyse a given test-case in either keystone or jKeystone.

4.4.3 Bug classification

Table 4.2 gives a breakdown of the bugs that were discovered using the sequence diagrams during the system tests. In total, 30 unique bugs were discovered during the system testing phase. The bugs are classified according to the IEEE classification for software anomalies [Sof93], the number of each instance of anomaly and the specific occurrence of the anomaly within jKeystone. The table is partitioned along the following lines: bugs due to misunderstanding of the original code and bugs introduced during porting. Rows 1-3 highlight the bugs introduced through misunderstandings whilst 4-10 are the bugs inadvertently entered during porting.

The bugs featured in rows 1-3 of Table 4.2 are the result of a shortfall in the comprehension of specific areas of keystone. The largest number of bugs within this category are from **IV 316**: "*missing condition test*". This is due to the fact that a pointer can be dynamically cast within an if statement in C++. This cast check was missing initially from the Java version. The bugs featured in rows 4-10 of Table 4.2 are bugs that entered jKeystone through human error. These bugs are spread over a number of anomaly classifications and can be considered to be "one-off" bugs. That is, the discovery of a bug did not unearth a plethora of similar bugs. Thus the use of the dynamic UML sequence diagrams was essential in isolating the difference in method calls across the two systems and discovering the approximate location of the bug within jKeystone.

4.5 Synopsis

In this chapter, we have outlined the successful test-driven construction of the Java back-end jKeystone by porting the existing back-end of keystone. The construction of an ORD via the algorithm outlined in Section 4.2.1 within the eXtreme Porting strategy allows the porting to take place in a manner that minimises the need for stubs during unit testing.

In addition to ensuring that the structure of the original system remains preserved under porting, it is possible to create a sequence diagram for the purposes

of comparison. Thus, a crucial phase of our testing is the ability to reverse engineer dynamic UML sequence diagrams via aspect-oriented programming. These reverse engineered sequence diagrams can be then compared side-by-side to prove that the behaviour is preserved across both systems.

The use of record and playback allowed the existing keystone front-end to be instrumented to generate driver code that simulated the operation of the front-end. The next chapter outlines the starting steps needed to create a fully functional front-end for jKeystone to replace this driver code.

Chapter 5

The design, implementation and testing of a GLR parser generator for Java

In this chapter, we describe the development of a GLR parser generator for Java. We also describe the approach used to develop, test and validate the generated parsers and the GLR algorithm implementation. The stages involved in creating the GLR parser generator are described alongside the black- and white-box test methodologies that were used in testing.

The processes described in this chapter are as follows:

1. The GLR parser generator is tested by using bison as an oracle for the generated parsers.

2. The generated parsers are tested by using Purdom's algorithm to generate test-cases from grammars for a selection of programming languages.

3. Finally, the implementation of the GLR algorithm is tested through the use of mutation testing on the parse-tables.

5.1 Creating a GLR parser generator

As outlined in Section 2.1.2, an LR parser *generator* is an automated tool for translating a grammar specification into LR parse tables that can be used by a driver program. This driver program uses a simple stack to push and pop terminals and non-terminals as the parse takes place. The grammar specification passed as input to the parser generator may contain conflicts and ambiguities, but, as discussed previously, these are typically resolved by choosing a shift in a shift-reduce conflict and the rule that occurs first in a reduce-reduce conflict.

Thus, the first stage when creating a GLR parser generator is to devise a mechanism where the conflicts are retained and encoded within the parse tables. This can be achieved by constructing an extra table to keep a list of all of the conflicts within the grammar specification. A pointer or symbolic entry within the standard action table can be used as an index into this conflict-table to enable *all* of the conflicting actions to be accessed as the input is parsed. Thus, for a shift-reduce conflict an entry is placed in the action table indicating that a conflict is present and an entry into the conflict table is made to allow not only the shift action to be processed but also all of the reduces as well. This is in marked contrast to canonical LR parsing, where the reduce actions would have been discarded at the parser generation time. The removal of these reduce actions at parser generation time has the effect of decreasing the number of valid sentences that the canonical LR parser can accept.

The second major change to the standard LR parser generator is to change the driver program so that it is no longer dependent on a single stack. The GLR algorithm calls for the use of a data-structure known as a Graph-Structured Stack (GSS), that is composed of many nodes. Each of these nodes must maintain, at the very minimum, their state and a set of links back to other stack nodes within the GSS. Each link may be labelled with a parsed symbol, either a terminal or nonterminal. For a deterministic parse, the GSS mimics the behaviour of a simple stack, with a solitary top of the stack pointing to its previous stack node. It is only in the presence of conflicts that the GSS exhibits the behaviour described

previously in Section 2.3.2.

The final change is to implement the algorithm in the driver program to utilise the properties of the GSS, such as forking to create new branches of the GSS on encountering a conflict, and sharing states at the top of the stack. The algorithm implemented in this research is Farshi's algorithm [NF91], presented in [Rek92, p25]. The modifications outlined above were applied to the existing LALR parser-generator, JavaCup [1], a bison clone for Java [Hud97]. We refer to our modified GLR parser generator as *JavaCup*$^+$ throughout the rest of this thesis. As we were unaware of any formal testing of the original port, it was necessary to test the parser-generator itself and compare the generated parsers directly with those generated by bison.

5.2 Testing the Parser Generators

In its most basic form, an LR parser generator receives a specification for a context-free grammar in a well defined format as input, and outputs a two-dimensional array that represents an encoding of all of the transitions in the LR state machine. Thus, it is possible to black-box test a parser generator by using a test-suite of grammar specifications and comparing the outputs to another parser generator's output. In the case of this research, we took the established parser generator, bison as the test oracle for testing the outputs of *JavaCup*$^+$.

The output from *JavaCup*$^+$ can be compared to bison under the following categories for each grammar used as input:

Number of States A simple check to see if there is an identical number of states in the state-machine for the grammar.

Number of Symbols If the number of terminals and non-terminals are identical then it is possible to form a one-to-one mapping of the symbol numbers.

[1] Java Constructor of Useful Parsers

Identical States By utilising the verbose output of both tools, it is possible to produce a list of every LALR item within a state. By sorting all of the items within a state and then by sorting each state, it is possible to check if the items in the states are identical. If there is a one-to-one correspondence between each and every state, then we can derive a mapping between the states of the two different tools.

Number of Conflicts Checks if the number of shift-reduce and reduce-reduce conflicts are identical for a given input grammar.

By processing the output *JavaCup*$^+$ action table and applying the mapping for each state and symbol, it is possible to translate the *JavaCup*$^+$ action table so that it can be compared side-by-side with the bison table. If the two tables are identical, then we can say with certainty that the generated parsers are identical because the parse tables represent an encoding of all of the possible paths through the LR state machine for the grammar.

5.2.1 The Grammar Test-Suites

A number of implementation and specification test-cases were used for this study. The implementation-based test-suite consisted of a number of grammars for popular and widely used programming languages while the specification-based test-suite contained some research grammars to test the functionality of the GLR algorithm. The grammars in the implementation suite, whose basic dimensions are summarised in Figure 5.1, were as follows:

C This is the ISO C grammar specification, which was standardised in 1990 [ISO90]. This grammar is quite small and contains only a solitary shift-reduce conflict, that of the "*dangling else*" problem.

Pascal This is the grammar taken from ISO Level 0 Pascal standard specification [ISO83] which is based upon Wirth's original Backus-Naur specification of the grammar. This grammar is free of conflicts.

Grammar	# Rules	# States	#Conflicts S/R	#Conflicts R/R	Avg. # Reduces Per Conflict
Implementation Grammars					
C	212	350	1	0	1.0
Pascal	254	410	0	0	0
Java 1.5	595	1073	0	0	0
$C^{\#}$	815	1291	49	547	7.25
Cobol	1931	2699	20888	23349	1.66
C++ Grammars					
ISO C++	482	822	407	135	2.63
Elkhound	497	908	58	70	1.62
Keystone	500	892	526	54	1.20
Willink	559	894	24	0	1.0
Roskind	643	1235	24	18	1.64
Specification Grammars					
Gamma	5	7	2	2	1.25
$\tau 1$	7	13	3	0	1.33
$\tau 2$	4	6	0	0	0
$\tau 3$	4	5	4	2	1.17
$\tau 4$	5	8	4	0	1.0

Table 5.1: An overview of the grammars used to black-box test the parser generators. All of the figures given are output identically by both *JavaCup*$^{+}$ and bison.

Java This grammar, free of conflicts, is the most recent specification of the Java programming language version 1.5 and includes all of the recent additions to Java, such as generics, enhanced for-loops and enums [GJSB05].

C$^{\#}$ This recently standardised grammar for Microsoft's $C^{\#}$ programming language [ISO06] continues the move away from specifying grammars in a manner such that they may be processed by LALR parser generators without modification. This grammar specification contains 49 shift-reduce conflicts and 547 reduce-reduce conflicts which makes the construction of a parser front-end for this grammar using parser generators quite difficult.

Cobol This is the VS-Cobol II grammar published by Lämmel [RL03]. The published extended Backus-Naur format was converted into a format suitable for processing by LR parsers in the Grammar Tool Box [JSE04]. This grammar is highly ambiguous and contains conflicts in an order of magnitude greater than the C# grammar. The grammar contains almost 21 *thousand* shift-reduce conflicts and over 23 *thousand* reduce-reduce conflicts.

ISO C++ This is the standard specification for the C++ programming language and the construction of a parser-front end for this language is one of the primary motivations of this research. This grammar, taken from Annex A of the ISO standard from 2003 [ISO03], contains a high degree of ambiguity, as outlined in Section 2.2 due to the presence of context-sensitive identifiers.

In addition to the ISO C++ standard, we add a number of different grammars for C++ to the implementation test-suite.

Elkhound The GLR parser generator Elkhound, has been used to generate a parser front end for C++ [McP02]. This parser is known as ELSA and is based upon a modified grammar derived from the standard. This grammar is far less ambiguous than the standard and contains only 58 shift-reduce conflicts and 70 reduce-conflicts. This grammar has been used to successfully parse large "industrial" sized C++ applications.

Keystone This grammar, taken from keystone version 0.2.3 is almost identical to the ISO standard except that this grammar marks up the context-sensitive identifiers through the use of token-decorated parsing. Consequently this grammar is less ambiguous than the standard as the number of reduce-reduce conflicts is halved.

Willink The number of ambiguities in the C++ grammar can be reduced completely by creating a superset of the grammar standard [Wil01] and disambiguating after the parsing is complete. This grammar produces only 24 shift-reduce conflicts and no reduce-reduce conflicts.

Roskind This grammar was one of the first attempts at creating a grammar for C++ [Ros89]. This grammar was written before language features such as namespaces and templates were implemented. Consequently, the grammar is relatively free of ambiguities but is of little use for parsing modern C++ programs.

In addition to the implementation test-suite, a small specification test-suite was drawn up with examples from [JSE04] and [SJ06].

Gamma This grammar is designed to test the worst case performance of the GLR algorithm. This grammar contains only 5 rules but has 2 shift-reduce and 2 reduce-reduce conflicts.

τ1-4 These small grammars are designed to test GSS construction. τ1 and τ4 contain conflicts. τ2 has no conflicts and τ3 is a pathological grammar containing cycles.

The summary of testing *JavaCup*$^+$ versus bison is presented in Table 5.1. This table shows the number of states, rules and conflicts per grammar. As each grammar was written "cleanly", i.e. without static rules to allow certain arithmetic operators have higher precedence than others, this allowed a side-by-side comparison of each grammar without bias. Consequently certain grammars, such as Elkhound and Willink exhibit more conflicts than their published specifications.

For each grammar shown, the generated parse table was identical for both *JavaCup*$^+$ and bison thus demonstrating the robustness of the *JavaCup*$^+$ port. The final column of Table 5.1 lists a summary from the conflict table of a measure of how ambiguous the grammar is. This column represents the average number of reduces per conflict. The greater this number, the greater number of branches that will be created in the GSS during the parse, and consequently the slower the parse will be. Thus from this crude metric it is possible to see that for each conflict that is encountered in ISO C++ during a parse, typically at an identifier, there are on average 2.6 branches from that point in the GSS. This contrasts with the keystone

grammar, which has marked up the context-sensitive identifiers, which only has an average of 1.2 reduces per conflict. Thus for a large input for an unmodified ISO C++ grammar, the GSS would become very large quite quickly.

Table 5.2 shows that statically *JavaCup*$^+$ produces identical parsers to those of bison. However it is crucial to ensure that these generated parsers perform as expected, that is, they recognise and accept valid sentences from the grammar and reject invalid ones.

5.3 Testing the Generated Parsers

The validation of the generated parse tables is only a first step in testing that parser. The runtime operation, i.e. the actual parsing itself, must also be rigorously tested. In this work, each generated parser from the from *JavaCup*$^+$ and bison GLR mode, are both black- and white-box tested for each of the grammars described.

The black-box testing consisted of passing a small test-suite of inputs for that grammar to the parser. The acceptable result was a successful parse for valid inputs and a rejection of invalid inputs. The white-box tests involved measurement of the coverage of the states of the LR state machine.

5.3.1 Black-Box Tests

The test-suites for the black-box tests were constructed using Purdom's algorithm for the automated generation of test-cases from a grammar. Although studies have shown that this algorithm is insufficient for testing compilers due to the fact that the test-cases produced are not semantically correct [MP01], the testing of a parser is a canonical application of Purdom's algorithm as only the syntax is taken into account.

The results for the black-box tests are presented in Table 5.2. The number of test-cases generated by each grammar is given along with the average number of tokens per test-case. With the exception of grammar $\tau 3$, every test-case for every

Grammar	# Test Cases	Avg. # Tokens Per Test-Case	% Unique States Covered	# Passed
Implementation Grammars				
C	21	17.29	99.71	21
Pascal	6	66.5	100	6
Java	50	27.12	96.83	50
C#	129	17.23	93.34	129
Cobol	113	45.36	89.29	113
C++ Grammars				
ISO C++	40	19.7	98.78	40
Keystone	50	18.54	98.88	50
Willink	57	16.18	97.32	57
Elkhound	69	13.39	92.73	69
Roskind	180	10.38	90.53	180
Specification Grammars				
Gamma	2	2.5	100	2
$\tau 1$	4	3.75	100	4
$\tau 2$	1	1.0	100	1
$\tau 3$	2	0.5	60	0
$\tau 4$	2	1.0	100	2

Table 5.2: The results of testing the generated parsers via black- and white-box testing. The first column lists the grammar, with columns 2 and 3 providing information about the number of test-cases and the average size of the test-cases generated by Purdom's algorithm. The coverage figures for the white-box tests are listed in column 4, while the final column lists the number of positive test-cases *accepted* by the generated parser.

---

grammar is accepted as a valid input for that grammar. The grammar presented in $\tau 3$ is a pathological grammar designed to be cyclic, thus any input to its generated parser causes a failure as expected. As these test-cases are generated directly from the grammar, the expected result is that they will all be accepted by the parser. In this experiment, it was not possible to compare the behaviour with that of bison. This is because the GLR mode of bison is not advanced enough to parse all of

the ambiguities without user-modification of the grammar to add dynamic precedence rules which allow for disambiguation. This is a time consuming process and for grammars such as C# and Cobol, it would require a lot of manual effort to customise the grammar so that it is acceptable to the GLR mode of bison.

5.3.2 White-Box Tests

The black-box tests give a simple "yes" or "no" for the Purdom test-case inputs. However, it is important to ensure that every state in the generated LALR state machine is covered by a test case.

As Purdom's algorithm is designed to create a test-case for every rule, it may seem intuitive that there would be a one to one correspondence between rule coverage and state machine coverage. However, an examination of the data in the fourth column of Table 5.2 shows that this is not always the case. This result was surprising and initially it was assumed that the ambiguities inherent in most of the grammars was causing Purdom's algorithm to fail to generate test-cases for all of the rules. Upon further examination, this was found to not be the case. Indeed, the test-suite for the unambiguous Java grammar specification only covered approximately 97% of the state machine. This behaviour was replicated in the bison generated parser for Java also, thus ruling out a problem with the *JavaCup*$^{+}$ parsers.

The reason for the lack of total coverage therefore, comes not from a deficiency in the generated parsers but from the algorithm that is used to construct the LALR state machine itself. Within the state machine it is possible to have an identical item with a dot at the end in two different states. As Purdom's algorithm is concerned only with ensuring that the rule is covered at *least* once, states with valid reduces may never be encountered by Purdom's algorithm. As the results in Table 5.2 show, the coverage shows a lower bound of approximately 90% for every grammar which was felt acceptable for the purposes of this stage of the testing. However to ensure that the implementation of the GLR algorithm was correct in the absence of a direct comparison with bison, some further testing strategies

were employed.

5.4 Testing the GLR implementation

As outlined in Section 5.3.1 above, it was not possible to compare the generated parsers directly with those generated by bison due to the limitations of the GLR mode in bison. Thus for the purposes of testing our GLR implementation, we were left with the simple black-box tests generated by Purdom's algorithm. As shown above, these test-suites were all positive, i.e. it is expected that each test-case should be accepted by the parser. Thus to prove that the GLR implementation is functioning as expected, it is necessary to show that the parser also *rejects* invalid input.

To demonstrate this, we employed mutation testing techniques to mutate the parse tables. This technique allows for the automated mutation of every element within the action table. This process is guaranteed to test the algorithm more effectively than arbitrarily mutating individual tokens within a test-case.

5.4.1 Mutation Testing

The mutation testing was designed to adequately test the GLR algorithm implementation in the absence of a direct comparison with bison. The testing was methodical, with each entry in the action table being mutated before the full test-suite was executed. Upon completion of the test-suite, the original value was restored before the next entry was mutated.

The mutation operators applied were similar to those outlined in Section 3.5. Each action in the range `-32767...-1` and `1..32766` was mutated twice using the absolute value replacement mutation operator. That is, each value was replaced by its negated value and zero. The values, `-32768`, `0` and `32767`, representing a reduce-reduce conflict, an error and a shift-reduce conflict respectively, were replaced by a random number generated in the range of `1..892`, where 892

# Mutants	# State/Symbol Pairs Covered	# Mutants Applied	# Mutants Caught Directly	# Mutants Caught Indirectly
51905	2011 +(143)	4022 +(143)	3759	406

Table 5.3: The results of applying the mutation tests for the keystone grammar. *The number of mutants caught directly via external comparison is shown in column 4. The number of mutants caught via observation of the number of branches killed off in the GSS is shown in column 5.*

represents the highest index for a state in the LR state machine for the keystone grammar.

Since producing a new parser front-end for ambiguous keystone grammar to integrate with the ported back-end is one of the main goals of this research, a parser for the keystone grammar was tested with the mutations. The mutations were applied to every symbol entry in every state of the action table. The results of applying mutations to the keystone grammar are presented in Table 5.3.

The total number of mutations applied to this parse table was 51,905. However the test-suite generated by Purdom's algorithm covered only 2,154 of the state/symbol pairs. As stated, Purdom's algorithm does not generate test-cases that cover every possible state/symbol in the grammar. Of the 2,154 state/symbols pairs that were covered, 143 of these pairs resulted in either a shift-reduce conflict, a reduce-reduce conflict or an error action, hence these entries were only mutated a single time. The other 2011 entries had two mutations performed on them. The expected result of applying each mutation was that there would be at least one less test-case accepted whilst running the full keystone test-suite. However from the fourth column of Table 5.3 it is obvious that not all of the mutations applied resulted in less test-cases being accepted by the algorithm. Thus closer examination of the internals of the algorithm is needed during a parse.

The fourth column of Table 5.3 shows the number of mutants caught through direct observation i.e. there was at least one test-case that failed to be recognised by the parser. The number of mutants displaying this expected behaviour was

less than the number of mutants applied. Thus, a finer measure of the parser's behaviour was needed to verify the effect of the mutations. As the GLR algorithm is far more tolerant than traditional LR parsing, it is likely that the mutations are being caught, but only internally within the GSS at runtime as discarded branches. By instrumenting the GLR algorithm to record the number of discarded branches during a parse, it becomes possible to measure how many mutants are caught *internally*. The figures listed in the fifth column give the number of mutants that were caught in this manner which brings the coverage up to 100% of the state/symbol pairs covered.

5.5 Synopsis

The work outlined in this chapter has shown the successful validation of the generated parsers against the *de facto* standard parser generator bison. Furthermore the correct runtime operation of these parsers for many large and practical grammars has been displayed. Finally the GLR implementation was tested using mutation testing techniques. *JavaCup*$^{+}$ has been shown to work as well as bison for deterministic grammars and is capable of recognising true ambiguities within an input without any need for modification to the specification. This is in direct contrast to bison's GLR mode which must have user-specified code for every possible merge. This is acceptable for smaller grammars but does not scale well for grammars such as C$^{\#}$ and Cobol.

The implementation of the GLR algorithm in recognising valid input and rejecting invalid input was the first step in creating a working front-end for the keystone C++ grammar. However, a fully functional parser needs to do more than accept or reject input, it must be capable of calling the semantic actions that drive the construction of the *Abstract Syntax Tree* (AST) for the input given sentence. The implementation of semantic actions in a GLR parser generator is not a straightforward task, and is discussed in the next chapter.

Chapter 6

Unifying the system using a hybrid approach to GLR semantic actions

The work in the previous chapter outlined the construction of a GLR parser generator and the creation of parsers for many of the popular programming languages alongside a number of dialects of C++. The construction of a parser alone is of limited practical value when seeking to interact with other tools and in particular with the keystone back-end. Hence the parser must be capable of executing arbitrary user code, typically known as *semantic actions*, whenever a grammar rule is parsed. In this chapter we outline the challenges encountered and the solutions developed during the unification of the generated GLR parser front-end and the ported keystone back-end.

6.1 Semantic actions in GLR

The work in the last chapter outlined the construction and testing of an enhancement to the parser generator JavaCup, to output GLR parsers. The testing of these generated parsers was designed to ensure the robustness of the algorithm in recognising valid input and rejecting invalid input. However, in order for a parser to be useful it must be able to not only recognise if a given sentence is valid, but it also

must be able to produce further specific output based upon the meaning of the input sentence. Thus in general, the generated parsers need to be able to store and pass information gleaned during the parse on to other tools or stages. In the case of jKeystone, the generated GLR parser for the keystone grammar needs to be able to communicate with the ported back-end.

The established method to achieve this is to embed fragments of user-code with the grammar-specification for the language. These code fragments are known as *semantic actions* and are typically executed as soon as a whole right-hand side of a rule has been recognised. For a grammar free of conflicts, this strategy could allow for the parser to output an abstract syntax tree directly from the given input in a single pass.

As we have seen in section 2.1.2, if a grammar-specification contains ambiguities, then an LALR parser generator will either reject the specification or will start to discard valid parses during the parse-table construction phase by always choosing shifts or the first reduce in a conflict. Hence, the construction of a parser that permits ambiguity and allows arbitrary semantic actions is not an easy task.

The strategy employed by bison since the GLR mode was introduced is to suspend the semantic actions upon encountering a conflict, record the interim rules chosen and resume the semantic actions as soon as the parse has been determinised. A similar strategy is also employed by the btyacc system which was used to generate the front-end for keystone. We decided to adapt this strategy for our GLR parser generator also.

The process for executing semantic actions during an ambiguous parse works as follows:

1. When the parse is deterministic it works identically to other parsers such as those generated by bison with the semantic actions executed as soon as a rule is reduced.

2. On entering a state containing a conflict, the parse is split into as many parses as there are conflicts. All semantic actions are disabled during the non-deterministic phase.

3. The multiple parses then continue through the use of the GLR algorithm. Each branch in the GSS can determine which rule was responsible for its creation. During this phase, all tokens are buffered to enable the future replay of the parse.

4. As soon as the ambiguity is resolved, the parse rolls back to the global split point. The current token is set to the beginning of the token buffer and semantic actions are re-enabled. The parse continues then as normal until the next conflict is detected.

The process in step 4 is based upon either: (a) all of the parses bar one being killed off due to local ambiguity or (b) a commit to a valid parse from one of the branches.

There are a number of practical issues associated with adding semantic actions to a GLR parser. These are:

1. **Split Points** When a conflict is encountered, the node containing the state and symbol must be recorded so that at a later stage the parse can be rolled back once a valid branch has been selected. A further complication, specific to token decoration implementation of keystone, is that the execution of semantic actions can change the token type information for identifiers. Thus nested split points are not stored. A nested split point occurs when a conflict is encountered and a split point already exists. To overcome the problem of semantic actions having global side-effects that can influence the parse, only the outer-most global split point is recorded. This ensures that the parse will roll all the way back to the first split point to resume the execution of semantic actions and continue the parse.

2. **Branch Descendants** Once a conflict has been encountered and the branches have been created, there exists a general problem of determining what parse-action was responsible for the creation of that branch. If multiple branches exist for a large duration of a parse then it must be possible at any stage to

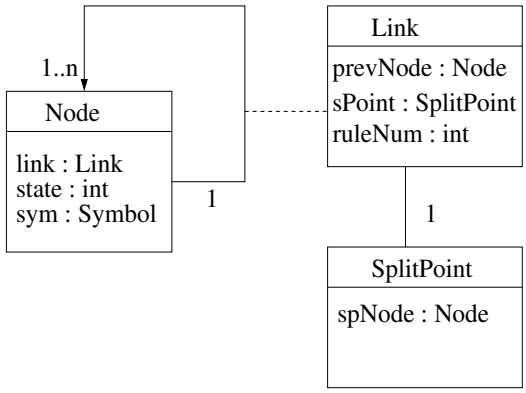


Figure 6.1: The UML class diagram for the classes involved in the GSS. *The Node class is used to store the current state and the semantic action, represented by a symbol. Each Node maintains a list of many links, each of the type of class Link. The class Link itself contains a reference to a single Node and a reference to the current global ambiguity represented by the SplitPoint class. This class maintains a reference to the global conflict and keeps track of which rule number the branch belongs to.*

---

identify which rule created a given branch. This enables a commit point to identify the parse action that was responsible for the creation of the branch we wish to preserve. The parser can then roll back to the split-point and continue using the committed rule.

3. **Replay of the Parse** Once the correct branch has been chosen or all of the remaining branches have been removed due to the ambiguity being local, the rule that created the remaining branch can be determined and the whole parse can be rolled back to begin deterministic parsing again.

   Once a conflict is detected, the process is repeated as nested conflicts are not determinised during the trial parse due to the fact that the symbol table information could be inaccurate.

The UML class diagram in Figure 6.1 shows the design of our GSS that allows for the resolution of the three issues above. The class Node is responsible for the

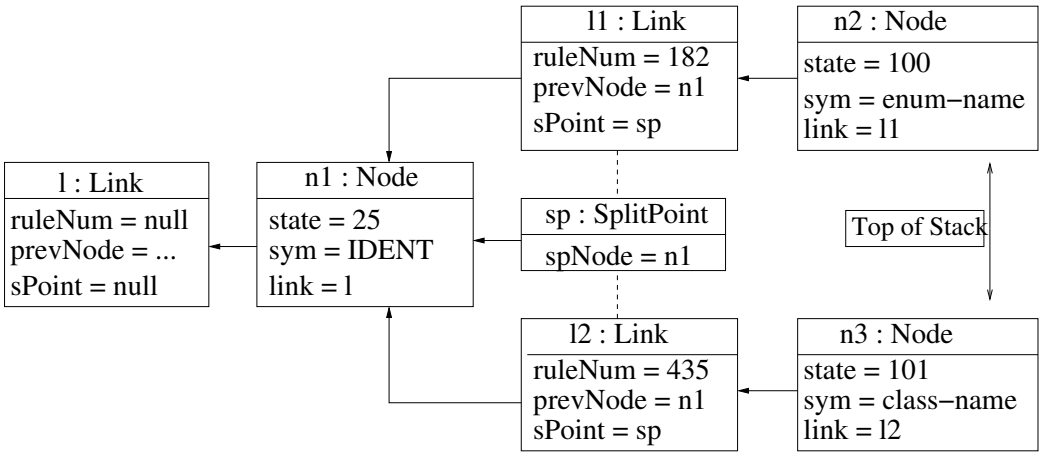


Figure 6.2: An instance of the GSS displayed as a UML Object Diagram. *In this example the Node instances maintain one or more Link instances. Each Link instance points to a solitary Node, with each Link instance maintaining a reference to the global split-point. Each Link object also stores the number of the rule that caused the branch.*

---

creation of the individual nodes in the GSS. These nodes maintain information about the current state alongside the semantic value associated with each grammar rule. Each Node instance then maintains a set of one or more Link instances. Each Link instance contains a reference to a single node, one level below in the GSS. The Link also stores the rule number that was responsible for the creation of its branch and it also contains a reference to the global split point that was responsible for the conflict. This design allows for the merging and splitting in the GSS to take place alongside the execution of arbitrary semantic actions.

The UML object diagram in Figure 6.2 shows an instance of a GSS during an ambiguous parse. In this case, the reduce-reduce conflict between rules 182 and 435 causes two active branches to be created. The top of the GSS then becomes `n2` and `n3` which refer back to `n1` via the links `l1` and `l2`. Both `l1` and `l2` refer to `sp`, an instance of the `SplitPoint` class. The purpose of `sp` is to ensure that a live reference to `n1` is kept so as to prevent Java's automatic garbage collector from

deleting the object. If `sp` were not kept then as soon as `n1` and `n3` are reduced past the split point the reference would be lost.

6.1.1 User-defined determinisation rules

The GLR algorithm allows for the parsing of non-deterministic inputs for grammars that contain conflicts. Simple checking to see if the inputs are valid is a relatively straightforward task, as no semantic output is required from the parsing stage. When semantic actions are required, the flexibility provided by the GLR algorithm poses a question: how do we pick the "correct" parse such that its semantic actions can be executed? There are two main ways of achieving this:

1. The first strategy, implemented by btyacc, is to allow special actions that can be embedded within the grammar specification that cause a rule to be accepted or rejected during a parse. There are two types of these special actions. The first, known as a *commit* action, keeps the branch that executes the commit action and discards the remaining branches. The parse will roll back to the point of the original conflict and resume the semantic actions choosing to use the committed branch. The second type of special action is a *reject* action. A reject action will cause a depth first parse to discard the current branch and try another. These special actions work well with the depth first approach of btyacc.

2. The second strategy, used by the GLR mode of bison, is to assign dynamic precedence to the grammar rules, so that one of two rules can be chosen at a merge point. This strategy is well-suited to the breadth first parsing approach of the GLR algorithm.

As our work was concerned with at least matching the operation of the btyacc generated parser, both approaches have been combined in *JavaCup*$^{+}$. This allows the functionality of btyacc to be utilised alongside the extra features of the GLR algorithm. Special commit actions were added to the generated parser that

replicated some of the commit actions of btyacc. These performed analogously to btyacc, with the first branch to encounter a commit point during a breadth-first parse being chosen as the correct parse. The second strategy was used at select merge points; if two branches with identical subtrees created from different rules are to be merged, then the rule that was declared first in the grammar was chosen. Thus the approach to determinisation in *JavaCup*$^+$ can be considered a hybrid of btyacc and traditional GLR approaches.

6.1.2 Default reduces in *JavaCup*$^+$

As *JavaCup*$^+$ is a direct port of the LALR(1) parsing algorithm, the behaviour of the algorithm in detecting whether any input is valid or not is identical across *JavaCup*$^+$, btyacc and bison. However the external behaviour belies some subtle internal differences between *JavaCup*$^+$ and btyacc.

Recall from Section 2.1.2 that a parser consults an action table using the state at the top of current stack and the current symbol as indices to retrieve the next action. This is the default action for *JavaCup*$^+$. However as an optimisation, the yacc family, of which bison and btyacc are members, allows a state to contain *default* reduce action. That is, if an item in a state contains a single reduction with *n* symbols in the potential lookahead then the entries in that state under the lookahead symbols will all refer to this reduction. In cases like this it is possible to say that a state has a guaranteed reduce associated with it, without checking the current input symbol.

This behaviour has the net effect of allowing a reduce to take place *before* the next token is read from the Token Decorator. As the semantic action is associated with the reduce, it is possible to have that semantic action *alter* the symbol table. Thus if the symbol table has been updated then the Token Decorator may return a marked up version of an identifier instead of a plain identifier if this is the next token. This small but significant difference could have resulted in the failure of a valid parse in jKeystone. jKeystone will read the token then perform the action. If the token read was an identifier, then the execution of the action could potentially

mark up the identifier to one of the context-sensitive identifiers.

This subtle difference was handled by emulating the check for a default reduce action in the runtime code of the GLR parser. This was achieved by placing a check for a default reduction at the point before the next token is read. All of the actions in the current state are checked to see if they are all identical reduce actions. If this is the case, the current state is said to have a default reduction and the reduce is performed without the call to the next token being made. If there is no default reduction then the parse continues as normal, with the next token being read and the parse action being determined by a lookup in the action table using the current state and current token.

6.2 Testing jKeystone

The testing of the completed jKeystone system was a crucial phase of the implementation. The testing strategy involved the use of the T_{ddj} suite to measure the progression towards full implementation correctness. Once the T_{ddj} suite reached full acceptance, two other test-suites were used to augment the system testing effort.

6.2.1 Test-driven integration of the jKeystone front and back-end

Once a method for enhancing the GLR algorithm with semantic actions was implemented, the generated parser could be plugged into the ported keystone back-end. As with any software-engineering project the testing of the completed system with the GLR front-end was of paramount importance. As the back-end had been successfully ported and tested, it could be treated as a separate component free of bugs. Thus all of the bug-fixes took place in the front-end's runtime code with the overall goal of ensuring that the functionality of keystone was replicated.

As the T_{ddj} suite was used primarily to measure the progression towards a fully

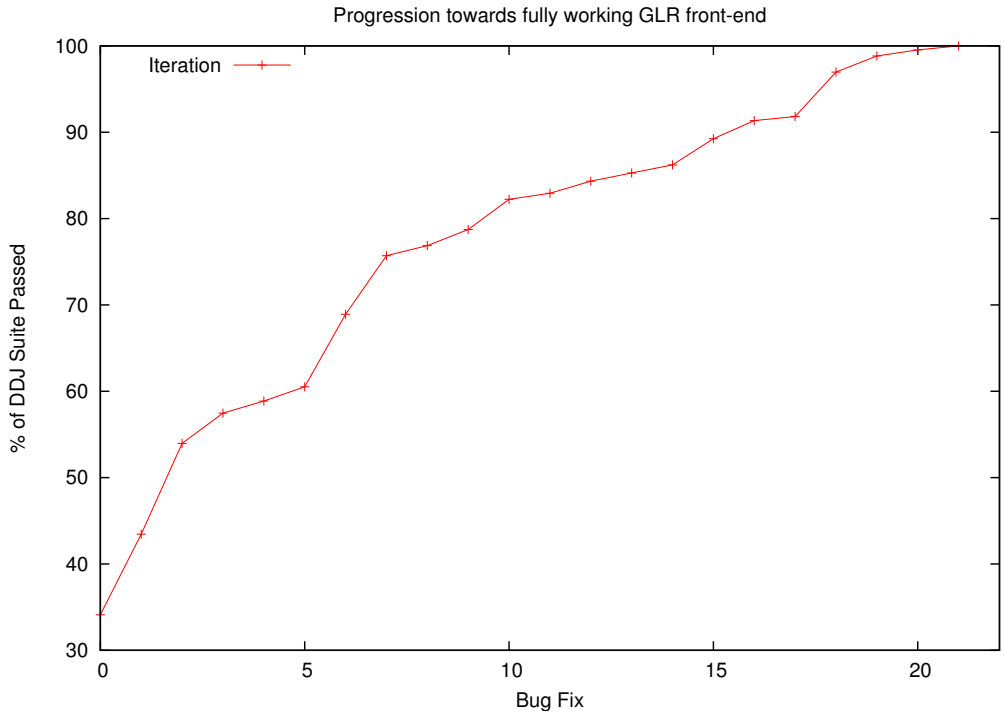


Figure 6.3: The progression towards a working front-end. *This graph shows the percentage of working test-cases in the T_{ddj} suite through each bug-fix. In total there were 21 iterations required, mostly related to fixing the semantic actions to work with GLR.*

---

operational jKeystone, the percentage of the T_{ddj} suite passed was measured with each bug-fix made to the front-end of the system. This is shown in Figure 6.3. As can be seen from this figure, there were 21 iterations in total. 7 of these bugs were as a result of small bugs related to the definition of non-terminal types within the specification of the grammar, and the other 14 were related to the implementation of the semantic actions in the GLR algorithm.

Test-Suite	# of Tests	Passed keystone	Passed jKeystone
T_{ddj}	447	428	436
T_{gcc}	1825	1451	1474
T_{Stress}	10	9	9

Table 6.1: The Test-Suites used to test jKeystone. *For each of the three test suites, this table shows the number of individual test cases, the number that were passed by* keystone*, and the number that were passed by* jKeystone.

6.2.2 System testing jKeystone

Once jKeystone had matched keystone's operation over the T_{ddj} suite, it was then tested with the T_{gcc} suite. The summary of the test-cases accepted by jKeystone is displayed in Table 6.1. As can be seen from this table, for both the T_{ddj} and T_{gcc} suites jKeystone passes more of the test-cases than keystone, with 8 more test-cases passed in the T_{ddj} suite and 23 more passed in the T_{gcc} suite.

In addition to the two test-suites used in the previous chapters, we have stress tested the parser with a number of large real-world programs which we have labelled T_{Stress} in Table 6.1. This test-suite has been used to test keystone's feasibility and robustness [Gib03]. The successful parsing of this test-suite by jKeystone would give another indication of the robustness of jKeystone. The programs contained in the T_{Stress} suite are:

- The test-case ADOL-C facilitates the evaluation of first and higher derivatives of vector functions that are defined by computer programs written in C or C++.
- Test case encrypt is an encryption program that uses the Vignere algorithm.
- The ep-matrix test case is an extended precision matrix application that uses NTL, a high performance portable C++ number theory library.
- php2cpp converts the PHP web publishing language to C++.
- The test case fft performs fast Fourier transforms.
- graphdraw is a drawing application that uses IV Tools, a suite of free XWindows drawing editors for Postscript, TeX and web graphics production.

- vkey125 is a GUI application that uses the V GUI library, a multi-platform C++ graphical interface framework to facilitate construction of GUI applications.

Each of these programs range in size from 1,000 to 14,000 lines of code and have a minimum of 3,000 tokens up to 30,000 tokens. As shown in Table 6.1, jKeystone performs identically to keystone for the T_{Stress} testsuite in parsing 9 out of the 10 test-cases. These results demonstrate the robustness of jKeystone in terms of the original implementation.

6.3 Comparing the Systems

As the previous section has demonstrated, the testing phase showed that jKeystone was accepting more test-cases than keystone. In this section we aim to give a side-by-side comparison of the two systems with respect to timings and parsing algorithms.

6.3.1 Timings

The graph in Figure 6.4 shows the timings of the two systems. The timings were performed on a *Dell* Optiplex GX280 PC, with a 3.06Ghz Intel processor, 1Gb DDR RAM running the Fedora Core 4 distribution of GNU/Linux. The bottom line represents the timings for keystone while the top two lines show the timings for jKeystone with and without the startup cost for the JVM respectively. As can be seen from the graph, jKeystone is slower than original system by a factor of about 20. This degradation can partly be attributed to the fact that the GLR runtime code is not optimised. For example the code to check for a default reduction must iterate over the action table for every state. In btyacc, the default reduce is a lookup table indexed by state. This functionality has yet to be implemented in the Java version. Furthermore, the indexing of the conflicts within the conflict table is less than optimal, as at present a linear search has to be performed each and every

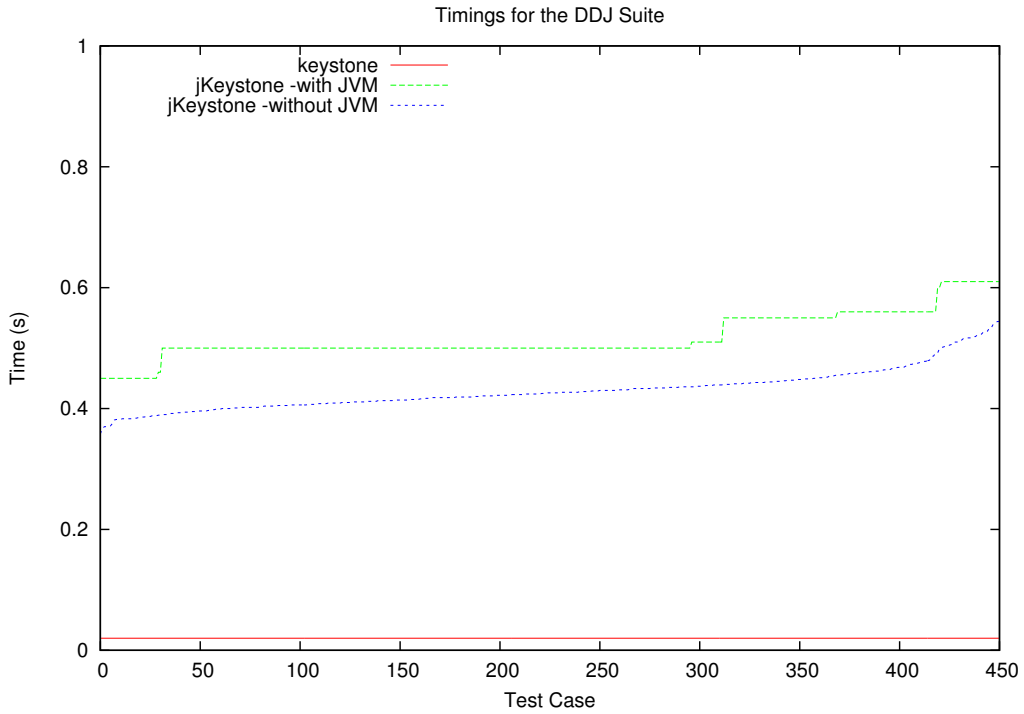


Figure 6.4: The timings for the completed jKeystone versus the original C++ version for the T_{ddj} suit. *The timings given show not only the comparison of the C++ and Java versions but also the difference in timings when the JVM start up costs are taken into account. In the figure above, the bottom line shows* keystone *remaining at an almost constant 20 ms. The Java version displays timings in the 400 ms range for each of the test-cases without the JVM start-up and in the 600ms range with JVM start-up accounted for.*

---

time a conflict is detected. It should be noted that since jKeystone is a prototype test-bed where correctness is paramount, the speed of the generated parsers was not as important during its development.

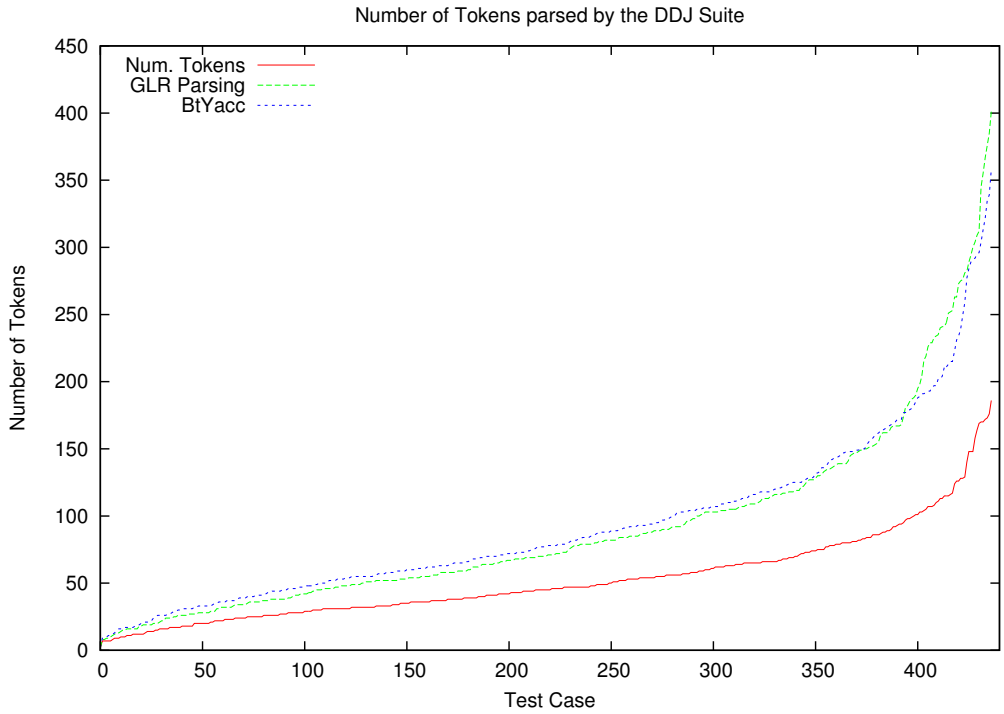


Figure 6.5: Comparison of GLR parsing versus btyacc. *The total number of tokens per test-case is displayed by the base line while the green and blue lines represent the number of tokens seen by the GLR and* btyacc *parsing algorithms respectively. As can be seen from the graph, the GLR algorithm needs to use less tokens then the* btyacc *algorithm for test-cases with less then 100 tokens. At around the 200 token level, the GLR algorithm starts to use more tokens during a parse than the* btyacc *version but only by a small number each time.*

6.3.2 Comparison of the Algorithms

In order to evaluate the underlying parsing algorithms behind the two generated parsers, a complexity-based comparison was made. The standard measure of the complexity of a parser is in terms of the number of times each token is processed. Both the btyacc-generated parser and the GLR-based parser effectively perform trial parses, and hence must process sections of the input more then once. Thus, the number of times a token needs to be read can be used to give a comparative

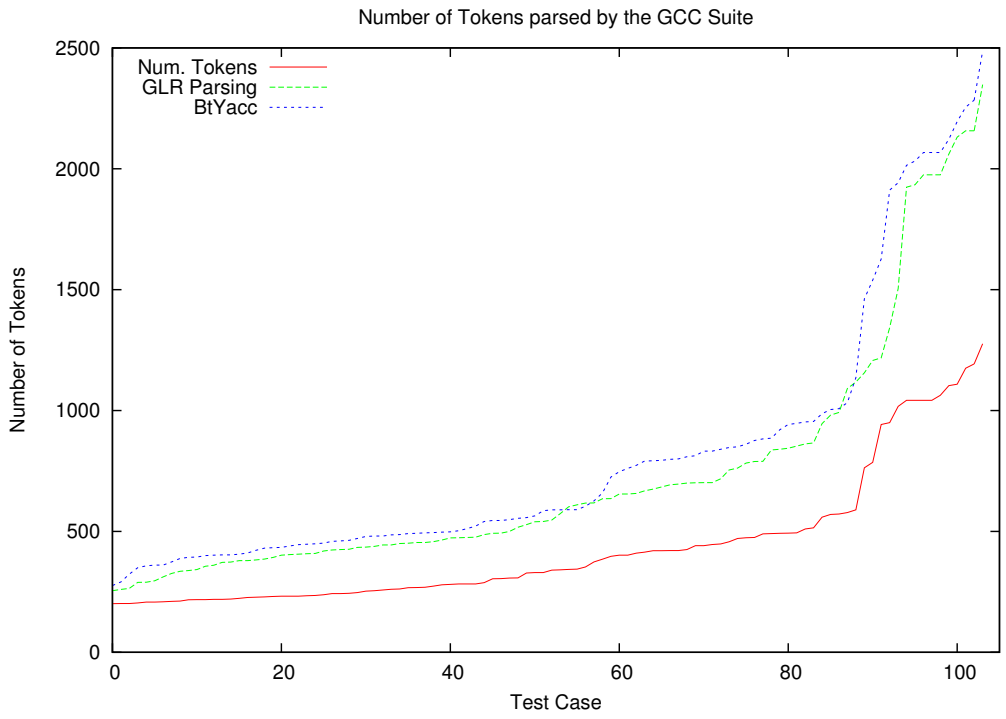


Figure 6.6: Comparison of GLR parsing versus btyacc. *The result of parsing the largest 100 tests from the T_{gcc} suite are displayed. Each test-case contains at least 200 tokens. The behaviour is similar to that displayed in Figure 6.5 in that the GLR parsing requires the re-parsing of less tokens in general than the* btyacc*, backtracking parsing strategy.*

---

evaluation of the parsing algorithms that is independent of the implementation language.

The graph in Figure 6.5 shows the number of tokens parsed by both of the systems for each test-case in the T_{ddj} suite. The base line represents the number of tokens in a given test-case. For a fully deterministic parse, each token would only need to be seen once. In the case of a simple GLR parse the tokens would only be seen once, but the need for semantic actions means that the parse must be replayed once a choice has been made as to which rule to use. Consequently, the tokens from the point of the conflict onward must be used again, this time to parse

after the conflict has been resolved.

The green line in Figure 6.5 represents the number of tokens used by the GLR parse while the top blue line shows the number of tokens used by btyacc. As can be seen from the graph, the GLR algorithm performs marginally better than the btyacc algorithm as it requires the re-parsing of less tokens in general than btyacc. There are localised test-cases where btyacc performs better but the general trend shows the GLR mode having an advantage. In Figure 6.6 the result of parsing the 100 test-cases containing 200 or more tokens from the T_{gcc} test-suite are shown. The results reinforce those shown in Figure 6.5 as once more the GLR algorithm requires less tokens to be re-parsed than btyacc.

The results shown in Figures 6.5 and 6.6 demonstrate that the time penalty measured in section 6.3.1 is not inherent in the move from btyacc to GLR. Indeed, we hope that as we continue to optimise the code generated by *JavaCup*$^{+}$, its speed will eventually match and even outstrip that of btyacc-generated parsers.

6.4 Synopsis

In this chapter we have outlined our strategy for creating a fully functional parser front-end for jKeystone based upon the GLR parsing algorithm. The construction of the front-end and the integration of this front-end into the ported back-end saw the completion of the whole jKeystone system. This allowed for the side-by-side comparison of the C++ and Java systems. The results of this testing showed the feasibility of using GLR in the construction of a parser front-end. Indeed the comparison of GLR and btyacc indicate that GLR performs better than back-tracking when applied to the ISO C++ grammar.

The completed jKeystone system was able to accept *more* test-cases than the original keystone system thus offering a further validation of the GLR algorithm. This was emphasised by the acceptance of *more* test-cases from the T_{ddj} and T_{gcc} suites than keystone had managed.

By providing the code-base in Java, with its features such as automatic garbage

collection, the overall system has become more stable and robust. This behaviour manifests itself when trying to analyse two large programs in batch mode, where keystone does not re-initialise successfully after analysing the first program. The behaviour of keystone, which has a manual memory management scheme is to crash with a segmentation fault. jKeystone on the other hand is capable of continuing in batch mode with no such error. Thus while there is currently a time overhead with jKeystone, its acceptance of more test-cases and its stability while parsing in batch mode make it a significant improvement on the original version.

Chapter 7

Concluding Remarks

In this thesis we have described a test-driven strategy for the development of a working GLR parser front-end for jKeystone, a Java port of the original keystone system. We have also investigated the idea of creating a small standardised test-suite for applications that receive ISO C++ as input. We have measured the effectiveness of these minimised test-suites with respect to code coverage and fault detection. We have also defined a strategy to port software from one object-oriented system to another via object-relation diagrams. In chapter 5 we described the creation of a GLR parser generator for Java and chapter 6 outlined the unification of a generated GLR parser with the ported keystone back-end to create jKeystone.

7.1 Main Findings

There are four main findings deriving from this work:

1. The investigation into whether grammar rule-coverage is an effective criterion when reducing test-suites for grammar-based software is a novel contribution of this thesis. This work seeks to discover if there is a relationship between coverage of the front-end and the back-end based solely on coverage of the front-end and is the first study of its kind. We believe that the study on test-suite minimisation can conclusively state that rule-coverage

as a reduction criterion is ineffective, and thus should not be considered in future.

2. The definition of an order for porting via ORDs is another novel contribution. While ORDs are typically used to define an order for inter-class testing, this thesis has shown that they can be applied to the porting of one object-oriented language to another.

3. The creation of a working GLR parser generator is a significant contribution to the field. To our knowledge there is no other GLR parser generator for Java. The GLR algorithm has been established for 20 years, yet there is still a dearth of tools in the mainstream software development community that utilise the algorithm. Our GLR parser generator, *JavaCup*$^+$, is easy to use and is capable of handling difficult grammars such as ISO C++, C$^\#$ and COBOL, thus showing its robustness. *JavaCup*$^+$ has been well tested and one of its generated parsers for ISO C++ has been successfully deployed and validated in a large scale software-engineering project.

4. We have described an approach to the addition of working semantic actions to *JavaCup*$^+$ that enable useful parsers to be constructed. This approach synthesised features of both btyacc and bison's GLR mode. We have demonstrated this with a fully functional C++ parser front-end. We have tested this by making it the parser front-end for jKeystone, the ported version of keystone.

   The fact that jKeystone outperforms keystone in terms of number of test-cases passed is not only a validation of the GLR algorithm but it also validates the original design goal laid down by the developers of keystone who employed the facade design pattern to enable any other parser to be deployed as the front-end.

7.2 Future Work

The trend established by the ISO C++ grammar has been continued by the recent definition of the C# grammar. Both grammars reinforce a move away from a grammar that is compatible with the yacc family of tools. Consequently it is desirable to develop tools that allow these grammars to be parsed without extensive re-writing of the grammar itself. We feel that there will be a growing movement towards GLR-based parsers in the coming years and our work shows that the algorithm is practical for many of the popular programming languages.

Future work emanating from the research conducted in this thesis should include:

1. A study into the effectiveness of *context-dependent* and *level-n* coverage of grammars [RL01]. These extra coverage criteria involve the generation of test cases that not only cover every grammar rule, but every valid grammar rule pair in the case of context-dependent coverage and every valid rule chain of length *n* in the case of level-n coverage. Test-cases exhibiting these kinds of behaviours would more than likely have to be synthesised using a technique similar to Purdom's algorithm. Developing test-cases in this manner that are also semantically correct would be a huge challenge.

2. A re-engineering of the jKeystone back-end to bring it up to the level of acceptance of gcc for all programs would be desirable. gcc is currently the *de facto* public domain compiler for C++ and acceptance by jKeystone of all the programs accepted by gcc could lead to widespread use of jKeystone within the program comprehension community.

3. There are a number of optimisations to be made to the GLR parser generator and the runtime code. The conflict table is currently a linear table which must be searched in time O(n) each time a conflict is encountered. One possible optimisation would be to re-factor this so that the conflict table is indexed by state. Furthermore all conflicts are stored within the table

and duplicates are only removed at runtime. By transferring this stage to parser generation time, another speed up could be achieved. Finally, another possible speed up is to create a table to check for a default reduction within a state. Currently a check for a default reduction takes at worst case O(n) with a linear search through the action-table for every state at the top of the GSS. If a new table was created that was indexed by state, this time penalty would be overcome.

Our work in this thesis has demonstrated the practicality and feasibility of GLR, and has opened up new avenues of development for jKeystone. This work has also continued the synthesis between parsing and software-engineering, reflected in the use of software-engineering techniques for parser development as well as the use of of parsers in reverse-engineering and program comprehension tools.

Bibliography

[Aco00] Dragos Acostachioaie. Doc++: Open source - open science - open systems. *Circles Electronic Magazine*, (36), 2000.

[ADV04] Robert Anisko, Valentin David, and Clément Vasseur. Transformers: a C++ transformation framework. Computer Science Technical Report 0310, LRDE EPITA, Le Kremlin Bicêtre cedex, France, May 2004.

[AHJM01] John Aycock, Nigel Horspool, Jan Janousek, and Borivoj Melichar. Even faster GLR parsing. *Acta Informatica*, 37(9):633–651, 2001.

[Ana06] CodePro AnalytiX. http://www.instantiations.com/codepro/index.html, Last accessed January 15, 2007, 2006.

[Ast03] Dave Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall, 2003.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Bec00] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 2000.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002.

[BG98] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, July 1998.

[Bin99] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, Boston, MA, USA, 1999.

[BLL05] L.C. Briand, Y. Labiche, and J. Leduc. Tracing distributed systems executions using AspectJ. In *21st International Conference on Software Maintenance*, pages 81–90, Budapest, September 2005.

[BLM03] L. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. In *10th Working Conference on Reverse Engineering*, pages 57–66, Victoria, BC, Canada, November 2003.

[BLW01] L. Briand, Y. Labiche, and Y. Wang. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. In *12th International Symposium on Software Reliability Engineering*, pages 287–296, Hong Kong, November 2001.

[BPM04] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program transformations for practical scalable software evolution. In *26th International Conference on Software Engineering*, pages 625–634, Edinburgh, Scotland, 2004.

[BS82] F. Bazzichi and I. Spadafora. An automatic generator for compiler testing. *IEEE Transactions on Software Engineering*, 8(4):343–353, 1982.

[CCRV+80] Augusto Celentano, Stefano Crespi-Reghizzi, Pierluigi Della Vigna, Carlo Ghezzi, G. Granata, and F. Savoretti. Compiler testing using a sentence generator. *Software: Practice and Experience*, 10(11):897–918, November 1980.

[Chi95] Shigeru Chiba. A metaobject protocol for C+. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, Austin, Texas, United States, 1995.

[Dij70] Edsger W. Dijkstra. Notes on structured programming. Computer science technical report, Technological University Eindhoven, 1970.

[dJKV99] Merijn de Jonge, Tobias Kuipers, and Joost Visser. HASDF: a generalized LR-parser generator for Haskell. Computer Science

Technical Report SEN-R9902, Centrum voor Wiskunde en Informatica (CWI), 1999.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayword. Hints on test data selection. *IEEE Computer*, 11(4):34–41, April 1978.

[Ear70] J. Earley. An efficient context free parsing algorithm. *Communications of the ACM*, 13, 1970.

[EDG] Edison Design Group. http://www.edg.com, Last accessed January 15, 2007.

[Egg03] Paul Eggert. Bison 1.875 is now available for download. comp.compilers newsgroup, 2003.

[Gib03] Tanton H. Gibbs. *The design and implementation of a parser and front-end for the ISO C++ lanaguage and the validation of that parser*. PhD thesis, Clemson University, 2003.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[GJ90] D. Grune and C. J. H. Jacobs. *Parsing Techniques: a practical guide*. Ellis Horwood, 1990.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2005. Third Edition.

[GMP03a] T. H. Gibbs, B. A. Malloy, and J. F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software: Practice and Experience*, 33(1):19–39, January 2003.

[GMP03b] T. H. Gibbs, B. A. Malloy, and J. F. Power. Progression toward conformance of C++ language compilers. *Dr. Dobbs Journal*, 28(11):54–60, September 2003.

[GNU06] GNU. Gnu bison. http://www.gnu.org/software/bison/manual, Last accessed January 15, 2007, 2006.

[HG04] Mats Per Erik Heimdahl and Devaraj George. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *19th IEEE International Conference on Automated Software Engineering*, pages 176–185, Linz, Austria, 20-25 September 2004.

[HGS93] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[Hir77] Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.

[HL00] J. Harm and R. Lämmel. Two-dimensional Approximation Coverage. *Informatica*, 24(3), 2000.

[HP05a] Mark Hennessy and James F. Power. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *20th IEEE International Conference on Automated Software Engineering*, pages 104–113, Long Beach, CA, USA, November 2005.

[HP05b] Mark Hennessy and James F. Power. Generation strategies for test-suites of grammar-based software. Technical Report NUIM-CS-TR-2005-02, Department of Computer Science, National Univesity of Ireland, Maynooth, April 13 2005.

[HP06] Mark Hennessy and James F. Power. Ensuring behavioural equivalence in test-driven porting. In *CASCON '06: Dublin Symposium*, IBM Campus, Dublin, Ireland, 17th October 2006.

[HPM03] M. Hennessy, J. F. Power, and B. A. Malloy. gccxfront:exploiting gcc as a front end for program comprehension via XML/XSL. Portland, OR., USA, May 9 - 11 2003.

[Hud97] Scott E. Hudson. Java based constructor of useful parsers. Technical report, Graphics Visualisation and Usibility Centre, Georgia Institute of Technology, 1997. http://www.cs.princeton.edu/ appel/modern/java/CUP/.

[HY05] Daniel Hoffman and Kevin Yoo. Blowtorch: a framework for firewall test automation. In *20th IEEE International Conference on Automated Software Engineering*, pages 96–103, Long Beach, CA, USA, November 2005.

[ISO83] ISO/IEC JTC1. *International Standard: Programming Languages - Pascal*. Number ISO/IEC 7185:1983. American National Standards Institute, first edition, 1983.

[ISO90] ISO/IEC JTC1. *International Standard: Programming Languages - C*. Number ISO/IEC 9899:1990. American National Standards Institute, first edition, 1990.

[ISO98] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E). American National Standards Institute, first edition, September 1998.

[ISO03] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:2003(E). American National Standards Institute, second edition, October 15 2003.

[ISO06] ISO/IEC JTC 1. *International Standard: Programming Languages - C#*. Number 23270:2006. American National Standards Institute, second edition, June 2006.

[Jav] JavaCC. https://javacc.dev.java.net/, Last accessed January 15, 2007.

[JCMCJM63] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, 1963.

[JH03] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–210, 2003.

[JL97] John Lilley. John Lilley's pccts-based LL(1) C++ parser. http://www.empathy.com/pccts, 1997.

[Joh75] S. C. Johnson. Yacc – yet another compiler compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, USA, 1975.

[JS03] Adrian Johnstone and Elizabeth Scott. Generalised regular parsing. In *12th International Conference on Compiler Construction*, pages 232–246, Berlin, 2003.

[JSE04] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. The Grammar Tool Box: A Case Study Comparing GLR Algorithms. In *4th Workshop on Language Descriptions, Tools and Applications*, Barcelona, April 2004.

[KGH+93] C. H. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima. Design recovery for software testing of object-oriented programs. In *1st Working Conference on Reverse Engineering*, pages 202–211, Baltimore, MD, USA, May 1993.

[KGH+95] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. A test strategy for object-oriented systems. In *Computer Software and Applications Conference*, pages 239 – 244, Dallas TX., August 1995.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[KLDM99] G. Knapen, B. Laguë, M. Dagenais, and E. Merlo. Parsing C++ despite missing declarations. In *7th International Workshop on Program Comprehension*, pages 114–125, Pittsburgh, PA, 1999.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *European Conference on Object-Oriented Program*, pages 220–242, Jyväskylä, Finland, 1997.

[KLV05] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[Knu65] D. E. Knuth. On the Translation of Languages from left to right. *Information and Control*, 8(6):607–639, October 1965.

[Kop97] Rainer Koppler. A systematic approach to fuzzy parsing. *Software: Practice and Experience*, 27(6):637–649, 1997.

[Lav96] Alan Lavie. *A Robust, Grammar-focused Parser for spontaneously spoken language*. PhD thesis, Carnegie-Mellon University, 1996.

[Mar03] R. C. Martin. *Agile Software Development*. Prentice Hall, 2003.

[MCL03] Brian A. Malloy, Peter J. Clarke, and Errol L. Lloyd. A parameterized cost model to order classes for class-based testing of C++ applications. In *14th International Symposium on Software Reliability Engineering*, pages 353–364, Denver, CO., November 2003.

[McP02] S. McPeak. Elkhound: A fast, practical GLR Parser Generator. Computer Science Technical Report UCB/CSD-2-1214, UC Berkeley, University of Berkeley, CA, USA, 2002.

[Mem02] A.M. Memon. GUI testing: Pitfalls and process. *IEEE Computer*, 35(8):87–88, August 2002.

[MLDP02] Brian A. Malloy, Scott A. Linde, Edward B. Duffy, and James F. Power. Testing C++ compilers for ISO language conformance. *Dr. Dobbs Journal*, 27(6):71–78, June 2002.

[MOK05] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[MP01] B. A. Malloy and J. F. Power. An interpretation of Purdom's algorithm for automatic generation of test cases. In *1st Annual International Conference on Computer and Information Science*, Orlando, FL., 2001.

[MP05] Brian A. Malloy and James F. Power. Exploiting UML dynamic object modeling for the visualization of C++ programs. In *ACM Symposium on Software Visualisation*, pages 105 – 114, St. Louis, MO, USA, May 2005.

[MPW02] B. A. Malloy, J. F. Power, and J. T. Waldron. Applying software engineering techniques to parser design. In *Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 75–82, Port Elizabeth, South Africa, September 16-18 2002.

[MRR02] A. Milanova, A. Rountev, and B.G. Ryder. Constructing precise object relation diagrams. In *International Conference on Software Maintenance*, pages 586–595, Montreal, Canada, September 2002.

[NF91] R. Nozohoor-Farshi. GLR parsing for ε-grammars. In *Masaru Tomita (Ed.), Generalized LR Parsing*, pages 60 – 75, Amsterdam, the Netherlands, 1991.

[Nta88] S.C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, 1988.

[OLR+96] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, April 1996.

[PHts] Plum Hall test suites. http://www.plumhall.com/, Last accessed January 15, 2007.

[PM00] J. F. Power and B. A. Malloy. Symbol table construction and name lookup in ISO C++. In *Technology of Object-Oriented Languages and Systems*, pages 57–68, Sydney, Australia, November 2000.

[PM01] James F. Power and Brian A. Malloy. Exploiting metrics to facilitate grammar transformation into LALR format. In *ACM Symposium on Applied computing*, pages 636–640, Las Vegas, Nevada, United States, 2001.

[PM02] J. F. Power and B. A. Malloy. Program annotation in XML: a parser-based approach. In *Working Conference on Reverse Engineering*, pages 190–198, Virginia, USA, October 28 - November 1 2002.

[PM04] James F. Power and Brian A. Malloy. A metrics suite for grammar-based software. *Software Maintenance and Evolution: Research and Practice*, 16(6):405–426, Nov/Dec 2004.

[PQ95] T.J. Parr and R.W. Quong. Antlr: A predicated-LL(k) parser generator. *Software - Practice and Experience*, 25(7):789 – 810, July 1995.

[PtsfIC++] Perennial test-suite for ISO C++. http://www.peren.com/pages/cppvs.htm, Last accessed January 15, 2007.

[PUM] Puma - The PURE Manipulator. http://ivs.cs.uni-magdeburg.de/ puma/, Last accessed January 15, 2007.

[Pur72] P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3):366–375, 1972.

[RAMT02] Ferenc R, Beszedes A, Tarkiainen M, and Gyimothy T. Columbus - reverse engineering tool and schema for C++,. In *International Conference on Software Maintenance*, pages 172–181, Montreal, Canada, 2002.

[RC05] Atanas Rountev and Beth Harkness Connell. Object naming analysis for reverse-engineered sequence diagrams. In *27th International Conference on Software Engineering*, pages 254–263, St. Louis, MO, USA, May 2005.

[RD95] S. Reiss and T. Davis. Experiences writing object-oriented compiler front-ends. Computer science technical report, Brown University, 1995.

[Rek92] Jan Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[RHOH98] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *International Conference on Software Maintenance*, pages 34–43, Maryland, USA, November 16-19 1998.

[RL01] Ralf Lämmel. Grammar testing. In *Fundamental Approaches to Software Engineering*, volume 2029 of *LNCS*, pages 201–216. Springer Verlag, 2001.

[RL03] Ralf Lämmel. http://www.cs.vu.nl/grammars/vs-cobol-ii, Last accessed January 15, 2007, 2003.

[Rop94] M. Roper. *Software Testing*. McGraw-Hill, 1994.

[Ros89] Jim Roskind. A YACC-able C++ 2.1 grammar, and the resulting ambiguities. www.empathy.com/pccts/roskind.html, 1989.

[Sci06] Scitools. Understand for C++. http://www.scitools.com/products/understand/cpp/product.php, Last accessed January 15, 2007, 2006.

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *40th International Conference on Technology of Object-Oriented Languages and Systems*, pages 53–60, Sydney, Australia, February 18-21 2002.

[SHE02] Susan Elliott Sim, Richard C. Holt, and Steve Easterbrook. On using a benchmark to evaluate C++ extractors. In *10th International Workshop on Program Comprehension*, pages 114–126, Paris, France, June 2002.

[Sib03] Siber. Btyacc: Backtracking yacc. http://www.siber.com/btyacc/, Last accessed January 15, 2007, 2003.

[SJ06] Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems*, 28(4):577–618, 2006.

[SLU05] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. AspectC++: an AOP extension for C++. *Software Developer's Journal*, pages 68–76, May 2005.

[Sof93] Software Engineering Standards Committee of the IEEE. *IEEE Standard Classification for Software Anomolies*. IEEE Standards Board, 1993.

[SR92] M. D. Smith and D. J. Robson. A framework for testing object-oriented programs. *J. Object Oriented Program.*, 5(3):45–53, 1992.

[TD97] Kuo-Chung Tai and F.J. Daniels. Test order for inter-class integration testing of object-oriented software. In *Computer Software and Applications Conference*, pages 602–607, Washington, DC, USA, August 1997.

[Tec06] MSquared Technologies. Resource standard metrics for C++. http://msquaredtechnologies.com/m2rsm/, Last accessed January 15, 2007, 2006.

[Tom85] Masaru Tomita. *Efficient Parsing for Natural Langauges*. Kluwer Academic Publishers, 1985.

[TOMG03] The Object Management Group. The Unified Modelling Language Version 1.5 OMG. Formal/2003-03-01, March, 2003.

[VAPT05] Pradeep Varma, Ashok Anand, Donald P. Pazel, and Beth R. Tibbitts. Nextgen extreme porting: structured by automation. In *ACM Symposium on Applied Computing*, pages 1511–1517, Santa Fe, New Mexico, March 2005.

[vdBvDH+01] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: a component-based language development environment. In *Compiler Construction*, pages 365–370, Genova, Italy, April 2-6 2001.

[vH06] Dimitri van Heesch. Doxygen - a Source Code documentation generator tool. http://www.stack.nl/ dimitri/doxygen/index.html, Last accessed January 15, 2007, 2006.

[WG97] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *Conference on Programming Language Design and Implementation*, pages 31–43, Las Vegas, Nevada, USA, 15-18 June 1997.

[WHLM98] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience*, 28(4):347–369, April 10 1998.

[Wil01] Edward Willink. *Meta-Compilation for C++*. PhD thesis, University of Surrey, 2001.

[Win01] A. Winter. GXL: Graph eXchange language. In *Dagstuhl Seminar Interoperability of Reengineering Tools*, Dagstuhl, Saarland, Germany, January 2001.

Appendix A

The ORD for keystone

The ORD for keystone is displayed in Figure A.1. The numbered nodes in the ORD correspond to the following keystone classes:

	Class	Weight	Stubs
1	Ast	0	None
2	Visitor	0	None
3	LocationTracker	0	None
4	KeywordManager	0	None
5	TokenPosn	0	None
6	TokenInfo	15	None
7	NameDeclaration	30	None
8	ActionsHelp	35	None
9	ArgumentStack	35	None
10	TypeInvariantFacilitator	70	Type
11	ScopeInvarinatFacilitator	70	Scope
12	NameDeclarationInvariantFacilitator	70	None
13	Scope	85	None
14	Type	85	None
15	ClassType	100	None
16	EnumType	100	None
17	IndirectType	100	None
18	FunctionType	100	None
19	TemplateTemplateParameterType	100	None
20	BasicType	100	None

21	NamespaceType	100	None
22	TemplateParameterType	100	None
23	NameOccurrence	100	None
24	NamespaceScope	100	None
25	PrototypeScope	100	None
26	TemplateParameterScope	100	None
27	DeclarationStack	105	None
28	TokenBuffer	105	ActionFacade
29	ContextManager	105	None
30	FunctionScope	135	None
31	ClassScope	135	None
32	LocalScope	135	None
33	TokenDecorator	155	ActionFacade
34	LookupController	175	None
35	ActionFacade	175	None
36	TypeInvariantVisitor	225	None
37	Actions	245	None
38	NameDeclarationInvariantVisitor	295	None
39	ScopeInvariantVisitor	330	None
40	Parser	400	None

Table A.1: The porting order for the keystone classes. *The number indicates the order in which the class was ported which was determined by calculating the weight from the ORD. In this case, only 3 distinct class stubs were needed during the test-driven porting of* keystone.

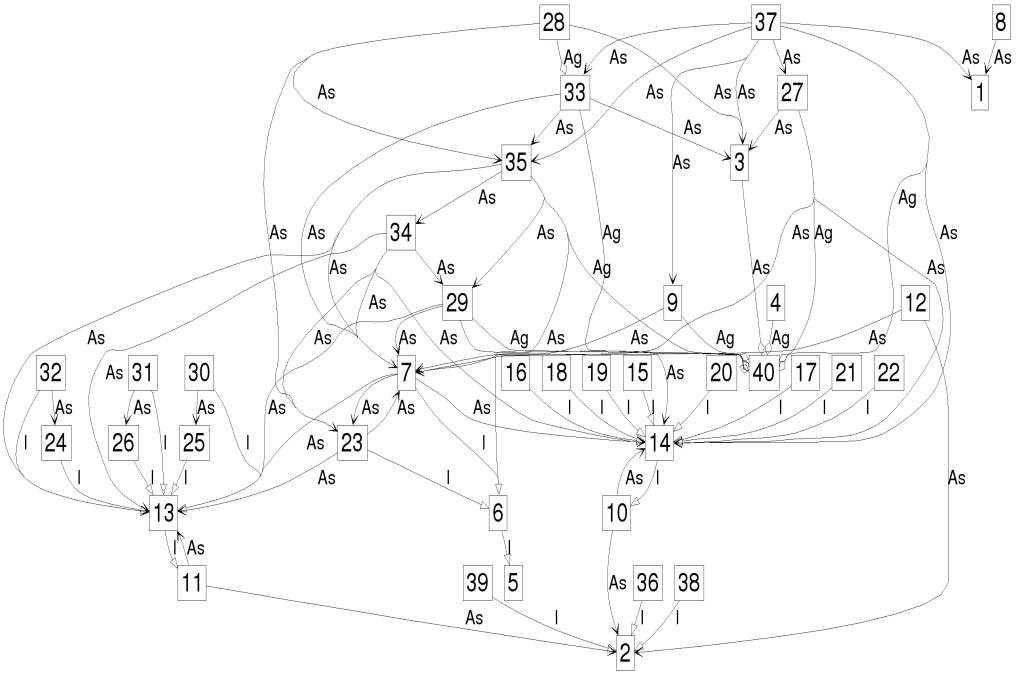


Figure A.1: The Object Relation Diagram for keystone. *In this diagram the nodes represent classes and the edges represent inheritance (I), association (As) and aggregation (Ag) relationships between the classes. The number at each node indicates the porting order, with 1 indicating the first class to be ported.*