# Language Constructs for C++-like languages
## Tools and extensions

Valentin David

Dissertation for the degree Philosophiae Doctor (PhD) at the
University of Bergen

September 7, 2009

Preprint.
Papers reprinted by permission from ACM and IEEE. All rights reserved.
Other content © 2009 by Valentin David.

*To myself, because I am an arrogant Frenchman.*

# Contents

**Preface** **xi**

**Abstract** **xiii**

**1 Introduction** **1**
1.1 Motivation . . . . . 2
1.1.1 Extensions: success or failure . . . . . 4
1.1.2 The transformations presented in this dissertation . . . . . 5
1.2 Writing tools . . . . . 5
1.3 Program transformation . . . . . 6
1.4 Language specification . . . . . 7
1.4.1 Syntax . . . . . 8
1.4.2 Semantics . . . . . 8
1.5 C++ . . . . . 9
1.5.1 Context-sensitive parsing . . . . . 9
1.6 ConceptC++ . . . . . 11
1.7 Overview . . . . . 13

**2 Program transformation and C++** **15**
2.1 Introduction . . . . . 15
2.2 C++ language's properties . . . . . 16
2.2.1 The pre-processor . . . . . 16
2.2.2 C++ is context-sensitive . . . . . 17
2.2.3 The context is locally not left-to-right . . . . . 17
2.2.4 The context is sometimes incomplete . . . . . 18
2.2.5 The context depends on template instantiation . . . . . 19
2.2.6 The context needs type inference . . . . . 20
2.2.7 The template instantiation is architecture-sensitive . . . . . 20
2.3 Front-ends . . . . . 21

2.3.1 Needs . . . 21
2.3.2 Determinants . . . 22
2.3.3 Yacc type . . . 23
2.3.4 Hand written recursive-descent . . . 25
2.3.5 Return of parser generators . . . 29
2.3.6 AG and disambiguation . . . 30
2.4 Internal representations and environments . . . 31
2.4.1 General representations . . . 31
2.4.2 C++-specific representations . . . 32
2.5 Conclusion . . . 33

**3 Modular specification for a context-sensitive syntax 35**
Introduction . . . 37
3.1 Context . . . 38
3.2 Disambiguation chain . . . 40
3.2.1 Upstream . . . 40
3.2.2 Attributes . . . 41
3.2.3 Downstream . . . 42
3.3 Attribute Grammars in SDF . . . 43
3.3.1 Attribute Rules in Syntax Definition Formalism (SDF) . . . 43
3.3.2 Grammar Modules Preprocessing . . . 43
3.3.3 Completion and Checking . . . 44
3.3.4 Evaluator Generation . . . 44
3.4 Case Study: C-Front . . . 45
3.4.1 An SDF attribute grammar for C . . . 45
3.4.2 Performance . . . 45
3.5 Discussion . . . 46
3.5.1 Other works . . . 46
3.5.2 Future Work . . . 47
3.6 Conclusion . . . 48

**4 Extending a context-sensitive syntax 51**
4.1 Introduction . . . 53
4.2 Related Work . . . 55
4.3 The Transformation Chain . . . 58
4.3.1 SDF Grammars . . . 58
4.3.2 Scannerless Generalized LR (SGLR) and Parse Forests . . . 58
4.3.3 Disambiguation . . . 60
4.3.4 Transformations . . . 61
4.3.5 Pretty-Printing . . . 63

4.4 Case Study . . . 64
4.4.1 Design by Contract . . . 64
4.4.2 Syntax . . . 65
4.4.3 Compilation to C . . . 65
4.5 Discussion . . . 65
4.6 Conclusion . . . 72
4.7 Acknowledgments . . . 73

**5 Providing mouldable failure management** **75**
5.1 Introduction . . . 77
5.2 Problem . . . 80
5.2.1 Alert Reporting Mechanisms . . . 80
5.2.2 Alert Handling Policies . . . 83
5.2.3 A Game of Anticipation . . . 84
5.3 Separation of Concerns . . . 85
5.4 Alert Language Extension . . . 86
5.4.1 Distinguishing Different Alerts . . . 86
5.4.2 Specifying the Alert Reports of a Function . . . 88
5.4.3 Alert Handling . . . 90
5.4.4 Abstraction . . . 92
5.4.5 Sending Information from Callee to Caller . . . 93
5.4.6 Granularity and Funspaces . . . 94
5.4.7 Interfacing with Legacy Code . . . 96
5.5 Implementation . . . 96
5.5.1 Translation scheme . . . 97
5.5.2 Implementation issues . . . 98
5.5.3 Compiling to Aspects . . . 98
5.6 Related Work . . . 99
5.7 Conclusion . . . 101

**6 Implementing ConceptC++** **105**
6.1 Introduction . . . 107
6.2 ConceptC++ . . . 109
6.3 Motivation for a ConceptC++ to C++ transformation tool . . . 111
6.3.1 Software development . . . 112
6.3.2 Shipping a generic library . . . 112
6.4 Specification of the transformation tool . . . 113
6.5 Transformation: A naive approach . . . 114
6.6 Transformation: Context free approach . . . 114
6.6.1 Concepts . . . 114

6.6.2 Concept maps . . . 118
6.6.3 Auto concepts . . . 120
6.6.4 Constrained class template . . . 121
6.6.5 Constraint function templates . . . 122
6.7 Transformation: Dealing with overloading of function templates . . . 123
6.8 A better feedback on inconsistent uses . . . 124
6.9 Conclusion . . . 126

**7 ConceptC++ for categorical constructions 129**
7.1 Introduction . . . 131
7.2 ConceptC++ . . . 132
7.3 Institutions . . . 134
7.3.1 Standard specification building operations . . . 136
7.4 C++ with concepts as a Software Institution . . . 138
7.4.1 Interface / Signature . . . 139
7.4.2 Formulas . . . 141
7.4.3 Models . . . 141
7.4.4 Satisfaction relation . . . 142
7.5 Putting concepts together to build specifications . . . 142
7.5.1 Amalgamation by renaming inside signatures . . . 143
7.5.2 Parameter substitution . . . 144
7.5.3 Providing signature morphisms . . . 144
7.6 Discussion and Conclusion . . . 146

**8 Reusing concepts for testing 157**
8.1 Introduction . . . 159
8.1.1 Axiom-Based Testing . . . 160
8.2 Concepts . . . 162
8.3 From Axioms to Test Code . . . 163
8.3.1 Reusable Tests . . . 167
8.3.2 Concept Combinations . . . 167
8.3.3 Test Drivers / Suites . . . 168
8.3.4 Axioms for Object-Oriented Code . . . 169
8.4 Generating Test Data . . . 172
8.4.1 Associating a Data Set with a Type . . . 175
8.5 Discussion . . . 175
8.5.1 Equality Testing . . . 175
8.5.2 Algebraic Axioms and Imperative Code . . . 176
8.5.3 Axiom Selection and Algebraic Specification . . . 177
8.5.4 Experiences with Axiom-Based Testing . . . 179

8.5.5 Tool Implementation . . . . . 180
8.5.6 Future Work . . . . . 180
8.6 Conclusion . . . . . 181

**9 Conclusion** **183**
9.1 Tool building . . . . . 183
9.1.1 Results for different kinds of extensions . . . . . 184
9.2 Remarks on the extensions and transformations . . . . . 185
9.2.1 Alerts . . . . . 185
9.2.2 Concepts . . . . . 185
9.3 Contribution . . . . . 186
9.3.1 Language design . . . . . 186
9.3.2 Software . . . . . 186
9.4 Direction for future work . . . . . 187

# Preface

Perhaps I should skip the part of the preface where I tell my horrible life I had during my PhD to go directly to the part everybody is waiting for.

I shall thank Magne Haveraaen for being a very attentive supervisor. Karl Trygve Kalleberg for being not only a good coworker, but also a nice coding mate and a good friend out of the university. Anya Helene Bagge for being a very pleasant co-author, commenting on my dissertation, and more generally, helping me with the system of the university all along my PhD. Joseph Young for helping me with my poor English. I guess I made him suffer. Daniel Quinlan for providing me an enriching experience at the LLNL. Akim Demaille for having trusted me a long time ago, which brought me here now. And finally the students from the Transformers team: Alexandre, Vincent, Warren, Nicolas, Olivier and all the others.

# Abstract

Programming tools can be powerful companions for the software engineer. Unfortunately, language-dependent programming tools are hard to develop for languages like C++, and hard to maintain. Only a large demand can help a tool survive. For this reason domain-specific tools are out: a narrower domain implies less users, less demand, and then less interest in investing time. In this thesis we try to show that in some contexts we can write tools quickly by showing some experimentation of language features.

**Program transformation and C++.** C++ is a complex language. Which is then hard to do program transformation with it. The syntax is context-sensitive. And the context includes even the architecture of the target. Many front-ends and transformation environments are available. They respond to different needs, but none are universal, covering all aspects of the language.

**Modular specification for a context-sensitive syntax.** It is possible to disambiguate a context-sensitive grammar expressed as an ambiguous syntax by using attribute grammars (AGs). The class context-free syntax is closed under union, and AG rules can also be composed. In the end, instead of separating concerns as compiler passes, we can separate language features.

**Extending a context-sensitive syntax.** With a case study, we show the importance of the modularity of a syntax specification. We show that by introducing a few context-free productions with a few attribute rules we can easily extend a context-sensitive syntax. In this case we extended C with pre- and post-conditions. The extension of syntax is non-intrusive, the C parser is reused.

**Providing mouldable failure management.** Handling failures is important for building quality software. Often a language provide exceptions, or the function returns a special value, but the choice is by the developer of a library, not from its user. Alerts proposes a unified way to handle errors and let the user

choose the way to handler error. We show an implementation of alerts which does not need modification of the run-time environment. We extend the syntax of C providing new syntactic constructions.

**Implementing ConceptC++.** ConceptC++ is an extension of C++ aimed at resolving some needs for template meta-programming. We show that by generating code using template meta-programming, ConceptC++ can be treated as syntactic sugar of C++. The transformations are then context-free and can be applied to parts of a program.

**ConceptC++ for categorical constructions.** Concepts are like specification, giving us software institutions. And ConceptC++ provides us signature morphisms. With this in mind, we show techniques of software building using categorical constructions. Though concepts is not completely adapted for the purpose and have some deficiencies. We propose changes to concepts that would resolve the problems.

**Reusing concepts for testing.** Concepts can be finally reused for a third tool: generating testing code. Specifications can be used for testing, so can concepts. Axioms are transformed to test oracles. The transformation generates test drivers. The result is a program that automatically tests code against its specifications for given sets of typed data.

In general the tools developed are simple, written using very few lines of code. We believe this simplifies maintenance, increasing the tools possible shelf-life.

# Chapter 1

# Introduction

This dissertation explores how to write programming tools for C++ specially, it concerns language extensions.

To achieve maintainability, correctness and efficiency, modern software engineering relies on the use of programming tools for implementation[1]. Usually many of those tools are bundled inside a unique integrated development environment (IDE). The developer can then edit, re-factor, compile, debug, test, etc.

At its core, the development environment has a programming language. Programming languages provide different paradigms. It will have an influence on what problem the software will be possible address. However, most languages still target a large set of problems. These are general-purpose programming languages.

Then we have domain-specific languages (DSLs). Some problems are harder to resolve by a program in a general-purpose language. When the program is generated from a higher level formalism, the development quality can be raised. Typical examples of DSLs are formalisms for formal syntax, used to generate parsers. For some algorithms, a small change in the syntax can imply big changes in the code. If a program generates the code automatically from a syntax definition, then the cost of maintenance is reduced. For handling DSLs, we need a domain-specific programming tool. Often this can be a program transformation that will generate source code in a general-purpose language.

DSLs can be embedded into a general-purpose language. Then we have a domain-specific tool. However, domain-specific programming tools do not have

[1] When we talk about implementation, we do not consider the methodologies for the design.

to be used simply for a DSL. Other kinds of programming tools can be dedicated to a specific domain. For example, it includes optimization, correctness checking, verification, abstraction. The syntax can stay the same. However, we extend the semantics. In general even though we do not necessarily extend the syntax, we consider domain-specific programming tools as a domain-specific extension of a language.

## 1.1 Motivation

Providing tools for a given specific domain is important as it will ease development and maintenance. But, what happens with the development and maintenance of such a tool? A project can hardly have its own tools. However, the word "domain-specific" also implies that the pool of potential users will be reduced. What should be the size of this pool to ensure a future for a programming tool? In case of C++, we expect a big number due to the difficulty of manipulating the language.

Providing tools also means extending a language. There are already different ways languages like C++ let the user add specific domain tools, but the different techniques have their own limitations.

The usual way to extend a language for a specific domain is by providing a library. A library, in a language like C++, provides a set of routines and type.

However, a domain might need to do more than just add routines or types. Languages have infrastructures for extensions. The first we think of is macro-expansion. C++ uses the same pre-processor as C [WG14, 1999, WG21, 2003], but it is very limited. There is no recursion. There is no information on the context or the syntactic level. It only works at the lexical level.

C++ has another system for extension called template. Templates provide possibility of meta-programming. Czarnecki et al. [2004] show how an embedded DSL can be implemented using meta-programming. However, meta-programs are hard to write: no debugging support, unreadable error messages, long compile time, hardly portable. Moreover the full syntax of C++ cannot be accessed by meta-programs. The only manipulation can be done on expression templates [Veldhuizen, 1995].

C++ also provides the ability to overload functions and operators. This is a typical example of language extension, but overloading acts on function calls and does not affect loops.

Extending control flow can be useful for domain-specific optimization. For example, OpenMP parallelizes loops when they are properly annotated [Dagum and Menon, 1998]. Some programming languages like C++ (pragmas) or Java

provide an annotation [Gosling et al., 2005]. In general, it is possible to provide annotations inside comments.

The Annotation Processing Tool (`apt`) is a tool for Java where you can introduce a meta-program for pre-processing Java annotations at compile time [Lorimer, 2005]. This tool is limited since it does not modify the code, but it gives some information about the types. Though, it is possible to generate code.

Some languages (Java, ObjectiveC, etc.) support reflection [Smith, 1982]. Reflection is a form of meta-programming. It gives at run-time access to the structure of the program itself. It can have read-only, or write access. The reflection can give access to declarations, or eventually down to instructions. And for these reasons, it can provide libraries the ability to modify its behavior depending on the user code. An example of such a library was the former versions of JUnit [JUnit.org]. A test case is often a collection of test oracles. Oracles can be written as routines. Registering test oracles is little cost. It might seem simple, but is prone to omission. And a test not registered, is a test not run. Discovering test oracles can be done by reflection.

There are tools which introduce syntax extension within the language. The old fashion way is to pass source strings to functions. For example, `printf` from the C library uses this technique. The validity of the string is more or less checked by modern compilers. The same occurs for database accesses. A common DSL for this is Structured Query Language (SQL). Both have raised security concerns because the language was not systematically checked at compile-time[2].

There are approaches that are far better. For example, MetaBorg [Bravenboer and Visser, 2004] proposes a technique to embed DSLs inside general purpose languages. This is good way for checking correctness of the domain-specific user code.

Not all tools need to introduce new syntax, but these tools can introduce new semantics. In Codeboost, optimization rules are described using C++ syntax [Bagge et al., 2003]. However this extra piece of source code is never used by the program and has a different set of semantics. In this case, it was for practical reasons.

There are also tools for reducing a language. Adding restrictions in a language can have several benefits such as introducing nicer semantics. Jim et al. [2002] provide restrictions (along with extensions) that ensures the safety of pro-

[2] `printf` provides `%s` as possible formatting, which writes the current number of characters printed as integer to the address by the corresponding parameter. If the attacker can provide a string, it can then put any value at some point of the memory. For the SQL, several commands can be passed to the database by separating them by a semicolon. In case of the attacker gives a string that is concatenated with a command, she can then escape this command and start another one.

grams. Stroustrup [2005b] uses this motivation for providing an environment for restricting languages.

Other restrictions of a language might be interesting. There are features that make optimization harder such as aliasing, dynamic binding, linear dependencies of iterations of a loop, side effects, etc.

The source code is usually transformed into an executable form, but we might want other products. Documentation generation usually takes the original code with annotations and generates a human-readable document. For this reason, we need to redefine parts of the language. In chapter 8, we present an example of transformation generating a program for testing the validity of the source code.

### 1.1.1 Extensions: success or failure

We can find several domain-specific extensions for C++ [Stroustrup, 2005b]. In this section, we look at two comparable extensions which have the same goal: OpenMP [openmp.org] and CC++ [Kesselman, 1996]. Both aim to provide a parallel extension of C++.

In the case of a shared memory architecture with multiple processors, it is important to spread computation across the processor resources. The common target for tools providing abstraction over parallelism is loops. Many loops have no dependency from one iteration to another. Iterations can be re-scheduled on several processors. This is what these two extensions propose.

CC++ has been developed in an academic context. OpenMP is the pseudo-standardization from different implementations in Fortran. OpenMP not only had an industrial background, it also had a pool of users and suppliers. By demand, OpenMP has been adapted to C++.

Secondly, OpenMP uses pragmas instead of modifying the front-end. In a context of a compiler, the same front-end is used for both programs using and not using the extension. Whereas with CC++ a compiler would have two front-ends.

CC++ has been designed as a pre-processing tool producing C++ instead of modifying a given compiler that can be reused with any compiler. This is the same approach we want. Then, the same unique implementation can be reused in front of any compiler. In theory, it is better to concentrate maintenance effort on a unique implementation than to develop it several times. However, in practice, nobody does it for you. It must be part of the compiler. This leads us to a requirement of human resource for maintenance.

Maintenance cannot be done by academics for several reasons. The researchers are not paid for maintenance, their students come and go, and funding is inconsistent.

Stroustrup [2005b] proposes an approach where the only modification of language is restrictions to enhance the semantics, which is checked by an analysis tool. Then all development is made on a library that relies on the semantic properties of the restricted language.

The goal of this dissertation is different. Reduction of maintenance can also be achieved when the language extension is specified in a concise way. The formalism is not the only goal. There might be a class of extensions that is suitable for concise language specification.

### 1.1.2 The transformations presented in this dissertation

While chapter 4 shows an example of extension, it does not intend to be used. It is a test for the tools. It provides pre- and post-conditions to the C programming language.

Chapter 5 provides an extension for *alert* handling. Alerts are comparable to exceptions. Alerts give more power to the user to choose how to deal with errors rather than being forced by the choice of the developer of the library she uses. A library can throw exceptions, or return a special value specified in the manual page as an error value (often a negative integer for example), like in C-style programming. Alerts propose an uniform interface to failures that are generated by exceptions or by special return values. The transformation generates the proper handler code by de-sugaring the extended syntax. In that chapter, the extension is made on C.

Chapter 6 is about ConceptC++. This extension of C++ has proposed by Siek et al. [2005]. An experimental implementation has been made on top of GNU Compiler Collection (GCC) [Gregor et al., 2006b]. We do a simpler implementation as a program transformation. This transformation is intended as a first step in the development of language tools for ConceptC++, as well as help with portability. It does not replace the ConceptC++ compiler.

Chapter 8 provides a tool for testing, using the ConceptC++ extension. ConceptC++ provides axioms. Concepts can actually be seen as specifications. It is possible to generate test drivers and oracles from specifications. The tool presented does this job.

## 1.2 Writing tools

When it comes to writing programming language tools for a language like C++, different methods are used. The first is to modify the compiler. The compiler is the software that has the most knowledge of the language. Although it can

be very difficult to understand the source code of a compiler (*cf.* section 2.3.4 for example), this is not the main problem. Maintenance is the main problem. The most famous compilers change very quickly, and maintaining modifications of a compiler can be very costly. For example, some transformations (CC++ for instance [Chandy and Kesselman, 1993]) were based on a front-end of GCC, but the front-end of GCC has been replaced for version 3.4.0.

Though, this is a good solution when the extension is maintained in the trunk of development on the compiler. For example OpenMP is developed directly inside different compilers.

Having a modular compilers looks the best solution. Once a compiler is developed as a library, the interface is stable enough to extend the compiler. Although this is possible, like shown by JastAdd [Ekman and Hedin, 2007] or Silver [Van Wyk et al., 2009], it is not done systematically. The goal of a compiler is not to directly provide this kind of feature and providing a larger interface complicates the maintenance of the compiler. Having a compiler that will be for sure maintained in the future is important for the choice of compiler used for a project. Nothing shows that the few existing modular compilers are sure to stay alive. Moreover no attempt has been done on pure C++ (OpenC++ [Chiba, 1995], Montana [Karasick, 1998], etc.[3]). Choosing a language like C++ has its interest mostly on the fact that the source code should be portable from one compiler to another. Any major modification of C++ that would make it not portable certainly has certainly as interest as a compiler for a completely new language.

Then there is source-to-source program transformation. The classical approach is having a representation format of a program, usually grammar-based [Cameron and Ito, 1984], and a front-end capable of building representation of source code.

## 1.3 Program transformation

A program source consists of data with structure and semantics. Usually a program is a flow of characters. The syntax gives us the structure of the program. The transformation definition give us the semantics.

A program transformation takes a program in one language to produce a program in the same or another language. Though the output program does not have be just a translation. A program transformation can be for example an analysis generating a report about the input program.

To simplify program transformation, we need abstractions of the program.

[3] Note that those projects are not maintained.

The commonly used abstraction is an abstract syntax tree representing the structure of the program. There are also graphs representing control flow, data flow.

There are several kind of languages, from regular to recursively-enumerable. Different classes imply different parsing algorithms, but also different formalism in the syntax definition. For instance, the most desirable languages should be context-free because it is possible to express any context-free syntax definition with a Backus-Naur form (BNF)-style of formalism, which is more friendly than formalisms for other higher classes of languages.

Recursively-enumerable languages correspond to languages needing a Turing-complete machine to be recognize. Context-sensitive languages need a machine that can be bound in memory and time. Context-free languages are the languages that can be expressed with BNF style of formalism. Deterministic context-free languages do not have any ambiguity. Regular languages can be treated by finite automatons.

Not all languages used in industry are context-free. This makes the writing of parser a bit more complex. Moreover, context-sensitive languages are hard to modularize.

To write program transformations, we need some tools. The most important are the parser generator, the abstract representation, and the pretty-printer generator[4]. A parser generator helps to make a parser, the abstract representation to manipulate the abstract program, and then the pretty-printer generator to generate the pretty-printer.

By convention, we also need to write some semantic analysis tools, and more importantly, a type checker. Those analysis tools are usually taken as transformations by adding information on the abstract representation.

There are environments providing quick way to build these tools, such as StrategoXT [Bravenboer et al., 2008], Turing eXtender Language (TXL) [Cordy, 2004], DMS [Baxter et al., 2004] and Algebraic Specification Formalism + Syntax Definition Formalism (ASF+SDF) [van den Brand et al., 2002].

Often, program transformation environments expect context-free grammars. Designers of program transformation tools usually think it is silly to want more. Designers of languages usually think that it is silly to want less.

## 1.4 Language specification

Defining a language can be done either by writing code or by adopting some formalisms. Formalisms can often provide a higher degree of re-usability.

[4] A pretty-printer is an inverse operation of parsing, from an abstract representation, it gives a possible source code.

### 1.4.1 Syntax

Specifying a syntax is usually done by defining a set of symbols, a start symbol, a set of tokens, and a set of productions. A production is a rule deriving a list of symbols or tokens into another list of symbols or tokens. Although formal languages can be defined with these kind of productions, it is usually not practical. First it is not natural. Secondly it may not terminate. The most common class of grammar which is used is the context-free class of grammars. The derivation rules are restricted to be a symbol on the left hand side.

There are other aspects as to why context-free grammars are the most seductive. One is the closure under union. This is the key for easy syntax extension.

Finally, some use deterministic context-free grammars (or subclasses) for efficiency reasons. They are also desirable because deterministic grammars have no ambiguity. This is the case of lookahead LR parser (LALR) grammars. Unfortunately deterministic context-free grammars and its subclasses are not all closed under union, which makes them hard to expand.

Parsers are still widely written by hand. This is especially true for C++. The reason is that C++ is believed to be faster to parse as a LL(*) with backtracking. Such an algorithm is usually written by hand. While there are few transformations to do on the syntax before translating it into a parser, it is still human-readable code, but it is hard to work on it.

### 1.4.2 Semantics

Like parsing, semantic analysis can be done by hand. There are also some more formal formalisms. The reasons for writing into a semantics formalism is the same as those for parsing. It is sometimes possible to easily do operations like union of semantic analysis.

One example is Algebraic Specification Formalism (ASF) [van den Brand et al., 2002]. It allows us to write conditional equations. An equation gives an equivalence between terms which gives transformation rules. The language is then described as an algebraic specification. The syntax is the signature, and the transformations rules are the specification. Instead of separating concerns of language by compiler pass, the separations can be done on the features of the language.

Another way to specify a language is using attribute grammars (AGs) [Knuth, 1968]. Each parsing derivation has attributes. These attributes are assigned values. To specify an AG, rules for assigning the values to attributes must be written. Each production rule of a syntax definition gets a set of rules. Each rule specifies how one attribute of the current production or a child production

must be assigned using other the value of other attributes.

On a given derivation (for a given source code), there is a dependency graph that can be found for finding the values of the attributes. In the same way that grammars have several classes for several algorithms, AGs also have properties that allow them to be evaluated by different algorithms. It is usually necessary to have a graph without cycles and where all attributes are assigned. This is the condition for a well-formed AG. Some other restrictions of AG systems can ask for a graph following the shape of the derivation tree of parsing so that the evaluation of the tree can be done at parsing-time.

## 1.5 C++

We see in details in chapter 2 why and how C++ is complex to process when doing program transformation.

The first problem is the syntax. C++ is a context-sensitive language which makes hard to process with general purpose program transformation environments.

The second problem is the complex semantics. However C++ already has a powerful system with template meta-programming. In chapter 6, we show an example of transformation that does not need (or very little) contextual information by generating meta-programming code.

### 1.5.1 Context-sensitive parsing

In theory, most languages with static type-checking are not context-free, they are at most context-sensitive if not just recursively-enumerable. The reason is that type-checking needs to know the context for accepting or rejecting an input.

In practice, the syntactic analysis and semantic analysis are separated. There is a syntactic language which is usually context-free. Then the semantic analysis rejects parts of the syntactic language. The overall language is context-sensitive, but the parsing technology uses a context-free parser. The syntax has to be convenient enough to reject as many illegal programs as possible.

In some contexts, it implies that the syntactic language is ambiguous. Java for example does not have this problem, but for this they rejected some programs from the language such as the following statement.

```
Integer a = (Integer) - 0;
```

If we try to parse it, the compiler will complain of not finding the symbol `Integer`. Here we are trying to cast the expression `- 0`, but as syntactic

language, this is ambiguous with the subtraction operation. To resolve the ambiguity, they decided to reject all expressions starting with a sign operator in a cast expression using a non-primitive type. Since the compiler is not expecting a type, it complains it cannot find the symbol as a variable or a constant.

C allows this kind of ambiguity. This means that several derivation trees are accepted, then the syntactic analysis accepts only one derivation. This is how it should work in theory. In practice, compilers break the design conventions and call the semantic analysis in the parser.

C++ would be recursively-enumerable but not context-sensitive if there was be no limitation of template recursions. However, even though a language can make the life hard for the developer of the compiler, it would be a big problem if a compiler did not terminate on some source code. For this reason the template recursion is limited. Other than that, templates still have hard properties for the developer of a compiler. C++ is in theory possible, is very complex to implement.

Consider the following piece of code in C++.

```
template <typename T>
struct A;

template <typename T>
struct B : public A<B<T> > {
};

template <typename T>
struct A : public B<A<T> > {
};

void foo() {
  (A<int>::t) - 0;
}
```

To know how to parse the expression `(A<int>::t) - 0`, we need to know the nature of `t`. For this we need to instantiate an infinite number of templates. The standard proposes to limit the level of recursion. Compilers usually allow the user to change the default value, but the limit is still to the discretion of the compiler. When writing code using recursion in templates, even if it halts, the code might be or not be in the language of the compiler.

C++ otherwise behaves more or less like C, but it has other complications. These are shown in chapter 2. The problem of parsing context-sensitive languages is addressed in chapters 3 and 4.

## 1.6 ConceptC++

ConceptC++ is a mainstream extension proposed for the C++0x standard. It has since been rejected. In our case, ConceptC++ provides a way to express specifications. Specifications open a possibility to a selection of programming tools: optimization, testing, verification, documentation, etc.

The need for concepts appeared with the use of templates. Templates offer parametric polymorphism to C++. This polymorphism being static, the compiler would never create an unsafe program, but templates are just templates and do not have a type. It cannot be checked in all uses. In the following piece of code, we have a universal quantifier on type `Iter`.

```
template <typename Iter>
typename Iter::value_type min(Iter start, Iter end) {
  if (start == end) throw 0;
  typename Iter::value_type ret(*start);
  ++start;
  for (; start != end; ++start)
    if (*start < ret)
      ret = *start;
  return ret;
}
```

It implies that `Iter` has several requested operations. It needs a `*` operator, a `!=` operator, a `<` operator, a `++` operator, and it can be copied. Finally, there is a member type `value_type` and this type is comparable with the return of `*`.

If we call this function with parameters whose types do not comply with this need, then there is an error. This is normal, but the error appears deep inside the code. We want the compiler to complain that we passed a wrong parameter to the function. This happens because the template does not have a type.

Cardelli and Wegner [1985] introduces a notion of bounded parametric polymorphism. Later, Haskell uses this feature by introducing *type classes* and *instances* [Wadler and Blott, 1989]. Bernardy et al. [2008] show that ConceptC++ *concepts* and Haskell *type classes* are very similar.

For the previous example, we can introduce a concept for iterators.

```
concept Iterator<typename T> {
  typename value_type;
  T operator++(T&);
  T operator++(T&, int);
  bool operator==(T, T);
  bool operator!=(T a, T b);
  value_type operator*(T);
}
```

Then we can safely add requirements to the function.

```
template <typename Iter>
requires Iterator<Iter>,
  std::CopyConstructible<Iterator<Iter>::value_type>,
  std::CopyAssignable<Iterator<Iter>::value_type>,
  std::LessThanComparable<Iterator<Iter>::value_type>
Iterator<Iter>::value_type min(Iter start, Iter end) {
  if (start == end) throw 0;
  Iterator<Iter>::value_type ret(*start);
  ++start;
  for (; start != end; ++start)
    if (*start < ret)
      ret = *start;
  return ret;
}
```

A concept is a set of requirements on a set of types. The requirements can be syntactic or semantic. Syntactic requirements are given by declarations. A concept can require associated types, operations (functions, methods, operators). It can also require other concepts by refinement or concept requirement. In chapter 7, we see that these refinements and requirements are constructions in the framework of institutions.

The semantic requirements are given by axioms. Axioms are sets of conditional equations. These requirements are not verified by the compiler. They are used for documentation, and are expected to be used by external tools, such as verifiers. The compiler has the right to use the axioms for optimization purposes.

```
concept Monoid<typename T> {
  T operator+(T, T);
  T id_elt();

  axiom Identity(T a) {
    a + id_elt() == a;
    id_elt() + a == a;
  }

  axiom Associativity(T a, T b, T c) {
    (a + b) + c == a + (b + c);
  }
}
```

A concept map is used to declare how types fit with concepts. For example, associated types need to be defined. In the example of iterator, we needed a `value_type`. We can also define missing operations.

```
concept_map Monoid<int> {
  int id_elt() {
    return 0;
  }
}
```

Concepts and concept maps can be seen as algebraic specifications and models. There are of course lots of other ways to see concepts [Zalewski, 2008].

## 1.7 Overview

In chapter 2, we first look at C++ and what makes it hard to write tools for this language. Then we look at the utilities available. We will see how these tools are designed. Then we see their limitations and in what context they are useful.

Chapter 3 is a paper explaining in more details a technique briefly presented in chapter 2 about using AGs for parsing C and C++.

Chapter 4 is an article showing by example how to extend C.

Chapter 5 talks about alert handling. This is a language feature that we introduces in C. It is comparable to exception handling but gives more control to the user.

Chapter 6 shows how we can handle ConceptC++ as a syntactic sugar of C++. It is actually not completely a syntactic sugar, but the paper discuses that subject.

Chapter 7 expresses some views about missing points in ConceptC++.

Chapter 8 is about testing and ConceptC++. The interesting point here is the reuse of a language extension for another purpose than the original one.

# Chapter 2

# Program transformation and C++

Before detailing how to write tools for C/C++, this chapter shows different environments for building C++ processing tools that exist in the real world. There is no perfect environment. Depending on the need, one might find an environment that is good enough.

Our needs revolve around the possibility to easily extend the syntax of the language. For some environments, the front-ends are not be designed for this purpose. While it is still possible to modify the front-end, in practice, the maintenance of the language extension is costly. For this reason, the extensibility is our first requirement.

This chapter is an overview of environments and does not describes how to write tools. Our method will be presented in later chapters.

## 2.1 Introduction

To do program transformation on a C++ program, we need tools. This includes a parser and a parse tree (PT) convention. In this chapter we look now to parse C++03 and tools available.

Dealing with C++ is hard in practice. In section 2.2 we look at the reasons why this is. In section 2.3, we list different front-ends available and see for what they are good and also what they are bad for. In section 2.4, we will look at different environments for program transformation and see how they are adapted to dealing with C++.

## 2.2 C++ language's properties

C++ is nearly an extension of C with which is already a language difficult to work. C++ is also a complex language as it brought a lot of novelties, but also a lot of problems on top of those already in C. These problems do not touch (directly) the user of C++, but they touch the compiler developers or any other person wanting to do program transformation of C++ source code. In this section, we look what makes C++ a hard language to work with.

We have to note that what makes C++ very hard is justified. C++ provides several modern programming paradigms while assuring backward compatibility with C and previous implementations of C++. This is very convenient for the user.

### 2.2.1 The pre-processor

The module system of C++ is archaic. It is based on the C preprocessor doing macro expansion on a stream of tokens. Depending on the environment of compilation, different tokens will have to be parsed. This can lead to explosion of the number of programs to parse. Thus, the preprocessor is too permissive. Finally, we may have code that is never parsed on our platform, and we can never check its validity or even that it parses. Problems arise once the source code is shipped for other platforms.

There are three ways to deal with the preprocessor. The first is to ignore at all pre-processing directives and do the transformation on the preprocessed code. Unfortunately, this is not enough for every program transformation. For example assisted source re-factoring needs to preserve the source code before being preprocessed.

The second is to try to find a way to include the syntax of the preprocessor inside the C++ syntax as proposed by Yao et al. [2006]. With this technique, it is not possible to process all programs. The syntax definition has to be written to accept conventional source code but disallow some valid code that is believed not to be written by developers. Comparably, Padioleau [2009] uses heuristics to recognize the purpose of a macro. For each purpose, a use of a macro has a certain syntactical property (*e.g.* a statement, an expression, etc.).

The third approach is to pre-process the source code but annotate so that pre-processing can be reversed. The PT still corresponds to a preprocessed source code, but you can access information about pre-processing. Also, portions of code that are not modified are re-transformed using the macros from the pre-processor. This technique may be the wisest one and it is used by Waddington

and Yao [2007] for example. However, this does not allow analysis of the whole source code, but simply preprocessed code given a configuration.

### 2.2.2 C++ is context-sensitive

If we intend to parse the following piece of code without context, we have some problems.

```
void fun() {
  (a)(b);
}
```

In this piece of code, if `a` was previously declared as a type, the expression `(a)(b)` is considered a cast expression. Conversely, if `a` is an object name (variable, function, field, method...), then the expression becomes a function call. Eventually `a` can be function pointer variable.

The problem originates in the C language. C++ inherits many parsing ambiguities from C, but some new ambiguities were introduced. For example, in C++, there are default values for function parameters. We can re-declare the function in whatever scope we want, in such a way that we change the default values for the parameters. It means that function declarations are legal inside function bodies. However, we also introduced declarations with constructor arguments. In some cases, both syntax conflict such as in:

```
void fun() {
  int a(b);
}
```

In this piece of code, we can have either a declaration of `a` as an integer initialized to the value returned by `b`, or a redeclaration of function `a`.

When we want to do program transformation on a small piece of code, we still need to know the exact context. The module system relies on the preprocessor that concatenates many files. As a result, the parser needs to parse a huge source code even though our source code is small. The headers of the standard library may be hard to parse and use lots with tricky techniques of template meta-programming. Thus, parsing correctly a "hello world" program needs a very robust parser.

### 2.2.3 The context is locally not left-to-right

We want to be able to declare fields after methods that use them. As in Java, C++ allows this. It means that the semantics of a method depends on code appearing

afterward. In itself, this is not a problem. Unfortunately, the semantics have a part in the parsing context. Consider the following piece of code.

```
struct A {
  void meth(int b) {
    (a) (b);
  }

#ifdef FOO
  typedef int a;
#else
  int a;
#endif
};
```

Here we have the same problem as before: to parse `(a) (b)`, knowing the nature of `a` is needed. Since the declaration of `a` appears after the method definition, the parser needs to skip the method until the class has been parsed.

Parsers vaguely parse a method body to find its end. This consists of counting curly braces (`{` and `}`). Then a stream of tokens is attached to each method. Once completed, parsing of the class is resumed. Once the parser reaches a point at a global or namespace scope, it comes back to those streams and parse them correctly. Eventually the second pass can be done once the complete file has been parsed.

### 2.2.4 The context is sometimes incomplete

C++ has a powerful template system. Unfortunately, it has to deal with the ambiguities of the C++ syntax. Templates are incomplete pieces of code that are reused in several different contexts. If the contexts can be different, how can parsing be handled? Consider the following piece of C++.

```
template <typename T>
struct A {
  void meth(int b) {
    (T::a) (b);
  }
};
```

Since we do not know `T`, we do not know the concrete nature `a` has. C++ says that in this case, `a` is a static field of `T`. If it is a member type, the developer has to provide a keyword to tell the parser `a` is expected to be a type when `A` is instantiated. Those keywords are **`struct`**, **`enum`**, **`class`** or **`typename`**. In

general, the latter is used. Here, if `a` is supposed to be a type, the user writes `(typename T::a) (b);`.

In practice, developers rarely remember to use this keyword there since it is not obvious for a human being when these things are ambiguous. Most developers just add the keywords in the right places once they get error messages from the compiler.

In the same order, the brackets for template arguments (<>) can be interpreted as comparison expressions. In case of an incomplete context, the developer needs to use the keyword template to express which names have to be template names. If not, the names are considered static fields, and everything is parsed as an expression. The following example has two different PTs depending on whether the keyword **template** is commented.

```
int c;
template <typename T>
struct A {
  void meth() {
     T::/*template*/ foo < 0 > ::c;
  }
};
```

In the case when the keyword is present, we access a static field `c` inside `T::foo<0>`. If the keyword is not present, we have two comparisons. First, we have one comparison between the static field `T::foo` and `0`. Second, we have one comparison with the result of the previous comparison and `::c`. This may not make much sense to have two comparisons like this. However first, C++ allows overloading of operators, and secondly to disallow this, we need a more complex syntax definition on expressions.

### 2.2.5 The context depends on template instantiation

Using members of a class template implies instantiation of the template in order to discover the nature of the member. For instance, in the following example, `a` can be either an enumerator or a type depending whether the parameter of `A` is even or odd. To be able to know if `g` is a function declared or an initialized integer variable, we need to instantiate the template.

```
template <unsigned N>
struct A : A<N-2>{
};

template <>
struct A<0> {
```

```
  enum { a };
};

template <>
struct A<1> {
  typedef int a;
};

void f() {
  int g(A<6>::a);
}
```

`A<6>` is not found and `A<N>` is instantiated with `6` for `N`. Then it instantiates `A<4>`, then `A<2>`, and finally `A<0>`. Member `a` is known in `A<0>` to be an enumerator. So here we initialize the variable `g` to the value of the enumerator.

If we were to use `A<7>::a` instead, we would have a function declaration.

### 2.2.6 The context needs type inference

The construction **`sizeof`** returns the storage size of a type. This size is an integer constant and can therefore be used as a template argument, but **`sizeof`** also returns the storage size of the type of the value returned by an expression. Since any expression has a constant concrete type, this is also a constant. Thus, it can also appear as a template argument.

To know this constant size, the parser needs to infer the type of the expression. This expression could actually use overloaded functions, template expressions, etc. The full type inference is needed.

This means that type inference has to be done at parse-time.

### 2.2.7 The template instantiation is architecture-sensitive

The construction **`sizeof`** is constant for one environment, but it may vary depending on the architecture. The standard simply gives order in sizes of basic types, but it does not specify what they should exactly be. We require not only basic types, but also alignment in size of structure. The architecture is not enough by itself. Sizes may vary depending on the strategy of compilation. For instance, alignment changes sizes depending on strategies for optimization purpose. The following example is parsed differently by GNU Compiler Collection (GCC) on a x86 architecture depending on the parameters passed. Here, the field `A::b` is aligned differently depending on the use or not of option `-malign-double`. Alignment also differs from one compiler to another.

```
struct A {
  char a;
  double b;
};

template <int N>
struct B {
  typedef int a;
};

template <>
struct B<12> {
  enum { a };
};

int main() {
  int f(B<sizeof(A)>::a);
  return 0;
}
```

The size of `A` in the case of no alignment of the type double, will be 12 bytes. With alignment, it becomes 16 bytes.

## 2.3 Front-ends

Before describing the different front-ends, we first look at the needs for a front-end. Different needs have different solutions. This means that a front-end used for a certain application cannot necessarily be reused by another. Then we will deduce the determinants to look at. And finally list a review of selected front-ends categorized by approach.

### 2.3.1 Needs

#### integrated development environment (IDE)

Integrated development environments (IDE) often contain a parser. It can be useful for syntax highlighting, indexation, re-factoring, collapsing definitions, etc. However, these purposes do not need a normal parser. For example, syntax highlighting needs to understand programs that are rejected by the language. Some parsers are not actually real parsers but just try to guess parts of programs based on heuristic methods. Though re-factoring, for instance, needs a normal parser since it can require a correct code before applying the transformations.

Not all components of an IDE require a normal parser. Parsers for such functionality from an IDE cannot always be reused for other purpose. However recent IDEs try to provide refactoring tools. These need a good abstract syntax tree (AST). In term, we can expect IDEs to provide a normal parser.

#### Documentation generation

Documentation generation does not need a full parser. It does need to parse definitions, but simply declarations. Most ambiguities of the language are in the definition. It means that parsers used for documentation generation do not need to be very advanced, and they usually are not.

#### Program transformation

We can split program transformation systems into two groups. On one side we have the systems that are language-dependent. Even if these systems can handle several languages, it is not possible to extend them easily. For example, Rose [Davis and Quinlan, 1998] is such a system. The other side is composed with systems that are language-independent. The problem of language independent systems is that these systems are tested only on simple languages. For instance Algebraic Specification Formalism + Syntax Definition Formalism (ASF+SDF), StrategoXT, and Turing eXtender Language (TXL) have not been able to process C++ as easily as language-dependent systems.

Another aspect is the ability to modify or produce a program. Some systems are made mainly for analysis, but it starts to become difficult when we want to generate new C++ source as intermediate output. Rose has a static computed data flow graph. However if you modify the program, there is no other way to re-generate it than to print out the source code and re-parse it. This is often what happens with C++ program transformation systems.

Not all systems have an easily modifiable C++ parser.

### 2.3.2 Determinants

Available front-ends all have strengths and weaknesses. Depending on the needs, there are different possible parser available.

There are few things consider.

**Level of compliance** This important point is actually hardest to determine for any front-end. There are benchmarks that help determine the compliance, but running them all on all front-ends is time-consuming. Here, we do

not enter in the details. A large amount of the existing source code is not standard-compliant anyway. This can be explained by the fact that compilers do not always respect the standard. Some illegal codes can pass whereas the standard rejects it. In practice, it is common to get errors when trying to compile a project with another compiler, or even with a newer version of the same compiler. Nevertheless, it is possible to quickly test if a front-end is capable by testing with few simple source code involving strange context. A huge number of front-ends do not pass the examples of section 2.2.

**Interface** The interface language for the front-end is important. If it is a different language from the one we want to use for writing our program transformation. This is another piece of code to develop and maintain (the front-end might be maintained, so this glue needs it as well).

**Extensibility** For most of our usage we want extensibility, either to change the semantics or to extend the syntax with a domain-specific language (DSL).

**License** The license is quite important. It might be important to know how to redistribute the software. However, for C++ front-ends, we have another issue. If we want to adapt the front-end to a domain-specific dialect, how can we redistribute the modified front-end?

**Support** There are many parsers without any active development for years. Others are used industrially.

### 2.3.3 Yacc type

In 1998, the first standard of C++ was released, but compilers have been written for C++ for many years prior. Different approaches for parsing C++ have been used. With the standardization, existing parsers may have had to be redesigned. C++ is not complex at the lexical level. It can usually be handled by classical methods. However, the next generation of C++ (C++ 0x) may need more modern techniques. The typical example is operator >>. It is one token, but cases exist where we can expect two tokens > in a row. In C++ 03, the user needs to separate them with a white space.

Yacc is a LALR parser generator. It generates C code. It has been widely used because the program is quite light, fast to execute, and can process most languages. Based around C, it is possible to modify its behavior.

It is not a good solution for parsing C++, but since it is popular for other languages, it has been often chosen. To handle C++ with a LALR parser gener-

ator, we need to do more. There can be pre-disambiguation at lexical analysis. Secondly, there can be post-disambiguation when the AST is being built.

Moreover, using LALR parsers, the grammar should not be ambiguous. Unfortunately, the syntax definition proposed from the standard is ambiguous. It means that that the syntax definition has to be modified from the one that is standard.

#### Harmonia C++ parser

Harmonia is a project for programming by voice intended for IDEs [Boshernitsan, 2006]. It has a C++ parser written in a language called Ladle, which is then translated to a Bison syntax definition. The syntax definition does not have ambiguity, but the parsing is wrong. Moreover, while the source is publicly accessible, the license is restricted for research and education.

This work can be found at `http://harmonia.cs.berkeley.edu/`.

#### FOG

Flexible Object Generator (FOG) is a meta-compiler for C++ providing extensions to replace the preprocessor. It contains a C++ grammar written in Yacc.

This C++ grammar does not disambiguate properly. Rather, it accepts the C++ as the standard defines it, but does not build a tree that corresponds to the semantics. The ambiguities are resolved statically, and no type information is used during the parsing. This causes, for example, `(a)(b)`, where `a` and `b` are identifiers, to be always parsed as a function call and never as a cast expression. This happens even if `a` has been declared previously as a type. Willink [2001] explains that the disambiguation has to be done during the semantic analysis afterward.

FOG has not been maintained since 2001. It is also hard to compile it without the Sun C++ compiler.

This work can be found at `http://www.computing.surrey.ac.uk/research/dsrg/fog/`.

#### OpenWatcom

Sybase has released the Watcom compiler as open source software. This compiler handles C++. The C++ parser is based on a Yacc definition. It is a custom Yacc dialect. The parsing is LALR(1) with parser stacks in case of ambiguities, which makes it quite complex. Watcom is an old compiler which was mainly developed before the standard was released.

Watcom is hard to compile and extract the front-end even if it is still possible. Watcom does not parse better than older versions of GCC.

This work can be found at `http://www.openwatcom.org/`.

#### Keystone

Keystone is a C++ front-end written in Flex, BtYacc and TreeCC. Keystone is released in GPL2. The grammar is written with BtYacc and the semantic analysis in C++.

Disambiguation on Keystone is made through token decoration. This process is described by Malloy et al. [2003].

Keystone is no longer maintained.

This work can be found at `http://www.cs.clemson.edu/~malloy/projects/keystone/keystone.html`.

#### Old GCC

The GNU Compiler Collection began to support C++ in 1988 with version 1.22. The parser was written using GNU Bison whereas the lexer was written manually. Later, the development of Experimental/Enhanced GNU Compiler System (EGCS) (a fork of GCC) lead to a better C++ compiler. On version 3.4.0 of GCC, the front-end has been completely changed.

GCC is released under GNU General Public License (GPL) license.

Prior to version 3.4.0, GCC was based on a Bison parser. The tree was mostly corrected during semantic analysis, but it seemed costly to maintain. The newest versions are treated in section 2.3.4.

This work can be found at `http://gcc.gnu.org/releases.html`.

### 2.3.4 Hand written recursive-descent

C++ is not LL(k), but having backtracking allows a recursive-descent parser to process it. Some parsers written in C++ have a nice way to handle backtracking without producing too much extra-code.

Note that method definition inside the class definition needs to be parsed once the parsing of the class has been finished since forward members may be used inside the method definition (*cf.* section 2.2.3). One technique is to attach to the method a balanced (in term of parenthesis) stream of tokens to be parsed once the class definition is parsed.

This kind of parser is widely used because it is very efficient.

### OpenC++

OpenC++ is a C++ front-end published under an open source (BSD-style) license. The parsing is made with recursive-descent with backtracking. It was originally made as a metaobject protocol (MOP) for C++ [Chiba, 1995].

OpenC++ transformations are done by writing them in C++ (meta-level) and attached as plugin to the compiler which transforms the transformed program (base-level) before feeding the C++ compiler. OpenC++ actually provides an extended C++ to be able to connect the meta-level and the base-level. It is not be hard to reuse the abstract syntax tree of OpenC++ in another environment by writing a meta-level program that outputs the tree while telling OpenC++ not to process the program.

Unfortunately, OpenC++ does not seem to provide a real C++ parser. For example ambiguities, such as the cast with binary and unary operators are not handled.

The last release of OpenC++ was in 2004. The source repository is no longer updated.

This work can be found at `http://opencxx.sourceforge.net/`.

### Synopsis parser

Synopsis is a front-end written in C++ and has bindings in Python. It is quite easy to reuse the library, but unfortunately, the documentation is incomplete. The parser is based on OpenC++. It is a recursive-descent algorithm with backtracking written in C++. Unfortunately, it does not disambiguate since there is no semantic analysis. Synopsis has been used mainly for documentation generation.

Synopsis is released in GNU Lesser General Public License (LGPL) 2.1. It is released less than yearly, but the repository seems to be active.

This work can be found at `http://synopsis.fresco.org/`.

### KDevelop C++ parser

KDevelop is an IDE for K Desktop Environment (KDE) under GPL license. It includes several parsers useful for syntax highlighting for example.

The C++ front-end of KDevelop is written in C++. It uses a classical recursive-descent algorithm with backtracking and a hand written lexer. I produces a simple AST . It does not include any semantic analysis which means that the parsing is not complete since disambiguation needs semantic analysis.

The last modification was done in 2003 according to the copyright message.

This work can be found at `http://www.kdevelop.org/`.

#### Eclipse C/C++ development tools (CDT) parser

Eclipse is an IDE written in Java that is very modular. There exists a plugin called the CDT that contains a GNU C++ parser written in Java. Eclipse is distributed under its own open source license.

This parser uses a recursive-descent algorithm. It is mostly a 5 kloc file. The main problem of this parser is that the disambiguation is made by static heuristic. There is then no semantic analysis done at parsing time.

For example `(a)+b` is always parsed as a binary expression, even if `a` was a type identifier.

This work can be found at `http://www.eclipse.org/cdt/`.

#### Edison Design Group (EDG) C++ parser

EDG is a front-end written in C++ as a recursive-descent parser. It is the most widely used parser because of its industrial level of quality. It is also considered the most standard-compliant front-end.

The only drawback is that it is proprietary. Though, it is possible to get an academic license.

#### GCC

GCC has seen its front-end rewritten for version 3.4.0. Now, the parser became hand-written, recursive-descent. It is closer to the ISO standard C++ than the previous parser.

The most seductive aspect of GCC is that it is a free software (GPL) while also a very well maintained project. The parser follows the standard closely. Eventual bugs are tracked. Though free software does not mean reusable. Specially for GCC, it is prevented from being reused. Making GCC more extensible has been discussed on the GCC's mailing list[1]. These ideas do not seem to advance and are kept to the step of discussion. For political reasons, it does not benefit the Free Software Foundation to make GCC reused for other purpose than the compiler itself[2]. Some work has been done to enable plugins with GCC [Callanan et al., 2007]. This work is just a branch in the project for the moment.

However the discussion has not completely ended. There are options for outputing data structures of GCC. Especially `-fdump-translation-unit`. This

[1] `http://gcc.gnu.org/ml/gcc/2007-11/msg00180.html`
[2] `http://gcc.gnu.org/ml/gcc/2000-01/msg00572.html`

prints out an parseable output text of the C++ AST . Also, this format is heavily modified from the original source code: it loses comments, the layout (except the line numbers), constructions are transformed (for example declarations are split), etc.

It is not impossible to reuse GCC, but it is still difficult. It takes a lot of time to be familiar with the structure of GCC [Corbet, 2007] and start modifying it. In addition, we need a lot of maintenance to follow its evolution. Several modifications of GCC exist which shows it is possible.

In case the look ahead is not enough to determine how to parse the stream, the state of the parser is saved. The parse starts to save the stream of tokens. On a parsing error, it is possible to backtrack to the previous context. Parser states are stacked so it is possible to have several layers of backtracking.

Parsing of method bodies inside class specification is done by saving tokens. Function `cp_parser_save_member_function_body` saves a stream of tokens to be parsed. It is really parsed the first time it gets out for the outer most class specifier.

Resolution of names is accessed during parsing which can itself do template instantiation.

This work can be found at `http://gcc.gnu.org/`.

**Puma**

Puma is a stand-alone parser before being integrated into the AspectC++ project.

Puma is based on a recursive-descent algorithm with back-tracking written in C++.

The interesting thing of Puma is the ability to be extended. AspectC++ which is an extension of C++ reuses Puma [Spinczyk and Lohmann, 2007]. The front-end can be actually extended using aspects which is quite elegant. The disambiguation is made at parse time. The semantic analyzer code is still more than 4 kloc. However it can be extended as well by the same principle (and it is extended for the AspectC++ parser).

AspectC++ has a double license: GPL and a commercial license.

Since Puma is integrated to AspectC++, they are maintained together. The last release was made in 2006. A source repository has been opened in 2007 to contain a source that has been updated since 2006, but there was no additional maintenance after the creation of the repository. However the quality of the parser seems to be good.

This work can be found at `http://www.aspectc.org/`.

### 2.3.5 Return of parser generators

The same methods used for Yacc style parsing can be reused with a Generalized LR (GLR) parser generator, making it possible to build an ambiguous tree. The grammar of the C++ language standard does not have to be changed for GLR. Instead of a re-qualification of the code, the bad branches are pruned.

#### TXL

TXL is a full framework for program transformation. Since trees can be dumped into eXtensible Markup Language (XML), it is easy to reuse the grammars from TXL. Unfortunately, TXL does not provide a complete parser for C++. The one provided is a C++-like that is context-free. As a result, lots of contextual ambiguities cannot be handled by its parser. For instance the following program is not parsed correctly. It is parsed as a binary operation instead of a cast with a unary operation.

```
void f() {
  typedef int a;
  int b;
  (a) - (b);
}
```

TXL is not open source, but the syntax definition is given and can easily be modified.

This work can be found at `http://www.txl.ca/`.

#### ANTLR C++ front-end

ANother Tool for Language Recognition (ANTLR) is a parser generator in Java. On the website, we can find a C++ parser provided by David Wigg. This grammar is LL(k), but it does not provide trees or disambiguation. It simply provides the grammar rules. One should write semantic actions for all rules to create a the AST and write disambiguation for it.

This work can be found at `http://www.antlr.org/grammar/list`.

srcML is another front-end using ANTLR to parse C++ [Collard et al., 2003]. The parsing is incomplete and does not parse expressions.

#### Elsa

Elsa is a C++ front-end based on Elkhound. Elkhound is a GLR parser generator. The lexical analysis is based on GNU Flex. The parser generated with

Elkhound outputs an ambiguous PT. The disambiguation is made during the type checking.

The lexer and parser are easily extensible since they are generated and that the parser generator is based on a GLR algorithm which is closed under union.

As said before, the parser generates an ambiguous tree. The type checking is then separated from the parsing and run afterward on the ambiguous tree. Unfortunately, this type checker is a 9 kloc C++ file. The type checker is written on methods belonging to the class of the node in the AST . However its name or signature can vary which makes it hard to approach so as to extend the type checker.

Its development has been stopped since 2005.

This work can be found at `http://www.scottmcpeak.com/elkhound/`.

### 2.3.6 AG and disambiguation

In section 2.3.5, we have seen that some parsers produce an ambiguous tree. Then there should be a pass which prunes bad branches out of the tree. Some front-ends use an attribute grammar system for that purpose. This permits to have a fully extensible parser.

Attribute computation rules can be associated to each derivation rules. This means that we can do an union of attribute grammars. In general, attribute grammars (AGs) are considered modular. For example JastAdd is a modular Java compiler using AGs. Ekman and Hedin [2007] explains how different extensions of Java can be specified reusing the specification of Java 1.4 without intruding in it. Unfortunately, this AG system does not provide the possibility of controlling parsing.

To disambiguate C++ with an attribute grammar, the evaluator needs to be able to do multi-pass evaluation because of the method bodies problem seen in section 2.2.3. The evaluator should also be able to handle code with ambiguities, since synthesized attributes may have different values depending on branches.

#### Semantic Design

Semantic Design provides a C++ front-end with a proprietary license. Akers et al. [2005] claim the tool has been used for large projects. It uses an attribute evaluator. In publications and publicly available documents, it is hard to find the details, but in a private correspondence, Ira Baxter claimed: “We parse context-free, build an ambiguous tree, and then use a following name and type resolution attribute evaluation pass to remove ambiguities that are inconsistent with the language definition. At the end of the name/type pass, the parse tree

is "correct" with all ambiguities removed, and a complete symbol table has been constructed. The symbol table is completely accessible."

#### Transformers

Transformers is a project from LRDE released under GPL. Its goal is to provide a modular C++ front-end. It is based on a home made attribute grammar system over a syntax definition in Syntax Definition Formalism (SDF) and parsed with Scannerless Generalized LR (SGLR). This font-end lacks maturity. While its goal focuses on providing a standard-compliant front-end, it is still incomplete due to lack of funding and human resource.

This work can be found at `http://transformers.lrde.epita.fr/`.

## 2.4 Internal representations and environments

There are several environments available to do source-to-source program transformation with C++. Some provide generalized representation of source code, other specialized for C++. In this section do not see all environments available, but describe some examples.

### 2.4.1 General representations

#### XML

XML is a widely use format for document structure where the data is represented as a tree. For this reason, program source representations can be done in XML.

GCC XML is an extension of GCC. It provides options for dumping trees in XML format. It started to be developed in 2001 even though GCC already had a dump functionality since 1999.

The main problem with GCC XML is that it does not provide tree definitions. It is mainly useful for documentation purposes. The reason for not correctly handling trees is that the funding project did not need it.

gccXfront [Hennessy et al., 2003] and CPPX [Dean et al., 2001] also propose patches of GCC while XOgastan parses the dumps from GCC [Antoniol et al., 2004].

srcML also proposes an XML representation using its own parser [Collard et al., 2003]. The representation is deeper. It does contain the complete source code embedded inside tags. It does not fully parse expressions.

#### TXL

TXL was not originally intended a proper program transformation system, Cordy [2004] claims it was simply for rapid prototyping and experimentation of language features. Rapid prototyping is still the main goal, TXL since gained since some success. But it does not seem to be well prepared for program transformation of C++ since base tools for C++ can hardly be rapid prototypes.

Section 2.3.5 talks about the C++ front-end of TXL.

#### StrategoXT and ASF+SDF

ASF+SDF is a language from the Meta environment van den Brand et al. [2002]. Stratego is a term rewriting language from the environment StrategoXT. Both environments use SGLR parser and syntax definitions are written in SDF [Visser, 1997a].

The internal representation is a term using the ATerm library, which provides maximal sharing van den Brand and Klint [2007], but both environment propose to use the concrete syntax for writing terms [Visser, 2002a, van den Brand et al., 2002].

A front-end for C++ is provided by Transformers and described in section 2.3.6.

### 2.4.2 C++-specific representations

#### Rose

Rose is an infrastructure for transforming C, C++, and Fortran source code [Schordan and Quinlan, 2003]. It is not designed for supporting other languages. The front-end is hard-linked to the system. It is a C++ library providing a representation of C/C++ ASTs.

The tree is fully annotated with semantic analysis results. This makes it a powerful environment for transformations that need a high level of semantic analysis. The rewriting itself is weaker since the semantic annotations are not be updated after the modification of the tree. To update the semantic analysis, the code needs to be output as source code and parsed again. While it is very hard to imagine a semantic analysis that can be locally updated, it should not be required to re-parse from the source. Unfortunately, Rose needs some analysis from the front-end itself.

The other problem is also the tight link of the environment to its front-end, the EDG front-end. Rose is released under Berkeley Software Distribution (BSD) license, but EDG has a commercial license. If we need an extension, you could

extend the AST. It is actually very easy since most of the code of the AST is generated, but we cannot parse it without changing EDG. In theory it is possible to change of front-end, but since Rose relies closely on EDG, it makes it very challenging.

Nevertheless Rose can provide information about pragmas. A transformation like OpenMP is possible for example.

Another question is, in case of change of semantic, or extension, how is it possible to modify the semantic analysis in Rose?

Typical transformations are written for Rose with traversals using visitor patterns.

#### Pivot

Pivot is also another library for C++ program representation similar Rose. It is also implemented on EDG but relies less on it. The interesting point of Pivot is the regularisation of the representation. This representation is called Internal Program Representation (IPR) [Stroustrup and Reis, 2005].

Pivot is simpler than Rose, but also newer and does not provide as many tools.

## 2.5 Conclusion

There are three different approaches to parsing C++. The first, based on LALR parsing is commonly abandoned since the C++ 98. The practical approach is usually writing a recursive descent parser by hand, while some try to propose solutions proposing declarative formalism.

Reusable front-ends which are fully compliant to the standard are rare. The maintenance often stops after few years. The possibility of extension is usually not present when the front-end is standard compliant.

Full transformation environments either propose language-dependent infrastructure which can be hardly extended. However they are more reliable whereas language-independent environments are not well tested on C++.

# Chapter 3

# Modular specification for a context-sensitive syntax

Of all the techniques for parsing C and C++, one consists of parsing the syntax exactly as described in the syntax definition of the standard using a Generalized LR (GLR) parser and giving a tree as output containing several branches for each ambiguity. Then a filter prunes the wrong branches to give a clean tree corresponding exactly to the syntax definition of the standard.

In the following paper, this technique is used combined with an attribute grammar (AG) system. The reason for this is modularity. GLR parsers handle all ambiguous context-free grammars: context-free grammars are close under union. This means that extending a GLR grammar is pretty easy. Unfortunately, in our case, we need a disambiguation filter. To cope with the problem of union, this paper proposes to define semantic rules for disambiguation defined with each context-free rule.

This paper was presented at ICCP'06.

# Attribute Grammars for Modular Disambiguation

VALENTIN DAVID AKIM DEMAILLE OLIVIER GOURNET

*In the proceedings of IEEE 2nd International Conference on Intelligent Computer Communication and Processing, Technical University of Cluj-Napoca, Romania 1 - 2 September, 2006*

**Abstract.** To face the challenges to tomorrow's software engineering tools, powerful language-generic program-transformation components are needed. We propose the use of attribute grammars (AGs) to generate language specific disambiguation filters. In this paper, a complete implementation of a language-independent AG system is presented. As a full scale experiment, we present an implementation of a flexible C front-end. Its specifications are concise, modular, and the result is efficient. On top of it, transformations such as software renovation, code metrics, domain specific language embedding can be implemented.

## 2.5 Introduction

Modern software engineering tools provide the programmer with a host of powerful features to manipulate source code. A trend to design such tools consists in building them from language-generic components: generic parsers, program transformation environments and pretty-printers, for instance, exist. It is then possible to provide a language with new tools (code metrics, refactoring environments, etc.) or new features (embedded SQL, design by contract, etc.).

In such a framework the parsing is truly context-free, but most programming languages are *not* context-free. To cope with this discrepancy, parsers actually accept an ambiguous superset of the language, leaving the context-sensitive disambiguation phase to a later filter. This paper presents a new approach

**© 2006 IEEE. Reprinted, with permission, from the proceedings of IEEE 2nd International Conference on Intelligent Computer Communication and Processing, Attribute Grammars for Modular Disambiguation, Valentin David, Akim Demaille, Olivier Gournet**

based on an extension of AG [Knuth, 1968] to cope with ambiguity. We report on an implementation of such an AG system, and demonstrate the soundness of the approach by describing a complete ISO C 99 polyvalent front-end. Thanks to the use of AGs, its implementation is both concise and modular. Both the AG engine and the C front-end are free-software covered by the GNU General Public License (GPL), freely available on the Internet LRDE — EPITA Research and Developpement Laboratory [2005].

The paper is structured as follows. section 3.1 presents the context of our work, and motivates the use of ambiguous AGs for the automatic generation of semantics driven disambiguation filters. section 3.2 details the user side: how to run a disambiguation chain powered by an AG. section 3.3 presents the implementer side: how to write disambiguating AG rules, and how the filter is generated. section 3.4 reports about a full scale use of these tools to implement and use a C flexible front-end. Existing and future works are presented in section 3.5. Finally section 3.6 concludes.

## 3.1 Context

**Generalized Parsing** Even though LALR(1) parsing ruled the world thanks to Yacc, truly context-free languages are infrequent: most are context sensitive. This dependency is usually addressed with ad hoc actions in the parser such as symbol table maintenance, which prevents any form of modularity. Because in addition no interesting class of deterministic languages is stable under union, techniques supporting the full class of context-free languages are desirable. GLR parsing [Tomita, 1985] meets these requirements.

GLR handles local ambiguities “for free” using unbounded look-ahead, and global ambiguities (requiring context sensitive information such as typing) typically by providing the user with a means to decide how to process alternatives. GLR parser generators such as Elkhound [McPeak, 2002] or GNU Bison extend the Yacc model: user actions are executed. Using tailored actions during parsing enables excellent performances, comparable to usual parsers, but using similar tricks. Then again modularity is lost: user actions need to be modified when mixing several languages.

Alternatively some generated parsers directly build the parse tree (PT) / abstract syntax tree (AST). Scannerless Generalized LR (SGLR) [Visser, 1997c] is one such tool; if the input is ambiguous it yields a parse forest that a latter pass is expected to prune (section 3.2).

**Flexible Front-ends** State of the art technologies maximize reuse, including of front-ends. This is in sharp contrast with the traditional front-ends that embed actions tailored to the job at hand: even if at some time the grammar was taken from a grammar base, hand editing made it diverge from its root. Using a formalism such as Syntax Definition Formalism (SDF) [Visser, 1997b] together with a generic parser such as SGLR [Visser, 1997c] makes it possible to design a library of program transformation components: parser, pretty-printer etc. The grammar behaves as a contract for the whole tool-chain [de Jonge and Visser, 2001]. These components can be used off-the-shelf, or modified in a modular way for a specific task.

Flexible front-ends unleash a host of new possibilities: domain-specific languages (DSLs) can be embedded in host languages, local idioms can be normalized (e.g., translation from GNU C to ISO C), etc. The MetaBorg method [Bravenboer and Visser, 2004, Bravenboer et al., 2006a] uses Dryad, a flexible Java front-end, together with a toolchain to implement *assimilation*, i.e., the compilation of extended Java down to regular Java. The authors demonstrate MetaBorg by hosting within Java a DSL for Swing interface design, another for regular expressions, and another for Java abstract syntax tree (AST) handling.

Unfortunately no such framework is available for C, let alone for C++. The Transformers project LRDE — EPITA Research and Developpement Laboratory [2005] aims at providing C (and eventually C++) flexible front-ends. Transformers is free-software covered by the GPL, freely available on the Internet.

**Tools for Program Transformation** Flexibility results from tight common conventions between the tools, we chose SDF as the spine for the Transformers project and the Stratego/XT tool set [Visser, 2004]. This collection of tools provides language generic components for the whole processing. In particular the Stratego programming language [Visser, 2001] provides powerful term rewriting features, controlled and composed by rewriting strategies. A rich set of operators allows to build arbitrarily complex transformations (i.e., strategies) from simple atomic ones. Context sensitive transformations are easily coped with thanks to dynamic (rewriting) rules. Finally, thanks to a tight integration with SDF, Stratego features concrete syntax: although rewriting rules do transform *abstract* syntax trees, rules can be written in the target's language *concrete* syntax [Visser, 2002a].

Although C and C++ front-ends already exist, two foundamental limitations prompted the design of an SDF/SGLR support for C and C++: firstly no other formalism features an equivalent level of modularity, and secondly this is mandated to enjoy the benefits of C or C++ concrete syntax in Stratego.

```
context-free syntax
  "true"          -> Bool
  "false"         -> Bool
   Bool "|" Bool  -> Bool
```

This grammar is taken from van den Brand et al. [2003]. Contrary to the Backus-Naur form (BNF) tradition, in SDF arrows are oriented as reductions, not productions.

Figure 3.1: A simple ambiguous SDF grammar.

## 3.2 Disambiguation chain

The decomposition of the analysis of context-sensitive languages in two steps, (context-free ambiguous) parsing and then disambiguation, is well known and well described in the literature [Klint and Visser, 1994, van den Brand et al., 2003, Bravenboer and Visser, 2004]. The main contribution of this paper is demonstrating that AGs, extended to support ambiguities, provide a nice way to implement such filters.

As a running example, consider the simple ambiguous SDF grammar from Figure 3.1. The associativity of `|` is unspecified, leaving two possible syntactic analysis of `true | true | true`. Such an ambiguity can be handled during the construction of the parser. Nevertheless we use it because it is extremely simple and already discussed by van den Brand et al. [2003]. Vasseur [2004] discusses a more complex yet simple ambiguous language (subsection 3.5.1).

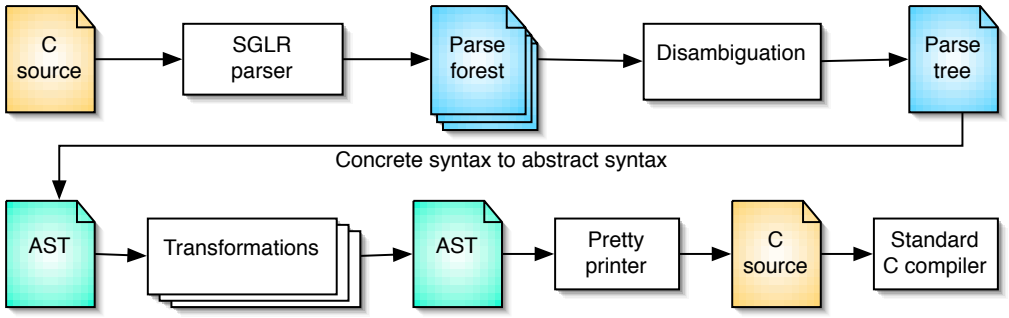
The following sections follow the stream depicted in Figure 3.2 from the original source text, down to the possibly transformed final source text.

### 3.2.1 Upstream

Before attribute evaluation several phases may have been run. For instance, if we were to process C or C++, one would have to run the preprocessor to handle all the # directives and macros. SGLR might be run by this phase, but whatever the architecture of the upstream phase, it must result in a (possibly ambiguous) PT.

In our running example, the analysis of a simple phrase such as `true | true | true` yields an ambiguous PT, which represents the two possible analysis[1]: `(true | true) | true` and `true | (true | true)`. In the fol-

[1]The parentheses are not part of the language under study, they are used as meta-notation to




Given a C grammar written in SDF, the SGLR parser reads the text and yields a set of parse trees: a parse forest. A disambiguation step keeps a single parse tree (section 3.2), transformed into an AST suitable for the transformation(s). Finally the AST is pretty-printed back into compiler-ready C source text.

Figure 3.2: A C program transformation chain

lowing, we chose the former, corresponding to the left associativity.

### 3.2.2 Attributes

We propose the use of AGs to specify context-sensitive rules. attribute grammars (AGs) [Knuth, 1968] is a formalism that supports syntax directed semantic analysis: (grammar) rules are decorated with a set of equations that relate a node's *attributes* with those of its parents and/or children. AGs allow to focus on local aspects, leaving the global evaluation order aside, under the responsibility of a generic engine. Although AGs cannot modify the trees, their use for disambiguation is straightforward. Attributes convey information, e.g., a symbol table. Conflicting branches of the parse forest are flagged, and a (language generic) filter is run afterward on the parse forest, pruning inconsistent alternatives.

**Attribute Evaluation.** In our implementation, rules relating attribute values are attached to the SDF grammar as *annotations* of type *attributes*. In the running example, left associativity expressed locally to a `|`-node stands as "no right-child of a `|`-node is a `|`-node". The following example presents a straightforward implementation of this idea.

```
context-free syntax
```

distinguish the alternatives.

```
"true" | "false"      -> Bool
   {attributes(assoc:
     root.is_atom := true
   )}

  b1:Bool "|" b2:Bool -> Bool
   {attributes(assoc:
     root.is_atom := false
     root.ok      := b2.is_atom
   )}
```

The symbol `root` always denotes the root of the node: the right-hand side of the (SDF-)rule. The user may use labels such as `b1`/`b2` to refer to symbols. A single attribute `is_atom` is computed for each node. The special attribute `ok` specifies whether the node is valid or not.

Technically the computation of the attributes is performed by the *evaluator* compiled from an AG. Its construction is presented in section 3.3. It runs on a PT and yields an *attributed* PT, i.e., a PT whose nodes are decorated by attribute values. It is thanks to the extreme flexibility of the PT format, AsFix2, and the tools that process it that we can decorate it so easily.

The evaluation of an *ambiguous* AG is somewhat different from the usual case because a synthesized attribute could have different values on different branches of an ambiguity. To select the right one, the evaluator depends on the `ok` attribute. Indeed, directly or indirectly, every synthesized attribute should depend on an `ok`. If `ok` is `false`, the value of synthesized attribute is `fail`, which propagates to "infected" attribute values. In the end, the ambiguity node is presented only with a set of at most one possible value; if it turns out all the values `fail`, the ambiguity node's attribute itself is set to `fail`.

This design frees the user from having to explicitly carry the `ok` attribute everywhere. It is comparable to an exception system which destroys a branch of a computation until an exception handler is met: an `amb` node.

**Pruning & Checking.** The evaluation phase is pure: it does not modify the PT, it merely decorates its nodes. In particular it flags invalid alternatives of an ambiguous PT with a false `ok` attribute. A filter then prunes invalid branches, and afterwards another makes sure no ambiguity remains.

### 3.2.3 Downstream

Although they fall outside the scope of this paper, it is interesting to introduce passes that are typically run once an unambiguous PT is obtained. *Implosion* transforms the PT into an AST: basically nodes are no longer labeled by their

corresponding production rule, but by simple labels, *constructors*, also specified as addition SDF rule annotations. For instance a PT node labeled by `Bool | Bool -> Bool` will implode into an AST node labeled by a simple constructor `Or`. This AST is the ideal format to run *program transformations* (section 3.1). Such transformations include software renovation, assimilation, and so on. Alternatively data can be extracted, such as code metrics. Finally, the AST is pretty-printed back into a source file.

## 3.3 Attribute Grammars in SDF

The generation and compilation of an AG evaluator for a specific grammar comprises several steps detailed below. A set of SDF modules with AG rules (subsection 3.3.1) are packed together and preprocessed (subsection 3.3.2), its AG rules are checked and completed (subsection 3.3.3), and finally compiled into an evaluator and a parse table (subsection 3.3.4).

### 3.3.1 Attribute Rules in SDF

From the point of view of syntax, there are many available options to specify attribute computation rules. Because developing an AG system for SDF was not our primary goal, we made a number of decisions to simplify the design. In the long run, several of these shortcuts should be reconsidered (subsection 3.5.2).

We embed attribute rules in the SDF grammar. The rationale is that since we use AG to disambiguate an ambiguous grammar, the disambiguation information really belongs to the grammar itself. As a matter of fact, this is not different from embedding precedence and associativity information in the grammar, which is already the case in SDF. Actually SDF features several other disambiguation types of built-in filters [van den Brand et al., 2002].

Rules are embedded as regular SDF annotations, therefore we can use the usual SDF tool set: no additional development and maintenance is needed. As a natural consequence, our disambiguation filters benefit from the exact same concept of modularity as the SDF modules themselves. We can freely mix SDF modules: the disambiguation AG taking care of context-sensitivity is also intermixed. Of course, additional disambiguation rules might be needed.

### 3.3.2 Grammar Modules Preprocessing

Before generating the attribute evaluator several preprocessing phases take place. First, `pack-esdf`, an extended version of the `pack-sdf` tool from Stratego/XT,

gathers and checks SDF grammar dependencies, and produces a self-contained unique grammar file. Some tweaks are also used to transform this unique file into a form suitable for `sdf2table`, the off-the-shelf parse-table generator. These parse tables are extremely rich and contain all the data needed to generate an AG evaluator: the production rules, the symbols labels, and all the annotations, including attribute rules.

### 3.3.3 Completion and Checking

The `attr-defs` tool processes these parse tables to implement a number of features such as automatic propagation. Indeed, some attributes such as symbol tables virtually traverse the whole PT; writing down their propagation is tedious and error prone. Support for synthesized (bottom up), inherited (top down), and chain (left to right) attributes is implemented. Without such a feature one could no longer benefit from operators such as `*`, `+`, `|` and so forth: one would have to decompose into a set of plain syntax rules spelling out the detail of the propagation of attributes. In other words, without such a feature no AG system can pretend to be modular.

Cycles in attribute dependencies are looked using an algorithm from Knuth [1968]. Because `attr-defs` is basically a graph cruncher, a data type with which Stratego is uneasy, it was rewritten in C++.

### 3.3.4 Evaluator Generation

The last module, `attrc`, handles two issues. First it removes all the AG related data to produce parse tables as expected by SGLR. Second and foremost, it generates the attribute evaluator.

This evaluator uses a strategy based on attribute dependencies to compute the order of evaluation. In this respect laziness is a nice feature that virtually determines the evaluation order by itself, a fact used in the implementation of Utrecht University Attribute Grammar system (UUAG) in the Haskell lazy functional programming language [Baars et al., 1999]. To benefit from laziness, we generate a single (huge) Stratego program compiled by a Stratego compiler modified to support a weak form of laziness. This weak support consists in implementing all the attribute values as functions with memoization: the first time its value is requested, it is computed and cached for subsequent calls.

## 3.4 Case Study: C-Front

As a real world experiment of our AG system, we report its use to disambiguate ISO C 99. Although C is one of the most used languages, there are few flexible front-ends usable off-the-self. The front-end we describe aims at filling this gap for SDF users.

### 3.4.1 An SDF attribute grammar for C

In order to enable the experimentation of extensions to the standard and to minimize the risk of recognizing a language very close to, but different from the standard, we chose to remain close to the grammar as specified by the C standard [WG14, 1999]. As a consequence the style is sometimes a bit convoluted and unnatural: not all the SDF features are used; for instance precedence and associativity are encoded in the grammar via additional non-terminals and rules. This results in bigger ASTs.

The C grammar counts 126 symbols and 356 rules, split into 53 small and manageable sub-grammars. The boundaries of these sub-grammars were chosen to address coherent, atomic, related issues; they are finer than those of the standard that breaks the grammar in only 4 parts [WG14, 1999, Annex A]. The AG part of the grammar counts 10 different kinds of attributes, and 190 attribute rules. The completion of attribute rules with automatic propagation raises that count up to 1183 (subsection 3.3.3).

### 3.4.2 Performance

Efficiency is measured in two different contexts: the compilation of an AG, and then its execution.

**Compile time.**

The compilation of the evaluator is slow but bearable: three minutes (Figure 3.3). Interestingly a significant portion of the computation time is spent uselessly in pretty-printing (aka, unparsing) at the end of a tool, immediately followed by parsing by the following tool. This will be easily solved by the authors of the tools, since they all share the compact binary representation of trees, ATerms [van den Brand et al., 2000].

The resulting front-end passes all its test suite, composed of about 800 tests from the GCC test suite (the tests that were left out address either GNU extensions, or issues not related to parsing) plus 100 additional tests tailored to exercise the disambiguation.

**Run time.**

| Filter | Duration (s) |
|---:|---:|
| `pack-esdf` | 3.12 |
| misc. processing | 9.31 |
| `sdf2table` | 7.28 |
| `attr-defs` | 0.41 |
| `attrc` | 10.08 |
| Stratego compilation | 112.95 |
| C compilation | 32.35 |
| **Total** | **175.54** |

The experiment was run on a P4 3Ghz with 1Gb RAM. The memory footprint is below 120Mb.

Figure 3.3: Compile time performances

Finding large programs in *standard* C, to bench our disambiguating chain, turned out to be troublesome. In particular, GNU C extensions are fairly commonly used; its support in our C front-end is future work.

The figures are both reassuring, and disturbing. Indeed, the run time of the disambiguation filter is acceptable in most situations, and we are actually confident that several ad hoc optimizations could significantly cut it down. Unfortunately the performance of the rest of the tool-chain is harder to improve. As a matter of fact, the slowness of SGLR has already been reported [McPeak, 2002]. It has been said that in the future SGLR might directly produce an AST instead of a PT, a much smaller data structure. More ideas for speed improvements are proposed in subsection 3.5.2.

## 3.5 Discussion

### 3.5.1 Other works

**Algebraic Specification Formalism + Syntax Definition Formalism (ASF+SDF)** [van den Brand et al., 2001] is a complete environment that parallels SDF grammar modules with ASF equation modules. ASF is a declarative term rewriting language featuring concrete syntax, conditional rewrite rules, and traversal functions. Given an ambiguous grammar, ASF can be educated to process its extension with `amb`(iguity) nodes. Then algebraic specifications prune invalid alternatives. This setup is presented by van den Brand et al. [2003]. Their approach shares several features with ours, most prominently declarativity. Indeed ASF equations are very comparable to attribute rules: one focuses only on the

local computation, leaving the global evaluation order aside. Some modularity follows as a natural consequence, but with performance issues.

In **Stratego**, rewriting strategies allow to specify evaluation orders. While strategies are comparable to ASF+SDF's traversal functions, the concept is pushed much further. Strategies are explicit and programmable: starting with a set of primitive strategies and strategy combinators, the user can design higher level strategies. Early experiments of a C++ disambiguation filter in Stratego demonstrated that it is able to tackle its most difficult part, that related to `template`. Since then, benefiting from more recent features of Stratego (in particular concrete syntax and dynamic rewriting rules), Dryad, the Java front-end developed at the Utrecht University, demonstrated that the approach proposed by the MetaBorg method [Bravenboer and Visser, 2004, Bravenboer et al., 2006a] is very successful and efficient.

In both cases the user has to spell out the traversal order, either by choosing it, or programming it. AGs frees the developer from this task, which also enables an attractive form of modularity: composing two components will create a whole new traversal order, unrelated to the two "primitive" traversals. This order is also naturally efficient. A more extensive comparison similar to Vasseur [2004] is underway.

### 3.5.2 Future Work

**Syntax.** The current syntax can be improved in many ways. Probably foremost, the fact that it is physically bound to its production rule goes against the separation of concerns. This simplification is acceptable for our current application — disambiguation — but if new applications, such as type-checking, were to be developed, they would add clutter to the grammar. A better designed DSL should also include support to declare attributes, declare their special properties (such as default propagation rules) etc. much as is done in UUAG [Baars et al., 1999].

In parallel some idioms could be isolated for frequent types of disambiguation schemes, for instance the dependency on the kind of an identifier (a type name? a value name?), and a dedicated syntax could be submitted.

**Performance.** The current run time enables to experiment C program transformations, explored in Borghi et al. [2006]. Several minor optimizations can probably cut down the execution time, nevertheless we believe that a better set up would be to evaluate the attributes during parse time. While it is clear that parsers that automatically build an PT or AST are an improvement over hand-written user actions à la Yacc, it is also a significant loss in performance: none of the authors believes this technique will ever be able to compete with indus-

trial strength compilers such as the GNU C++ Compiler that includes parse-time disambiguation actions. The true added value is in *modularity* and *separation of concerns*, not the *physical* separation of phases. Provided modularity and separation of concerns are available there is no reason to banish the possibility to generate a tailored parser. The benefits are immediate: because most real-world ambiguities are solved by a simple left-to-right reading, most of the ambiguous nodes would not even be built. The peak of memory consumption, the size of the graph the evaluator would have to compute, would both become much smaller, henceforth, much faster to process. As an example out of 68 195 ambiguities in *Eval* in Figure 3.4 no less than 58 460 are due to value identifiers that can either denote a variable name, or a enum value. This possibility to disambiguate at parse time is specific to AGs. This requires a complete rewrite of SDF, a whole topic in itself.

## 3.6 Conclusion

We have proposed a new declarative approach to the specification of context-sensitivity using ambiguous attribute grammars. We report about the implementation of a such tool well integrated in SDF frameworks such as ASF+SDF or Stratego/XT. This tool was used to implement a fully C standard compliant flexible front-end. These experiments demonstrated that attribute grammars are very well suited to the generation of disambiguation filters. In particular, their specifications are very concise, and modular. The run time performance of the whole system are very satisfying, enabling the handling of C in any SDF powered transformation framework.

**Acknowledgments** The authors thank Karl Trygve Kalleberg for his comments on earlier drafts.

| | **Queens** | | **HelloW** | | **Lemon** | | **Eval** | |
|---|---|---|---|---|---|---|---|---|
| **Lines of code** | 56 | | 448 | | 4 135 | | 28 392 | |
| **Ambiguities** | 78 | | 103 | | 6 410 | | 68 195 | |
| **PT Sizes** | Tree | Mem | Tree | Mem | Tree | Mem | Tree | Mem |
| **Ambiguous** | 10 | 14 | 63 | 62 | 912 | 642 | 10 280 | 5 689 |
| **Attributed** | 9 | 28 | 109 | 403 | 813 | 2 884 | 7 625 | 19 661 |
| **Final** | | 15 | | 162 | | 1 104 | | 5 497 |
| **Duration** | s | % | s | % | s | % | s | % |
| **Preprocessing** | 0.09 | 29 | 0.2 | 2 | 0.2 | 1 | 0.2 | 1 |
| **Parsing** | 0.04 | 12 | 1.3 | 33 | 4.8 | 17 | 33.6 | 10 |
| **Concatenation** | 0.01 | 3 | 0.7 | 18 | 1.5 | 5 | 3.2 | 1 |
| **Evaluation** | 0.07 | 22 | 0.8 | 20 | 11.4 | 40 | 177.6 | 55 |
| **Pruning** | 0.05 | 16 | 0.5 | 13 | 5.6 | 20 | 60.4 | 18 |
| **Cleaning** | 0.01 | 3 | 0.2 | 4 | 1.5 | 5 | 12.2 | 3 |
| **Checking** | 0.01 | 3 | 0.1 | 1 | 0.6 | 2 | 4.6 | 2 |
| **Conversion** | 0.03 | 9 | 0.2 | 3 | 2.1 | 7 | 29.3 | 9 |
| **Total** | 0.31 | 100 | 3.8 | 100 | 28.0 | 100 | 322.5 | 100 |

*Queens* is an extremely short program that includes no header. *HelloW* is the simple "hello, world" program with an `#include <stdio.h>`. *Lemon* is a parser generator that fits in a single C file. *Eval* is the AG evaluator for C generated in C by the Stratego compiler. The number of lines was computed by `sloccount`. The sizes of the parse tree count the total number of nodes and its memory footprint in Kb; the measurements were performed before the disambiguation (*Ambiguous*), after the disambiguation and pruning (*Attributed*), and after the removal of the attributes (*Final*). The timings were performed on a P4 3Ghz with 3Gb RAM.

Figure 3.4: Running time of the C-Transformers chain

# Chapter 4

# Extending a context-sensitive syntax

In chapter 3, we detailed how we can do disambiguation of context-sensitive syntax based on an ambiguous context-free specification using an attribute grammar (AG) to prune invalid branches from the parse forest.

Using this kind of formalism is very useful to allow extension of a context-sensitive syntax. This article demonstrates the modularity with a simple extension of language.

The extension adds pre- and post-conditions in C. Very few rules are added to the C syntax definition. A tool written in Stratego then translates from the extended C to pure C.

This article was published in ACM Crossroads.

# C-Transformers: A Framework to Write C Program Transformations

ALEXANDRE BORGHI VALENTIN DAVID AKIM DEMAILLE

*Crossroads, 12.3, Programming Languages Spring 2006*

**Abstract.** Program transformation techniques have reached a maturity level that allows processing high-level language sources in new ways. Not only do they revolutionize the implementation of compilers and interpreters, but with modularity as a design philosophy, they also permit the seamless extension of the syntax and semantics of existing programming languages. The C-Transformers project provides a transformation environment for C, a language that proves to be hard to transform. We demonstrate the effectiveness of C-Transformers by extending C's instructions and control flow to support Design by Contract. C-Transformers is developed by members of the LRDE: EPITA undergraduate students.

## 4.1 Introduction

New trends in programming languages set a new challenge to the researcher: productivity. One trend focuses on providing the programmers with more productive languages, to this end *program transformations* techniques are extremely powerful. To implement these transformations, *language-specific frameworks* are needed. To compose these frameworks, *language agnostic tools* are needed that can be used to quickly address a particular issue in whatever language.

**Program transformations** encompass virtually every type of program processing. **Data extraction** from sources allow to perform code metrics (number of lines, statements, function arguments, maintainability etc.), to generate documentation. **Software renovation** consists in improving existing code, possibly very large and ancient programs. Even when a thorough design was made,

**© 2006 Association for Computing Machinery, Inc. Reprinted by permission from Crossroads, 12.3, Programming Languages Spring 2006, C-Transformers: A Framework to Write C Program Transformations, Alexandre Borghi, Valentin David, and Akim Demaille**

when experienced programmers are involved, when high level development tools are used etc. refactoring is needed to keep a program healthy: maintainable. Programmers are used to refactor by hand, but today much can be mechanized, including the use of advanced constructs such as Design Patterns [Ziane, 2001]. The development of such refactoring tools for Integrated Development Environments is an active topic of research. **Optimization** is another active field, including domain-specific optimizations [Bagge et al., 2003] and standard optimizations, possibly very high level ones such as partial evaluation [Olmos and Visser, 2002]. **Translations**, such as compilation, typically make use of program transformation techniques, with several refining steps. Most of the time transformations are written in a general language (C, C++, Caml...), but some exploit dedicated techniques (e.g., Tiger in Stratego [Visser, 2002b], or even the Stratego Compiler itself [Visser, 1999]). As a specific instance of translation, **language extension** provides an existing language with additional features and write a translator that "compiles" (**assimilation** in the words of Bravenboer and Visser [2004]) the extended language down to the host language. As a running example, this paper will focus on such an application.

**Language-specific frameworks.** Writing a compiler is a tremendous task, and the implementation of a transformation framework makes no exception: one needs (i) a parser to read the input, (ii) possibly a disambiguation step, (iii) optionally a type checker, (iv) the core transformations themselves, and finally (v) a pretty printer to convert the program back to text. Every step of this framework is language-specific. The infrastructure can outweigh the transformation by far, therefore, in order to make the transformation implementation productive, language-specific frameworks are needed. In this paper we present the C-Transformers project LRDE — EPITA Research and Developpement Laboratory [2005], a framework enabling the seamless implementation of transformations programs in C— or C variations.

**Language generic components.** Vast research and many tools were developed to provide the technology needed to implement each type of component of the framework, sometimes leading to nice generalizations. It is well known for instance, that there are tools that generate parsers from grammars; it is less known is that other steps also enjoy the existence of language generic components that can easily be tailored to a specific language [de Jonge and Visser, 2001]. C-Transformers is using Stratego/XT [de Jonge et al., 2001] [Visser, 2004] as its library of language generic components. These off-the-shelf tools allow us to focus directly on C-specific issues (e.g., its disambiguation).

The C-Transformers is a free software project available on the Internet LRDE — EPITA Research and Developpement Laboratory [2005] developed by EPITA

undergraduate students. EPITA is a French private engineering school dedicated to computer science. The Research and Development Laboratory of EPITA, LRDE, recruits amongst the best students willing to follow a more academic curriculum, possibly leading to a PhD. While members of the LRDE, they work on research topics supervised by assistant professors. V. David and A. Borghi are members of this group working actively on disambiguation under the supervision of Akim Demaille.

C-Transformers can be used to implement virtually any kind of C program transformation, but currently, because of its lack of a reversible pre-processor (section 4.6) it cannot be used for software renovation.

The Olena [Duret-Lutz, 2000, EPITA Research and Developpement Laboratory (LRDE), 2009] and Vaucanson [Lombardy et al., 2003, VAUCANSON group, 2007] projects gave the LRDE a strong experience in the development of C++ fast and generic libraries. New C++ programming techniques were then invented Burrus et al. [2003], unfortunately resulting in somewhat obfuscated code. The Transformers project was created to address this issue, for instance by adding syntactic sugar to C++.

The structure of this paper is as follows. section 4.2 presents similar projects. section 4.3 introduces the C-Transformers chain components. In section 4.4, to demonstrate its use, this C generic chain will be used to implement an extension of C, from its parsing down to its compilation to (ISO) C. section 4.5 is devoted to deeper discussion on the results. Then section 4.6 concludes and presents leads for future work.

## 4.2 Related Work

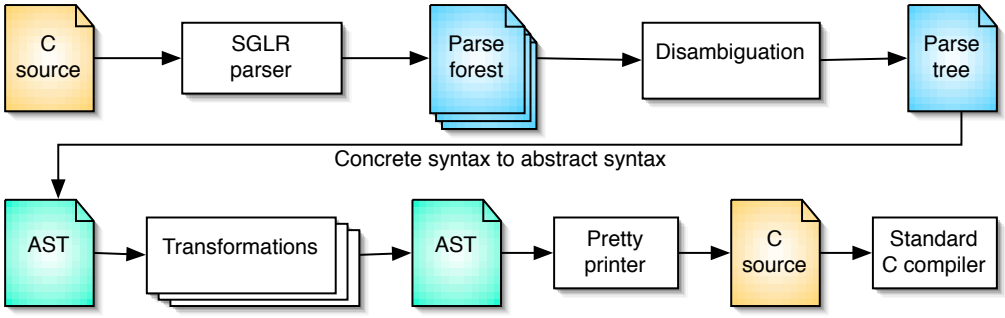
Program transformation draws a lot of attention these days, with several existing projects. Three of them are specially close of ours, CIL addresses ISO C, METABORG has the exact same goals but focuses on Java, and Proteus share goals, techniques, and target language.

**CIL** Necula et al. [2002], Necula [2005] is an extremely complete and mature front end for the C language. It also includes a type checker, and a normalization of C towards a clean and simpler subset of C. This considerably eases the transformation of C programs by reducing the number of cases to handle. Nevertheless CIL does not (and cannot) provide some of the features that prompted the development of C-Transformers. (i) Transformations for CIL are more naturally expressed in Caml, the language it is written in. This does not prevent using CIL as a front-end to a Stratego program, but glue code is needed. (ii) CIL's grammar is hardwired and thus cannot be easily extended, especially not in a modular

way. (iii) Using an Syntax Definition Formalism (SDF) grammar and using SDF tools is mandated to enjoy concrete syntax in Stratego. (iv) CIL's output is not "syntactically faithful", i.e., the output program is semantically equivalent but it is a deep modification of the input program, with a complete loss of layout, comments, and preprocessor directives. CIL is inadequate for code factoring.

In the words of its authors [Bravenboer and Visser, 2004, Bravenboer et al., 2006a], "MetaBorg provides generic technology for allowing a host language (collective) to incorporate and assimilate external domains (cultures) in order to strengthen itself. The ease of implementing embeddings makes resistance futile". Basically the MetaBorg project has exactly the same purpose as the Transformers project, but on a different host language: while Transformers focuses on C and C++, MetaBorg started with Java. Java is a much cleaner language than C, let alone C++ for which the development of a parser and disambiguation filter proved to be a daunting task. The MetaBorg chain relies on the exact same tools except for semantics driven disambiguation, which is written directly in Stratego (described in 4.3.4). In Bravenboer et al. [2006a] the power of the system is demonstrated by several extensions of Java to include domain-specific languages (DSLs): seamless syntax and semantics for XML, Swing, and even Java programs. Such a tour de force was made possible thanks to the modularity of the suite (section 4.5). Contrary to Transformers they already have a type checker, which allows to simplify the syntax even further by taking types into account during the disambiguation.

The goals of the **Proteus** [Waddington and Yao, 2005] are extremely similar to Transformers': building a C transformation framework that makes it possible to preserve the programming style. These projects share many tools (SDF, Scannerless Generalized LR (SGLR), Stratego), but they differ on some aspects. We strictly adhere to the standard grammar, they tailored theirs; we have a workable solution to disambiguate C and extended C abstract syntax trees (ASTs), but their paper does not mention disambiguation; we stick to Stratego with C concrete syntax, they introduce an other language, YATL, compiled into Stratego; we explicitly want to experiment grammar extensions, they focus on standard C program refactoring, finally Transformers is free software [LRDE — EPITA Research and Developpement Laboratory, 2005]. The initial development effort also differs: Transformers attacks the modular disambiguation of C (and C++) first, and Proteus first made sure they can preserve the programming style — not only comments and layout, but also preprocessor directives.




Given a C grammar written in SDF (subsection 4.3.1), the SGLR parser reads the text and yields a set of parse trees: a parse forest (subsection 4.3.2). A disambiguation step keeps a single parse tree (subsection 4.3.3), transformed into an AST suitable for the transformation(s) (subsection 4.3.4). Finally the AST is converted back into compiler-ready C source text, a process named pretty-printing (subsection 4.3.5).

Figure 4.1: The C-Transformers chain

## 4.3 The Transformation Chain

In this section we present the C-Transformers framework, component by component. Figure 4.1 schematizes the whole process.

### 4.3.1 SDF Grammars

The Stratego/XT tool set uses SDF [Visser, 1997b] as its backbone: the tools are parametrized by an SDF grammar specifying the language at hand. In other words, grammars are contracts [de Jonge and Visser, 2001], therefore it must be carefully crafted. This grammar syntax is modular, which improves maintainability and extensibility, by splitting the grammar in several modules. It is also extensible: additional information can be packed in the grammar via annotations.

Roughly the grammar can be written following two guide lines. It can be designed to be simple, making full use of SDF capabilities to handle precedence, etc. This is an attractive way since it results in rather short and elegant ASTs. We made "the" other choice: to stick rigorously to the grammar defined in the ISO C standard [WG14, 1999] in order to guarantee our strict compliance with the standard, and to provide an environment of choice to experiment extensions to the standard (which is precisely the theme covered in section 4.4). As a result our ASTs are somewhat more convoluted — for instance the AST for a simple `return 42;` is 26 nodes deep.

The C grammar counts 126 symbols and 356 rules. To ease the maintenance, the grammar is split into 53 small, manageable, sub-grammars. The boundaries of these sub-grammars were chosen to address coherent, atomic, related issues; they are finer than those of the standard which breaks the grammar in only 4 parts [WG14, 1999, Annex A].

Figure 4.2 demonstrates some of SDF features.

The running example of section 4.4 will extend `Declarator` to support an additional form of function declaration (subsection 4.4.2).

### 4.3.2 SGLR and Parse Forests

A technology supporting ambiguity and yielding parse forests is needed. Amongst available techniques Generalized LR (GLR) is most attractive [Tomita, 1985]. Not only does a generalized parser relieve us from obfuscating the grammar to cope with the limitations of the parsing technology — such as the infamous shift/reduce or reduce/reduce conflicts — it is actually indispensable to have the necessary level of modularity (see section 4.5).

```
module Declarators
imports ConstantExpressions TypeQualifiers ParameterDeclarations
exports
  sorts Declarator DirectDeclarator Pointer PointerSeq
  context-free syntax
    PointerSeq? DirectDeclarator              → Declarator
    Identifier                                → DirectDeclarator
    "(" Declarator ")"                        → DirectDeclarator
    DirectDeclarator "[" TypeQualifierList?
                         AssignmentExpression? "]"
                                              → DirectDeclarator
    DirectDeclarator "[" "static" TypeQualifierList?
                         AssignmentExpression "]"
                                              → DirectDeclarator
    DirectDeclarator "[" TypeQualifierList
                       "static" AssignmentExpression "]"
                                              → DirectDeclarator
    DirectDeclarator "[" TypeQualifierList? "*" "]"
                                              → DirectDeclarator
    DirectDeclarator "(" ParameterTypeList ")" → DirectDeclarator
    DirectDeclarator "(" IdentifierList? ")"   → DirectDeclarator
    Pointer+                                  → PointerSeq
    "*" TypeQualifierList?                    → Pointer
```

Contrary to the (E)BNF, production rules are rather oriented as "reduction rules". This excerpt of a C grammar module focuses on function "declarators", i.e., signatures that are used both to declare and to define functions. The interface of the module specifies that it provides the symbols `Declarator`,..., `PointerSeq`, and requires from other modules the symbols `ConstantExpressions` etc.

Figure 4.2: SDF excerpt of the C grammar

```
#include <stdio.h>
int main (void)
{
  printf ("Hello, world!\n");
  return 0;
}
```

Figure 4.3: Hello world, the famous "first" C program

Stratego/XT uses the state-of-the-art SGLR [Visser, 1997c] parser. It provides all the required features, modularity and ambiguity support, and produces parse forests efficiently encoded thanks to the ATerm library [van den Brand et al., 2000], sporting small memory footprint and maximal sharing of common subtrees. Actually, it is an "ambiguous parse tree" that is built, with `amb` nodes grouping alternative (sub-) parses in a more useful way than genuine parse forests.

As another consequence from having chosen to use the ISO C99 standard grammar verbatim, we inherit its syntactic ambiguities, some of them being "gratuitous", others requiring more powerful context sensitive disambiguation techniques. The most typical context sensitivity of C is its dependency on identifier types. For instance depending on the kind of entity the identifier `a` was associated with, `(a) * (b)` might cast `*b` to type `a`, or, multiply `a` by `b` — symbols denoting unary and binary operators exhibit the same ambiguity, e.g., `-`, `+` and `&`.

Consider Figure 4.3 as a running example. This program (text) is the input provided to SGLR, which will result in a parse forest. Besides the numerous ambiguities within the `stdio.h` file, there is one in the `main` part: `printf`, according to the ISO C99 standard grammar, can be either the name of an enumeration constant, or an identifier (i.e., for a variable, type, or function). Figure 4.4 precisely shows the two production rules in competition.

### 4.3.3 Disambiguation

In the tradition of Yacc, context-sensitive ambiguities are addressed by an elaborate cooperation between the parser and the scanner, maintaining a common table of symbols — was the identifier `a` declared to denote a type, or a variable? This results in a deterministic parsing: at most one parse tree is found. On the contrary, in the SGLR approach the ambiguous AST is traversed to gather

context-sensitive information used to prune invalid parses (an approach called "semantics driven disambiguation" by van den Brand et al. [2003]).

Because the C-Transformers projects aims at modularity and extensibility, we wanted to (i) embed the disambiguation filters in the SDF grammar (thus enjoying modularity for free), (ii) be declarative, and (iii) relieve as much as possible the programmer from specifying the order and types of tree traversals. AGs fit very well these constraints.

**Attribute grammars** [Knuth, 1968] is a formalism that supports syntax directed semantic analysis: each (grammar) rule is decorated with a set of equations that relate a node's **attributes** with those of its neighbors. AGs allow to focus on local aspects, leaving the global evaluation order aside, under the responsibility of a generic engine. Although AGs cannot modify the trees, their use for disambiguation is straightforward. Attributes convey information, e.g., a symbol table. Conflicting branches of the parse forest are flagged, and a (language generic) filter is run afterward on the parse forest, pruning inconsistent alternatives.

Since no AG engine existed for SDF, we developed one. Attribute rules are embedded in the SDF grammar as additional annotations.

Figure 4.4 demonstrates the use of AG to disambiguate C. The performances of the system are very satisfying (see section 4.5): disambiguation is negligible compared to the whole parsing (including conversion into AST).

In our Figure 4.3 example, each ambiguity branch of the ambiguity will be evaluated. The interpretation of `printf` is one such ambiguity according to the rules of Figure 4.4. During the traversal of the `stdio.h` part, `printf` will be recorded in `lr_table_in` as declaring a `Function`; during the traversal of the `main` part, the disambiguation rules of Figure 4.4 will therefore flag as valid the derivation `Identifier` → `PrimaryExpression`, and *invalid* the one with `Identifier` → `EnumerationConstant` because the lookup for `printf` in`lr_table_in` will not match the `Enum` kind.

A small auxiliary tool will prune all the invalid options afterward, yielding a unique parse tree. This parse tree (which includes all the details about the layout, comments, exact characters that were used etc.) is then simplified into a much more compact AST (freed from the layout, lexical details, etc.).

### 4.3.4 Transformations

The C-Transformers project, and its peer C++-Transformers, is somewhat ill-named, since it does not directly address transformation; rather it is a *workshop*

```
Identifier → PrimaryExpression
       {attributes(disamb:
          root.ok := <lookup> (Identifier.string, root.lr_table_in)
                     ;  (Variable <+ Function)
       )}

Identifier → EnumerationConstant
       {attributes(disamb:
          root.ok := <lookup> (Identifier.string, root.lr_table_in)
                     ;  Enum
       )}

Identifier → TypedefName
       {attributes(disamb:
          root.ok := <lookup> (Identifier.string, root.lr_table_in)
                     ;  Typedef
       )}
```

This example focuses on an ambiguity of C: an identifier `foo` might denote a variable, a function, a value of an enumeration type, or a type name. When traversing a node of the first kind, make sure that the `Identifier` was declared to be a variable or a function. If not, mark the node `root`'s attribute `ok` to be failed. This will prune this (incorrect) alternative from the associated ambiguity node. The other cases are similar. Note that the table `lr_table_in` is automatically propagated from Left to Right.

Figure 4.4: AG-driven disambiguation

*for implementing C program transformations*. Any transformation system is suitable, provided it supports our format for ASTs.

Amongst the possible engines to express transformations, we particularly like the Stratego programming language stratego-language.org. Because subsection 4.4.3 demonstrates a transformation written in Stratego, it is worth being described here.

In Stratego every piece of data is a **term**, i.e., a (abstract syntax) tree. **Conditional rewriting rules** specify how a particular tree matching a specific shape should be transformed. A specific transformation, such as translating extended C to C, involves several rewriting rules to apply at different places of the tree, and in a specific order. **Rewriting strategies** provide an elegant means to control when and where to apply rewriting rules. In addition, a rich set of operators allows to build arbitrarily complex transformations (i.e., strategies) from simple atomic ones.

Many transformations are sensitive to the static scoping rules of the target language. For instance $\alpha$-conversion, the renaming of variables, must assign different names to the same identifier occurring in different scopes. **Dynamic (rewriting) rules** handle scopes gracefully: they can be created at any time but have their existence bound by scopes. To perform $\alpha$-conversion, traverse from left to right, and for each variable declaration create a rewriting rule that maps the identifier to a fresh one. Entries and exits of scopes trigger the creation and destruction of the associated dynamic rules. This is much simpler than having to write generic (static) rules dealing with tables of symbols.

Finally, thanks to a tight integration with SDF, Stratego features **concrete syntax**: although rewriting rules do transform *abstract* syntax trees, rules can be written in the target's language *concrete* syntax. Examples of Stratego are given in subsection 4.4.3.

In the Figure 4.3 example, a transformation could be performed on the AST, e.g., replacing the call to `printf` by a call to the faster `fputs` function.

### 4.3.5 Pretty-Printing

Pretty-printing, i.e., conversion from an abstract syntax tree to concrete syntax (text), is performed by Generic Pretty Printer (GPP) [de Jonge, 2000], driven by language specific tables. These tables are generated from the SDF grammar, with embedded handcrafted directives to improve the result.

In the Figure 4.3 example, the final pretty-printed result of our chain would be *very* different from its input since instead of `#include <stdio.h>` one would have the whole content of the file. This issue is discussed in section 4.6.

## 4.4 Case Study

C-Transformers provides a simple and powerful framework to transform C programs. To demonstrate its capabilities, we extend C into ContractC: C with "design by contract" support. The C-Transformers framework will be used to compile ContractC down to ISO C.

### 4.4.1 Design by Contract

Design by Contract is a software design and implementation methodology invented and promoted by Bertrand Meyer for his language, Eiffel [Meyer, 1997]. The starting point is to consider that a function call involves two parties, the supplier and the client. The signature of the function is a (weak) form of a contract:

- the types of the incoming argument(s) are requirements put on the client (the caller) by the supplier (the callee);
- the type of the outgoing result(s) are guarantees given to the client by the supplier.

A successful function call requires that both parties respect their part of the contract. Statically typed programming languages enable statical checks, i.e., at compile time, while dynamically typed languages delay the verification until execution time. Note that in addition the signature of the function is a (weak) form of documentation. For instance, the signature of the square-root function, `double -> double`, specifies that it requires and returns a floating point number. Such information is always provided either in the documentation, or in comments, in dynamically typed languages.

Design by Contract extends signatures to include predicates between incoming and outgoing arguments. For instance the square-root function requires a non negative argument (a **precondition**) and ensures that the square of its result equals the incoming argument (a **postcondition**). Support for and use of pre-/postconditions dramatically improve the safety of programs, in particular when reusing components [ISE Software, 1993] — as an extreme example, Eiffel promoters claim that design by contract could have avoided the failure of the Ariane 501 launcher [Jézéquel and Meyer, 1997].

To demonstrate the use of C-Transformers, in the following we add support for pre- and postconditions to the ISO standard of C, based on the proposal of [Crowl and Ottosen, 2005] for a C++ standard extension. The resulting language is here named ContractC.

```
double sqrt (double r)
    precondition
    {
        r >= 0.;
    }
    postcondition (result)
    {
        result >= 0.;
        equal_within_precision (result * result, r);
    };
```

Figure 4.5: An function declaration example in ContractC

### 4.4.2 Syntax

The adaption of the C++ extension proposal [Crowl and Ottosen, 2005] to C results in adding support for pre- and postconditions to function *declarations*, not implementation. Indeed, since pre- and postconditions are pieces of formal documentation, they belong to the header file, which corresponds to the interface of a module in C parlance. Nevertheless, when compiled, the contract is to be integrated in the implementation of the function: it is the callee which will ensure that pre- and postconditions are properly met.

See Figure 4.5 for an example of ContractC: a set of pre- and postconditions put on the function `sqrt`.

To implement ContractC in Transformers, the first step is to extend its grammar with an additional rule for function declarations: Figure 4.6.

### 4.4.3 Compilation to C

The ContractC compiler (towards C) translates the contract into code run by the supplier — the function called. As an example, the ContractC declaration of Figure 4.5 transforms the regular C implementation of Figure 4.7 into that Figure 4.8. Writing such a transformation in Stratego is simple: see Figure 4.9, Figure 4.10, and Figure 4.11.

## 4.5 Discussion

To be effective, the C-Transformers tool chain must be easily extensible, i.e., every component must be easily configurable. In other words, featuring modularity is

```
module PrePostConditions

imports Declarators

exports
  sorts
    DirectDeclarator ReturnValueDeclaration
    Assertion PostCondition PreCondition

  context-free syntax
    DirectDeclarator "(" ParameterTypeList ")"
      PreCondition? PostCondition?  → DirectDeclarator
```

This rule is based on [Crowl and Ottosen, 2005, Section 2.3].

Figure 4.6: Extension of the C grammar to support pre- and postconditions

```
double sqrt (double r)
{
  return _libc_sqrt (r);
}
```

Figure 4.7: A C function implementation

```
double sqrt (double r)
{
  double result;
  {
    assert(r >= 0.);
  }
  {
    result = _libc_sqrt (r));
    goto end;
  }
end:
  {
    assert (result >= 0.);
    assert equal_within_precision (result * result, r);
  }
  return result;
}
```

This is the final result, when the contract specified in the function declaration (Figure 4.5) is inserted in the body of the function implementation (Figure 4.7).

Figure 4.8: A C function implementation with contracts installed

```
prepost = io-wrap(alltd(FunDecl <+ Contract))
```

In a single top-down traversal (`alltd`), for each function declaration with contract create a rule to install the contract, or for each function definition, install the contract.

Figure 4.9: The top-level of the transformation

```
FunDecl:
  Decl|[ ret fn (args) pre post ; ]| ->
  Decl|[ ret fn (args) ; ]|
    where
      rules (Contract :
        FunDef|[ ret' fn (args') cpstm ]| ->
        FunDef|[ ret' fn (args') cpstm' ]|
          where <transform(|ret', pre, post)> cpstm => cpstm')
```

Each function declaration with a contract must be rewritten without, and create an additional instance of `InstallContract` dedicated to the current function name `fn`.

Figure 4.10: Handling a ContractC function declaration

not merely satisfying with regards to current programming mottos, it is a must-have for every single tool involved in the chain.

In order to support **modularity**, the *parsing technique* virtually needs to support the full class of context-free grammars — for usual proper subclasses such as LL and LR are known not to be stable under union. The scanner-less generalized LR parser, SGLR supports the full context-free class of grammars — and actually some more. In addition SDF provides powerful and convenient means to compose modules. The built-in support for (possibly ambiguous) AST construction also enable to focus on actual issues, instead of having to code lengthy AST support classes or to fight parser conflicts.

The *disambiguation* also requires modularity in its strongest sense: a simple disjunctive "union" of disambiguation tools won't suffice. For instance, the sample ambiguities cited in the introduction, `(a) * (b)`, obviously affect the pre- and postconditions. Not only do we need to be able to add new disambiguation rules to those of the host language, but we also need to intermix them — just as freely as for SGLR modules. Attribute grammars handle this gracefully, being modular by nature, and better yet, sharing the exact same definition of modularity as the grammar itself.

The *transformation* needs to focus on specific spots; it is not concerned by most of the nodes. Stratego provides generic traversal operators that not only relieve the programmer from tedious work, but also guarantee the independence of the transformation from changes in the host grammar. In other words, Stratego also meets the modularity requirements.

Finally the *pretty printing engine* needs to support plug-ins to express the

```
InstallContract(|as1, as2, type, res):
  CmpdStm|[ body ]| ->
  CmpdStm|[
     {
       type res;
       {
         as1
       }
       body'
     end:
       {
         as2
       }
       return res;
     }
  ]|
    where <alltd(ReturnToGoto(|res))> body => body'
```

Installing the contract takes more code to take several details into account: pass conditions to `assert`, gather the return type to declare the `result` variable, handle possible name clashes with its name, replace `return` with assignment-and-goto etc. Ultimately, once converted the pre-/postconditions assertions `as1`/`as2`, the `return` in the `body`, and the name of the result variable `res`, the function declaration is transformed.

Figure 4.11: Installing the contract in the function implementation

| Type | Length | Comment |
|---|---|---|
| **Grammar** | 6 rules | Pre-/postconditions, function declaration |
| **Disambiguation** | 6 lines | Same as for regular function declaration |
| **Pretty Printing** | 4 rules | Pre-/postconditions, function declaration |
| **Meta-variables** | 3 rules | Pre-/postconditions, assertions |
| **Transformation** | 60 lines | 20 of which obey C formatting rules |

Figure 4.12: ContractC implementation effort

visual structure of the additional constructs. While GPP does support such add-ons, we find embedded pretty-printing rules more convenient. Being bound to the grammar, they share its modularity in the exact same sense, like attribute rules do.

If any of these components were to lack modularity support, or even slightly deviates from the standard set by SDF and SGLR, then the transformation might become undoable, or would require massive infrastructure. The result would also become extremely fragile: any change in the host grammar or in the extension can possibly require a full overhaul. C-Transformers features the full concept of modularity, allowing concise and robust implementations of transformations.

Although still young, C-Transformer is already very **usable**: Thanks to modularity the implementation of ContractC is quite straightforward and very compact. The volume of the full project is given in Figure 4.12.

The whole processing chain is composed of several steps, each one adding its own overhead to the actual transformations.

**Preprocessing** We run a home-grown preprocessor, equivalent to POSIX `cpp`, but producing slices of the input.

**Parsing** The execution of SGLR on each slice.

**Concatenation** All the different slices are pasted together.

**Evaluation** The attributes are computed on the parse tree.

**Pruning** Removal of the alternatives of ambiguities flagged as incorrect by the evaluation.

**Checking** Making sure no ambiguity remains.

| | Queens | | Hello, World | | Lemon | |
|---|---|---|---|---|---|---|
| **Lines of code** | 76 | | 448 | | 3550 | |
| **Duration** | s | % | s | % | s | % |
| **Preprocessing** | 0.11 | 15 | 0.13 | 1 | 0.18 | 0 |
| **Parsing** | 0.09 | 12 | 2.41 | 25 | 8.11 | 4 |
| **Concatenation** | 0.01 | 1 | 1.3 | 13 | 2.73 | 1 |
| **Evaluation** | 0.12 | 16 | 1.14 | 12 | 14.16 | 6 |
| **Pruning** | 0.08 | 11 | 0.8 | 8 | 8.15 | 4 |
| **Checking** | 0.01 | 1 | 0.2 | 2 | 1.53 | 1 |
| **Implosion** | 0.13 | 18 | 3.15 | 33 | 179.5 | 81 |
| **Transformation** | 0 | 0 | 0.02 | 0 | 0.35 | 0 |
| **Pretty-printing** | 0.18 | 25 | 0.53 | 5 | 6.95 | 3 |
| **Total** | 0.73 | 100 | 9.68 | 100 | 211.66 | 100 |

"Queens" is extremely short and includes no header. "Hello, World" is presented in Figure 4.3. "Lemon" is a parser generator that fits in a single C file.

Figure 4.13: Running time of the C-Transformers chain

**Implosion** Conversion to a suitable form for transformations (construction of an AST from the parse tree).

**Transformation** Compilation of ContractC to C.

**Pretty-printing** The AST is converted back to a concrete syntax program.

Three examples were chosen to measure the contributions of each step, see Figure 4.13.

The great variation of the figures is due to the fact that these samples are quite small, the last one being the only significant example. Then the whole processing is dominated by the conversion of the parse tree in an AST. We conclude that the cost of our technology AG-driven disambiguation is negligible, and there is not even reasons to try to optimize it further. Unfortunately we also conclude that, currently, parsing ambiguously grammars as ambiguous as those of C and C++ and then disambiguating is somewhat prohibitive.

We have been told that future versions of SGLR might perform the implosion during the parsing. Maybe some of the cost will be lowered, but unless SGLR is also able to run AG-driven disambiguation, it will still have to build massive (ambiguous) ASTs that will be pruned afterward.

## 4.6 Conclusion

We have presented modern program transformation techniques using C-Transformers as an example. To demonstrate the intrinsic power of these techniques, we implemented an extension to ISO C in an extremely reduced number of lines without sacrificing readability. We have emphasized how essential modular and language generic tools are.

In the future, several issues deserve more work.

The current AG system works perfectly well with very satisfying performance, but many improvements are expected. The most notable addition will be... additional syntactic sugar to cover the most common idioms we find during the implementation of semantics driven disambiguation filters: currently the attribute rules are lengthy and repetitive. Up to date AG engines, e.g. Utrecht University Attribute Grammar system (UUAG) [Baars et al., 1999], abstract attributes from production rules, which allows shorter and more readable declarations. Because our AGs are written in SDF and transformed with Stratego/XT, the methodology described in this paper can be applied to them.

The most severe issue with C-Transformers is that it "disrupts the programming style" [Waddington and Yao, 2005]: it processes and produces *preprocessed* C, i.e., with all the `#include` and other `#define` expanded. For instance, the simple "hello world" 4 liner program (see Figure 4.3) actually contains 450 lines of code coming from `#include <stdio.h>`. Not only does this clutter the result, it also makes it non portable. Indeed, much of the portability of C is handled by system headers that make wide use of operating system and/or architecture-specific code. This issue can be a show stopper to some possible applications of the C-Transformers project, for instance when users wish to ship the product but not the producer, or when the result is meant to be maintained by humans — as opposed to pre-processing phases. This limitation is a deliberate *temporary* choice: our limited resources were assigned to address the disambiguation first, and *then* programming style will be preservable. To cope with this issue we planned to develop a reversible preprocessor that embeds annotations in the AST to allow their reversal, very much in the spirit of Proteus project in fact [Waddington and Yao, 2005].

Once C completely tamed, we will focus (again) on C++. C++ inherits ambiguities from C such as `(a) * (b)`, but it also adds ambiguities of its own, even when identifier kinds are known. For instance, let `T` be a type, depending on the context `T(a);` denotes either the declaration of the variable `a`, or a call to the constructor `T::T(a)`. The `template` mechanism makes the process complex ad nauseam, requiring not only to carry symbols in tables, but also ar-

bitrarily long ASTs! Our C++ grammar is complete and the disambiguation filter is almost completed. In a foreseeable future, also making use of the reversible preprocessor, developing useful transformations with C++-Transformers should be possible.

## 4.7 Acknowledgments

The Transformers was started by Robert Anisko while an LRDE student; today it is developed by other LRDE students under the supervision of Akim Demaille. Valentin David was the architect of the current disambiguation chain (see subsection 4.3.3), now maintained by Alexandre Borghi. Other LRDE students have contributed significant portions of the Transformers project: Clément Vasseur, Nicolas Pouillard, and Olivier Gournet. The authors thank particularly Olivier Gournet who, behind the scene, made ContractC work.

# Chapter 5

# Providing mouldable failure management

Handling failures is often the choice of the developer of a library. It can throw an exception. It can return a special value. When calling a function, the proper code handling return values or exceptions has to be written and cannot be much optimize. We wanted to experiment an extension of language where the user would have more possibilities for choosing how to handle failures.

To experiment, we decided to extend the syntax if C. It does not provide exceptions, but error handling in traditional C is complex enough to make the experiment interesting.

Alerts are a bit similar to exceptions, except that they are handled statically instead of using a run-time mechanism. The transformation consists then just on converting extended C source code into a pure C source. Interestingly, the transformation is not syntactic de-sugaring like in chapter 4, the caller site is modified rather than the callee since that the user decides about the error management.

This paper was presented for GPCE'06.

# Stayin' Alert
# Moulding Failure and Exceptions to Your Needs

Anya Helene Bagge Valentin David Magne Haveraaen
Karl Trygve Kalleberg

*In the proceedings of Fifth International Conference on Generative Programming and Component Engineering (GPCE'06), October 22-26, 2006, Portland, Oregon, USA.*

**Abstract.** Dealing with failure and exceptional situations is an important but tricky part of programming, especially when reusing existing components. Traditionally, it has been up to the designer of a library to decide whether to use a language's exception mechanism, return values, or other ways to indicate exceptional circumstances. The library user has been bound by this choice, even though it may be inconvenient for a particular use. Furthermore, normal program code is often cluttered with code dealing with exceptional circumstances.

This paper introduces an alert concept which gives a uniform interface to all failure mechanisms. It separates the handling of an exceptional situation from reporting it, and allows for retro-fitting this for existing libraries. For instance, we may easily declare the error codes of the POSIX C library for file handling, and then use the library functions as if C had been extended with an exception mechanism for these functions – a *moulding* of failure handling to the user's needs, independently of the library designer's choices.

## 5.1 Introduction

Wherever there is software, there are errors and exceptional situations, and these must always be considered when writing and maintaining programs. Pro-

**© 2006 Association for Computing Machinery, Inc. Reprinted by permission from the proceedings of the 5th international Conference on Generative Programming and Component Engineering, Stayin' Alert: Moulding Failure and Exceptions to Your Needs, Anya Helene Bagge, Valentin David, Magne Haveraaen, and Karl Trygve Kalleberg**

gramming failure handling code is a tedious and error-prone task. Dealing with every possible exceptional situation leads to cluttered and hard to read code; not dealing with errors can have costly or perhaps even fatal consequences.

Some have argued that error handling should be avoided altogether. Instead, programs should be written so that errors never occur. Algorithms should be formulated so as to remove the exceptional corner cases, as this improves both the readability and maintainability of the code. This view is fundamental to the design of SPARK Ada [Barnes, 1997], where the Ada exception mechanism has been removed in an attempt at making validation and verification easier. This ideal advocated by such a "keep errors out" approach is certainly desirable. It is generally preferable to write algorithms with as few corner cases as possible.

In many cases, however, removing the errors altogether is simply not feasible [Romanovsky and Sandén, 2001]. Most modern applications run in multi-user, multi-process environments where they share resources such as storage and network with other applications. In these situations, operations on files, network connections and similar operating system resources can always fail, due to interaction with other programs on the running system or external devices.

Errors and exceptional situations need not always be caused by external factors, however. Even in situations where resource requirements are known in advance and guaranteed to be available, exceptional situations may occur, as none of the mainstream languages support resource-aware type systems [Truong, 2005].

As an example, consider the implementation of a simple abstract data type, say, a hash table, that is intended for other developers to reuse. In the case where the user (the *caller*) tries to look up a value for a non-existent key, an exceptional situation has occurred. Some possibilities for dealing with such a situation are:

- *Undefinedness*: this situation is outside the specified behaviour of the hash table. The caller cannot have any expectations as to what will happen.
- *Termination*: the program will terminate when this situation occurs. It is up to the caller to ensure that this does not happen.
- *Alert the caller*: report that an exceptional condition occurred. Given proper language mechanisms, alerts allow the user of the hash table to implement alert handling, such as logging, recovering from or ignoring the failure.

Undefinedness requires no language support, and termination can usually be implemented by a call to an `exit` function. In languages supporting Design

by Contract (discussed in Section 5.2.1), termination is automatic if a function fails to satisfy declared conditions either before or after invocation.

Several different alert reporting mechanisms are in common use. Goodenough [Goodenough, 1975] first introduced the exception handling mechanism[1] that is now found in most modern languages, and is currently the recommended way of handling exceptional situations. Returning a special error value, often `-1` or `null`, or setting a global variable is another common technique, often used in older code and languages. Other than exceptions, most reporting mechanisms are ad-hoc, in that there is no way to declare which mechanism is used. Conventions do exist – for example, most POSIX [POSIX.1] functions report errors by returning `-1` – but they are not declared explicitly in the code, making it difficult to automate alert handling. We therefore propose that each function declares its alert reporting. For example, a hash table lookup function may declare that it returns `null` if the key was not found:

```
val lookup(tbl t, key k)
  alert NotFound post(value == null);
```

Handling alerts is no easy task either. Different reporting mechanisms have different default handlers – exceptions, for example, typically terminate the program if they are not caught, whereas return values are ignored if they are not explicitly tested. Furthermore, different alert reports are checked in radically different ways; exceptions are received by a **`try/catch`** clause somewhere in the call hierarchy, return values must be checked after each return – often tedious and inconvenient. Changing the report mechanism means changing all handlers. We propose a way of declaring handlers which is independent of the alert reporting mechanism, and which can apply at various granularities, from a single expression to the whole program. For example, the following handler ensures that `NotFound` errors from the `lookup` function is handled by substituting the string `"(unknown)"`:

```
on NotFound in lookup() use "(unknown)";
```

Our contribution in this paper is a detailed discussion of failure handling mechanisms and the proposal of a language construct for alerts: Alert reporting may be declared for precondition and postcondition violations, exceptions, error

[1] The word 'exception' was coined as a way of emphasising that exceptions are not just for handling errors, but can be used for any kind of exceptional circumstance. However, it is easy to confuse the concept of handling exceptional circumstances and the exception handling language constructs found in many languages. We have therefore elected to use the word 'alert' for any reported exceptional situation, independent of the alert reporting mechanism and the alert handler, which receives the alert report and deals with it appropriately. The word 'exception' on its own will refer to the language construct.

flags and return codes, simplifying the task of staying alert for run-time problems. Alert handlers can be defined independently of the reporting mechanism, allowing a library implementor to alert its user in a way convenient for the library, and a library user to handle the alert in a convenient way at the call site. Alert handlers can be declared at a per-function and per-call-site basis, but it is also possible to declare policies common to a group of functions, such as a class or a library. In this way we can relatively easily retro-fit alert declarations for legacy code, e.g., the POSIX C library, easing the burden of checking in all kinds of strange ways for relevant I/O errors. Hence we approach *mouldable programming*, a way of moulding programming to our needs, and not being forced to program in strange ways due to arbitrary choices from language and library designers, or from perceived expectations from a user community.

This paper is organised as follows. In Section 5.2, we elaborate on the problem of handling failures and exceptional situations. In Section 5.3, we discuss separation of concerns, and granularity. In Section 5.4 we introduce our alert language extension, and continue by discussing its implementation in Section 5.5. In Section 5.6 we discuss related work, leading up to a concluding discussion of our language extension in Section 5.7.

## 5.2 Problem

The problem we are facing, is implementing an *alert protocol* between callers and callees that can transmit status information from the callee about the validity of its computed result back to the caller. Goodenough [Goodenough, 1975] points out that this is a way to extend an operation's domain (input space) or range (output space). The caller will declare an *alert handler* for the types of alerts it wants to handle, and the callee may *report an alert* during its computation, thus becoming the *(alert) reporter*.

### 5.2.1 Alert Reporting Mechanisms

Current programming paradigms and languages provide a number of ways for dealing with failure, dating back to the earliest days of programming. Hill [Hill, 1970] discusses possible mechanisms, anno 1969, which includes specific return values, use of gotos to parameterised labels, callbacks, global error flags, and passing pointers to variables which will receive an error status.

**Return Values** Designating at least one value in the domain of the return type as an error marker is perhaps the most prevalent form of alerting. This technique

is frequently found in operating system APIs, such as POSIX and Win32, in many language standard libraries, and in many frameworks. Functions returning objects often use `null` as such a marker, functions returning numeric values for file handles or indexes often use `-1`. If the return type only allows for one error marker, an additional mechanism, such as a global flag, is needed to distinguish between different kinds of errors.

The IEEE floating-point arithmetic standard [Goldberg, 1991] allows a wide range of error return values. Some of these automatically propagate through an expression, like NaN - "Not a Number". NaN occurs as a result of $0/0$, $\sqrt{-1}$, $\log(-1)$, etc. A more interesting error value is $+\infty$ or $-\infty$, which is the result of e.g. $M/m$ where a very large number $M$ is divided by a very small number $m$, resulting in numerical overflow - a number too large to be represented as a floating point number. Infinities propagate through addition, subtraction and multiplication, but disappear after division. The expression $a + 4/(M/m)$ yields $a$, as 4 divided by infinity yields 0.

If the return value for failure can also be a valid return value, for example if division by zero returns zero, we are faced with the so-called semipredicate problem: it is not possible to know if the return value signifies a failure or a valid value.

A property of the return value mechanism is that it will only propagate the alert one level, to its immediate caller. Also, it requires no alert handler setup or teardown, and thus has no overhead.

**Global Error Flags** Many older APIs, such as POSIX and Win32 use global error flags, often in conjunction with special return values, to elaborate on a failed function. In Win32, the function `GetLastError` is used to retrieve the failure code of the previously executed system call. In POSIX, the global `errno` variableserves an identical purpose.

The use of a global error flag variable is not thread safe. Unless special consideration is taken, multiple threads in the same process will share the same error flag variable, making it impossible to know which of the previous threads' system calls a given error belongs to. This is alleviated by having global error function, like `GetLastError`, instead.

**Long Distance Jumps** In C, the functions `setjmp` and `longjmp` are used to transfer control directly from one stack frame to one which is arbitrarily higher up. This report mechanism is often used to propagate errors many levels, but can only send an integer value. This is a low-level C/Unix-specific technique, which is also found as `RtlUnwind` in Win32. Both alternatives rely on low-level

machine-specific register set saving and compiler knowledge. Another drawback is the difficulty of freeing allocated resources properly before the handlers for such resources leave the variable scope.

**Exceptions** Today, the most common way of alerting is to use the exception mechanism introduced in [Goodenough, 1975], in languages that provide this, such as CLU [Liskov et al., 1977], C++ [Stroustrup, 1997], Java [Gosling et al., 1996], Ada [Taft et al., 2001], ML [Milner et al., 1997] and Python [van Rossum and Drake, 2003].

Raising an exception consists of two parts: First, a function, say A, sets up an exception handler listening for a particular type of exception E, using a **`try/catch`** (Java), **`handle`** (SML) or **`try/except`** (Python) construct. It then invokes the function B, either directly or indirectly. B raises the exception E by invoking the **`raise`** (Python, SML) or **`throw`** (Java) language construct, and the search for an appropriate exception handler starts. Each stack frame is consulted in succession, until one with a handler for E is found. If no new handler for E is declared in the functions between B and A on the stack, control is transferred from B to the handler declared in A.

For the languages above, after the handler in A finishes, execution continue in A. In other languages, it is possible to either resume after the **`raise`** statement in B, or restart B, see Section 5.6 for details.

Exceptions may be either *checked* – must be declared by all functions that may throw them, both directly and indirectly – or *unchecked* – may be thrown without being mentioned in a function's list of throwable exceptions. In Java, checked exceptions are the default, but unchecked exceptions are used for catastrophic errors, such as out-of-memory errors and disk failure.

Exception handlers need not only be declared as markers on the stack. They can also be attached to classes, statically giving each class its own handler, or to objects giving each object a specific handler. This is discussed in Section 5.6.

**Condition System** The PL/I **`ON`** condition system, allows the programmer to attach handler blocks for pre-defined language exceptions that may occur in expressions, such as division by zero and end of file. These handlers are installed and removed dynamically. Extensions of this idea can be found in Dylan, Smalltalk and Lisp, which have systems for detecting exceptional situations based on (a restricted description of) the state space of a program. When a given failure condition is met, control is transferred to a specified error handler which can elect to try error recovery followed by resume, or terminate the function that threw the exception.

**Event Handlers** Event handlers and POSIX signal handlers both provide a callback mechanism which may be used for passing error notifications from the operating system to the application, or between parts of an application. This model requires no special language support, and is usually tied to the API or framework the application was written with, e.g. POSIX (signals) or Win32 GDI (events).

**Guarding** A pre-condition may be declared on a function, testing beforehand whether the function will return normally with the data. Effectively, pre-conditions ensure that the input falls within a function's domain, and attempts to ascertain whether the state of the system allows the function to complete. Formulating such a pre-condition may not always be possible, e.g., during complex interaction with external resources.

**Contracts** A significant extension to guarding is *design by contract*, described by Meyer [Meyer, 1997]. In this technique, explicit pre- and post-conditions are declared on every function. Whenever either fails, the program terminates immediately. A contract should never be checked by the caller; contract verification must happen during the implementation phase, not at runtime. Eiffel [Meyer, 1992] was the first language to support and enforce contracts, but also comes with a notion of exception handling. A routine may have a rescue handler declared for it, which may either provide some default return value, retry the routine, or fail. In the latter case, the failure will be propagated to the method's caller.

**Goto** The use of goto as an exception handling technique has almost disappeared with the introduction of various exception handling language features. In some restricted domains, such as the kernel code of operating systems, where space and performance considerations outweigh readability, gotos are still prevalent.

### 5.2.2 Alert Handling Policies

Even using the same basic alert reporting, different usage policies lead to large differences in alert handling in the design of frameworks and libraries. The policy about retrying on failure, is one example. Unix leaves it to the user to retry failing operating system calls, for example if a long running kernel operation is preempted. Windows (and BSD Unix variants), on the other hand, retries preempted operations automatically.

In many languages, guiding principles exist about using the exception handling feature of the language. This is the case for Java, where the general recommendation is to use exceptions for alerting. Despite such principles, there are numerous examples in the library where error return values are used, among them in the implementation of the hash table.

### 5.2.3 A Game of Anticipation

The state of the art is to use design experience on a case-by-case basis to provide suitable alerts. Specifically, there exists no declarative way to select the desired error handling mechanism for part of a program. The need for design experience comes from the fact that fundamental tradeoffs between the caller and callee of an abstraction must be managed. The caller is the party which will be implementing the alert handling. As the various handling techniques have different affinities with alert reporting, and every caller is potentially different, the implementer of the callee must anticipate the handling techniques that will be used by the callers.

From the callee side, the ideal reporting mechanism may depend on the implementation of an algorithm. For instance, if we are within a deeply nested data traversal, it may be more convenient to throw an exception than to use return values.

Another consideration is who should do error checking on the input parameters. Should the callee accept erroneous input and produce garbage? Should the callee do all checking? This decision is usually coupled with significant performance tradeoffs.

The amount of anticipation required by the implementer of the callee is significant, perhaps especially in core language libraries. In Java, an example can be found in `java.util.Queue`, which provides pairs of identical functions, save for differences in alert reporting: `poll()` and `remove()` can both be used to remove the head of a queue. `poll()` returns `null` if there is no head, whereas `remove()` throws an exception in the same case. Similarly, `add()` throws an exception if a new element cannot be added to a queue, whereas `offer()` returns `false` if the insertion failed.

Determining a suitable reporting mechanism when implementing a callee is compounded by another problem: the caller is the final arbiter of what is normality and what is failure. Returning `null` from a hash table lookup may in one application be completely acceptable, and not constitute a special case in the algorithm using the hash table. In another application, it will be the sign of severe data corruption and violation of crucial invariants. In the first case, returning a *null* is neither an exceptional case, nor an error, and this situation

is therefore not a candidate for alert handling. In the second, it is critical that proper alerting be used.

## 5.3 Separation of Concerns

**Separation of Alert Reporting and Handling** Although the mechanisms in Section 5.2.1 are essentially equivalent in that they all report exceptional situations (possibly with additional information), the default action taken when an error occurs differs. For return values and error flags, the default is to ignore the error. For exceptions, the default is to propagate the exception through the call hierarchy, possible leading to termination of the program. For guarding, the default is not to guard, i.e., ignore the error.

In all cases, the callee implicitly decides the default action in case of an error, by choosing a given report mechanism. This is unfortunate, since the goal of raising an alert to the caller is to let the caller decide the appropriate course of action (otherwise, the callee could simply handle everything on its own). Additionally, the choice of alert mechanism is often based on implementation pragmatics, rather than whether the default action is likely to be appropriate for the severity of the error. If the callee is changed to use a different mechanism, all call sites must be updated. Thus, we have a *tangling* of alert mechanisms (callee implementation) and alert handling (caller implementation).

**Separation of Normality and Exceptionality** With existing techniques for handling exceptional situations, we get a tangling of a program's normal behaviour and its exceptional behaviour. If our handling policy is to report small errors to the user and abort the program on serious errors, we have to code this into all places where errors are handled. Thus we end up with a mix of code dealing with normal circumstances, and code dealing with exceptional circumstances (alert behaviour). This leads to cluttered code and maintainability problems: if we wish to change our policy for some errors, we may need to change a lot of code in many different parts of our program.

This problem has also been observed in [Lippert and Lopes, 2000], where aspects are used to untangle the alert behaviour from the normal behaviour. The authors advocate that alert handling code be put in separate declarations – aspects – instead of being scattered around the code (see Section 5.6).

**Granularity** In some cases, mixing normal and alert behaviour may be beneficial; for example, by taking alerts into account locally, we may compensate, e.g., by substituting a different return value. In some cases, such decisions must be

made for each call site; in other cases, we may be able to provide a common policy for an entire class, a module or a set of functions.

An example of this is IEEE floating-point expressions. Here a NaN is propagated through the expression and may be tested for, while an overflow ($\pm\infty$) may be propagated or consumed depending on the expression itself.

## 5.4 Alert Language Extension

While the existing body of research on exception handling addresses many of the concerns we have mentioned above, one area of the design space remains relatively unexplored: how to extract and declare separately the handling of exceptional situations. This is what we will address in the next sections.

Our mouldable abstraction of alert handling provides for separation of mechanism and handlers, separation of normal behaviour and failure behaviour, and allow decisions to be taken at the appropriate level of granularity, i.e. at the expression, statement, function, class, module or component level. Furthermore, our proposed solution allows the callee to declare what is normality and what is exceptional; allows the caller to declare the desired alert handling policies; can be applied retroactively to existing libraries; and is able to distinguish different types of errors.

A grammar for the alert extension is presented in Figures 5.3, 5.4 and 5.5. Although our prototype implementation (discussed in Section 5.5) is an extension of the C language, we will discuss the extension in terms of a C/C++/Java/C#-like language with an exception facility.

The grammars in this paper are meant for human consumption, and not for use directly in an implementation. Non-terminals written in upright font are meant as hooks into the base language. Non-terminals ending in *name*, *type* or *expr* are all names, types or expressions of the appropriate type. The notation " $^{*}_{,}$ " and " $^{+}_{,}$ " is used for comma-separated lists.

### 5.4.1 Distinguishing Different Alerts

Alerts are declared with the **`alert`** declaration, using syntax similar to the **`enum`** declaration (see grammar in Figure 5.1):

```
alert {MyAlert};
```

Multiple comma-separated alerts can be declared in the same declaration. For example, a selection of POSIX error codes is listed in Figure 5.2. These errors can occur during normal file operations, such as `open`, `read` and `write`.

declaration ::= **alert {** *alert-def**, **}** *super-alert** **;**
*alert-def* ::= *alert-name* [(parameter-list)] *super-alert**
*super-alert* ::= **:** *alert-name*

Figure 5.1: Grammar for the declaring new alerts.

| | |
|---|---|
| EBADF | Bad file descriptor |
| EINTR | Interrupted system call |
| EIO | Input/output error |
| ENOENT | No such file or directory |
| ENOMEM | Insufficient memory available |

Figure 5.2: A small selection of POSIX error codes, used, e.g., for `open`, `read` and `write`. The codes are set in the global `errno` variable, and should be checked whenever a function raises an error (typically by returning `-1`)

We can give each of them a unique alert name with the following declaration, with different (case-sensitive) names to avoid name clashes:[2]

```
alert {eBadF, eIntr, eIO, eNoEnt, eNoMem};
```

If we look at the selected error codes (and a POSIX reference), we see that they fall into roughly four categories: temporary conditions (EINTR); system problems outside the program's control (ENOMEM, EIO); problems that might be correctable with user help (ENOENT); and programming errors (EBADF). Thus, it is useful to be able to group them, so that we may, for example, automatically retry temporary failures, ask the user for a new file name on permission or missing file problems, and abort the program on system errors and programmer errors. To do this, we organise our alerts in a hierarchy, similar to an inheritance hierarchy in OO programming (which is also used for exceptions – in Java, for example):

```
alert {Retry, AskUser, FatalSys, FatalBug};
alert {eNoEnt : AskUser};
```

The colon separates a sub-alert from its super-alert in a declaration. Multiple alerts can be assigned a common super-alert:

```
alert {eIO, eNoMem} : FatalBug;
```

[2] Alert names are in a separate namespace, but error codes are commonly declared with macros in C, which ignores namespace boundaries.

declaration ::= fun-dcltr (**alert** *alert-rep* | throws-clause)* [fun-body]
declaration ::= **alertrepdef** *alert-rep alert-rep-name* **;**
declaration ::= **funspace** *funspace* (**alert** *alert-rep*)+
*alert-rep* ::= [*alert-name*] **pre** [**unless**] **(** cond-expr **)**
*alert-rep* ::= [*alert-name*] **post** [**unless**] **(** cond-expr **)**
*alert-rep* ::= [*alert-name*] **on throw** *exception-name*
*alert-rep* ::= *alert-rep-name*
*alert-rep* ::= **{** *alert-rep**, **}**

Figure 5.3: Grammar for specifying alert reporting. The *funspace* non-terminal is defined in Figure 5.5.

Additionally, an alert may have more than one super-alert:

```
alert {StupidMistake};
alert {eBadF} : FatalBug : StupidMistake;
```

To avoid cycles in the inheritance graph, alerts must be declared before they are used as super-alerts. The built-in alert `Alert` is the super-alert of all other alerts.

Finally, we note that it is sometimes useful for the callee to pass some information back to the caller. To do this, the alert must be declared with one or more arguments. For example, an `Error` alert which allows a message to be passed to the caller:

```
alert {Error(char *msg)};
```

We will see below how the values are passed from callee to caller.

### 5.4.2 Specifying the Alert Reports of a Function

Alert reports are specified at the callee side with an **alert** clause in the function declaration, c.f. grammar in Figure 5.3. Possible reporting mechanisms include condition checks before (`pre` conditions, useful for guarding) and after (`post` conditions, for checking return values and global error flags) a call, and exceptions. For example, to declare that a hash table lookup function `lookup` returns `0` on failure, we write:

```
val lookup(tbl t, key k) alert post(value == 0);
```

To use a more specific alert than the default Alert, we simply add the name of the desired alert:

```
val lookup(tbl t, key k) alert NotFound post(value == 0);
```

The **post**-clause takes an expression, which is evaluated after the function call returns – if the expression is true, the function has failed. The special keyword **value** can be used to check the call's return value (the type of **value** is that of the return type of the callee). Similarly, the **pre**-clause takes a condition which is checked *before* the function is called.

The condition expressions may be arbitrarily complex, but should only use globally accessible names, arguments to the call, and **value**. Arguments are referred to by the name they are given in the function declaration, and have whatever values they have at the time of checking (before or after the call).

```
// alert if table t cannot be expanded due to memory constraints
int insert(tbl t, key k, val v)
  alert eNoMem post (value == -1 && errno == ENOMEM);
```

Specifying a condition with the **unless** keyword negates it, providing a more intuitive way of specifying invariants and pre/post conditions (which are often specified in terms of what is normal, and not in terms of what is exceptional).

```
// separate success/failure flag if no return values can be used for alerting
val lookup(tbl t, key k, bool *success)
  alert NotFound post unless(*success);
```

Exceptions (if the language supports it) can be declared with a **throws** (Java) or **throw** (C++) clause, provided that the exception name has also been declared as an alert:

```
alert {AnException};
int f() throws AnException;
```

If a mapping between exceptions and alert names is desired, an **on throw** clause may be used:

```
int f() throws AnException
  alert Error on throw AnException;
```

In all cases, information may be passed to a handler using alert parameters:

```
val lookup(tbl t, key k)
  alert NotFound(k) post(value == null)
```

The callee must still provide some way sending this information, either through updating of arguments, return values, global variables or exception objects – alert parameters are merely a declaration of whatever mechanism is used. For exceptions, the exception object is available for use in alert parameters:

```
int f() throws AnException
  alert Error(e.msg) on throw AnException(e);
```

declaration ::= **handler** *handler-name* **(** parameter-list **)** statement
declaration ::= **on** *alert-pattern* statement
statement ::= **retry** [ **(** argument-list **)** ] [**max** int-expr] **;**
statement ::= **use** expr **;**
statement ::= *handler-name* **(** argument-list **) ;**
statement ::= statement-body **on** *alert-pattern* statement
*alert-pattern* ::= *single-alert*$^{+,}$ [**in** *funspace*]
*alert-pattern* ::= *alert-pattern* **or** *alert-pattern*
*single-alert* ::= *alert-name* [ **(** parameter-list **)** ]
*expr* ::= *expr* **<:** [*alert-pattern*] **:** *handler*
*handler* ::= expr | **{** statement **}**
*handler* ::= *handler-name* **(** argument-list **)**

Figure 5.4: Grammar for alert handlers. The *funspace* non-terminal is defined in Figure 5.5.

### 5.4.3 Alert Handling

The *alert handler* will treat all alerts the same, whether they are reported by return value, condition check, or exception. The grammar for alert handlers is presented in Figure 5.4. There are two alert handling constructs: **on**, for specifying an alert handler at any scoping level, down to a single statement, and the *handler operator*, **<::**, which specifies a handler for a single expression.

#### The on Construct

The **on** declaration takes a *alert-pattern* and a statement. The declaration is lexically scoped and applies to all call sites it matches within its block. The statement form of **on** applies to a single statement. If more than one handler matches, the most specific one closest in scope applies, or a compile-time error is given if there is more than one equally suitable handler.

The *alert pattern* specifies for which combination of alerts and callees the handler applies. The handler itself is a single or compound (block) statement, which should provide a replacement value, retry the computation, refer to another handler, or terminate the caller. It is an error for a handler to complete without providing a replacement return value when one is needed – in this case, we terminate the program (though we could check statically whether this can

occur, and other design choices are certainly possible).

Within a handler, the `use` statement may be used to provide a replacement value; `use` exits from the handler as if the callee had returned normally with the value provided. For example, the following defines a handler for `NotFound`s in the `lookup` function which substitutes the value `"Doe, Jane"` for failed lookups (e.g., when mapping ID numbers to names):

```
on NotFound in lookup() use "Doe, Jane";
```

The following does the same for all alerts in `lookup`:

```
on Alert in lookup() use "Doe, Jane";
```

The next two declarations both do the same for `NotFound` in all functions (the `%` matches all functions[3]):

```
on NotFound in % use "Doe, Jane";
on NotFound use "Doe, Jane";
```

The following `NotFound` handler applies to just the preceding `print` statement:

```
print(lookup(tbl, key)) on NotFound use "Doe, John";
```

The **`retry`** statement tries the failed call again (possibly with a maximum retry count, specified with "**`max`** *number*" – the default is to retry indefinitely). The **`retry`** statement also takes an optional list of arguments, which will replace the arguments in the failed call. It will exit from the handler, continuing execution at the point of the call which reported the alert – except when the maximum retry count has been reached, in which case execution continues with the next statement after **`retry`**.

For example, the following specifies that on a `NotFoundError`, we should ask the user for a new name and try again (maximum 5 times). If our recovery attempts fail, we substitute an empty string.

```
on NotFoundError in readfile(char *name) {
  warn("trying again...");
  char* name = askUser();
  if(name != NULL) retry(name) max 5;
  warn("giving up...");
  use ""; }
```

### The Handler Operator

The handler operator provides a convenient way of handling alerts at the expression level. The left operand is an expression to be evaluated, and the right operand is a handler to be used if an alert was reported in the expression. Within

[3]The `%` was chosen to avoid confusion with pointers (`*`) in C/C++, and is used in a similar fashion in AspectC++ [Spinczyk et al., 2002].

the operator itself, one may specify which alerts should be handled. Alerts are specified in the same way as for **on**, the handler can be either an expression giving a replacement value, a call to a previously defined handler, or a statement list (enclosed in braces). In the following example, a string is substituted if a lookup fails:

```
print("result:", lookup(t,k) <:: "Unknown");
```

An alert pattern can be specified between the colons:

```
print("result:", lookup(t,k) <:NotFound: "Unknown");
```

Furthermore, handler code can be provided, as for **on**:

```
fd = open(name, flags) <:eNoEnt: {
   char *newname = askUser();
   if(newname != NULL) retry(newname,flags);
   else giveUp("couldn't open file"); };
```

This will try to open a file, and if the file is not found, the user will be asked for another name. If the user provides one, we try that instead, otherwise, we abort with a message.

### 5.4.4 Abstraction

Our extension provides abstractions for alert handling and reporting. The **handler** construct declares handlers which may be used later on by the **on** declaration or the handler operator. For example,

```
handler log(msg, dflt) {
  print("An error occurred: ", msg);
  use dflt; }
```

which may be used as:

```
on NotFound in % log("Lookup failed", "");
name = askUser()
  <:eNoEnt: log("No response from user", "--");
```

Handler abstractions look deceptively like functions in both definition and use, but are not functions, since the **retry** and **use** statement would be tricky to implement in a separate function. Instead, the definitions are expanded inline wherever they are needed. Hence, (mutually) recursive handlers are not allowed.

The **alertrepdef** declaration declares alert reporting mechanisms for use in a function declaration. It follows the same pattern as the C/C++ **typedef** construct. This is useful when several functions share the same alert behaviour. For example,

```
alertrepdef alert Error post(value == 0) ErrorOnZero;
```

`ErrorOnZero` can then be used for functions raising errors with a zero return value:

```
int f() alert ErrorOnZero;
```

### 5.4.5 Sending Information from Callee to Caller

Alerts can have associated values (alert parameters), allowing a callee to provide additional information to a caller. A similar idea is found in exception handling (e.g., in Java or C++), where exceptions are objects that may contain information relevant to the exceptions. As shown in Section 5.4.1, valued alerts should be declared with arguments:

```
alert {Error(char *msg)};
```

At the callee side, we provide a suitable value in the **`alert`** clause:

```
int read(int fd, void *buf, size_t count)
  alert Error("read error") post(value == -1);
```

The value can be obtained at the handler side from the alert pattern:

```
on Error(msg) in read() { print(msg); exit(1); };
```

In this example, `msg` is declared as a string, and gets the value `"read error"` from the failed `read()`. If more than one alert is given in the alert pattern, all of them must have the exact same argument list. It is not necessary to mention the arguments if they are not needed by the handler.

If the return value of the callee is to be available to a handler, it must be passed as a parameter, as return values are not always available (e.g., for exceptions and `pre` conditions), and there is no way for the handler to distinguish between different alert reporting mechanisms.

Note that, unlike exceptions, we need not construct an alert object as an aggregate of values. Instead, code is generated in the handler which obtains the information directly (which is why only arguments, return values and global variables can be passed from the reporter). Thus, as for alert conditions, we are restricted to expressions which have the same meaning for both the callee and the caller (i.e., global names and operators, constants and arguments, either before or after the call).

In the case of functions which change their arguments (or modify global data structures), it is possible that the state of these variables is inconsistent when the handler is invoked. In this case, it is up to the handler to put things in a consistent state or terminate execution. Ideally, functions would ensure that the program state is rolled back to a safe point before an alert is reported, or at the very least, declare that this may not happen for some or all alerts. This problem

*funspace* ::= [return-type | **%**](function-name | **%**)
[ **(** parameter-pattern-list **)** ]
*funspace* ::= **funspace** *funspace-name*
*funspace* ::= **{** *funspace**, **}**
declaration ::= **funspacedef** *funspace funspace-name* **;**

Figure 5.5: Grammar for declaration of function spaces.

is also found with the common exception handling mechanisms. We have not dealt with this problem yet.

### 5.4.6 Granularity and Funspaces

By granularity, we refer to the coarseness of a declaration in the hierarchy from expression through statement, function declaration, and optionally class, module and subsystem level, all the way to the system level. Our language extension provides additional granularity alternatives, among them groups of functions, which we call funspaces. Funspaces can be applied to both alert handling and reporting.

#### Granularity of Handlers

In most languages, exception handlers are specified at the statement level, and the exception declaration (e.g., `throws` in Java) occurs along with the function declaration.

In our system, the declaration of both alert handlers and alert reporters can occur at various levels of granularity. As we have seen, handlers are declared with **on** declarations. These can occur at any level in the scope hierarchy (from global, through namespace/package and class, down to blocks and single statements within a function) and apply to the scope in which they are declared. Additionally, handlers can be declared at the expression level using the handler operator (**<::**). If multiple handlers are in conflict, the most specific handler takes precedence, i.e. the one with the most specific alert pattern at the finest level of granularity.

The concept of a scoping level can be refined using *funspaces*. A **funspace** declares a set of functions, i.e. a subspace of the namespace for function names. The grammar for funspaces and funspace declarations is presented in Figure 5.5.

A funspace is basically a list of function patterns. For example, (a non-exhaustive list of) the file operations of POSIX can be declared as follows:

```
funspacedef { open(), close(), creat() } posix_io;
```

Each entry in the funspace list conforms to a pattern. For C and languages without overloading, giving a function's name is sufficient. In languages with overloading, the full signature must be given;

```
funspacedef {
   int open(const char *pathname, int flags),
   int open(const char *pathname, int flags, mode_t m),
   int close(int fd),
   int creat(const char *pathname, mode_t m) }
  posix_io;
```

The pattern can also contain wildcards, with `%` matching any single item, and `..` matching any argument list. For example, The pattern `% %(const char*, mode_t)` would match any function with any return value, that takes two parameters: a `const char*` followed by a `mode_t`, e.g. `creat`:

```
int creat(const char *pathname, mode_t mode);
```

Funspaces, being sets of functions, can be merged, allowing us to construct the `posix` funspace from smaller, task-specific funspaces.

```
funspacedef {
   funspace posix_io,
   funspace posix_memory,
   funspace posix_process }
  posix;
```

This is not merely a syntactic convenience. Different subsets of a given API often use different sets of errors, each specific to that subset. Sometimes, the same numerical error value is reused with different meaning across different subsets. `ENOMEM` when returned from `mmap` has a different meaning than `ENOMEM` returned from `stat`. Using funspaces, these differences can be captured at the granularity of function groups, rather than having to be specified on per-function basis.

### Granularity of Alert Reporting

In the previous sections we saw how alert reporting is declared on individual functions. Using funspaces, reporting mechanisms may conveniently be declared on groups of functions. The following declares that the *eNoMem* alert will be reported on any function in the POSIX funspace if it returns -1 and the global variable `errno` is set to `ENOMEM`.

```
funspace posix alert eNoMem post(value == -1
```

```
        && errno == ENOMEM);
```

Both handlers and alerts can be declared at any scoping level and on funspaces, but the declarations are completely independent. For example, the alert *eNoMem* may be specified for all POSIX functions, as above, while a handler for this alert could be declared for only one expression inside a particular function in a given program, e.g.,

```
f() { open("foo",O_RDONLY) <:eNoMem:exit(EXIT_FAILURE);}
```

Or, it could be declared for all POSIX functions:

```
  on eNoMem in funspace posix { exit(EXIT_FAILURE); }
```

Multiple, overlapping funspaces may be declared, and both alerts and handlers may be specified independently for each funspace.

### 5.4.7 Interfacing with Legacy Code

Introducing new failure handling disciplines typically means that legacy code must be rewritten if it is to take advantage of it. This is the case with exceptions, for instance: if you want exceptions in an existing library which reports errors with return values, you will have to either rewrite the library or write a wrapper for it.

Funspaces, together with handling and reporting abstractions can be used to specify alert reporting mechanisms and handling for a large number of existing functions in a few lines of code. This makes reuse of existing libraries simpler, which is especially important since many older libraries use exotic alert reporting mechanisms which may be inconsistent with newer code.

If the library comes with structured documentation, it may be possible to automatically extract alert specifications from the documentation. For example, the POSIX standard [POSIX.1] comes with structured manual pages in HTML format, available online. A tool could be written to extract from the manual pages function names, return value style (“returns -1 on error”) and which error codes are applicable to the function, and generate the necessary declarations. We are currently exploring this option.

## 5.5 Implementation

We have made a prototype implementation [C-Alert] of the language extension for C that implements the compiler pipeline shown in Figure 5.7. The prototype is implemented using the Stratego/XT [Bravenboer et al., 2006b] program transformation language, and the C-Transformers [Borghi et al., 2006] framework for

C99 transformation.

The prototype consists of an extension to the C99 grammar [WG14, 1999] (written in the grammar formalism SDF2 [Visser, 1997a]) and a set of transformations translating the extended C code to standard C. As seen in Figure 5.7, the parser will recognise the extended C language and produce an abstract syntax tree (AST). Minimal type analysis is then performed to check that the handlers are type consistent with the functions they will be applied to. Next, the alert extension is "peeled off" in a desugaring step before the the AST is pretty-printed into text and fed to a normal C compiler.

### 5.5.1 Translation scheme

The translation algorithm works roughly as follows (for C99, following traversals can be combined into one pass, since the language requires declaration before use):

First, traverse the AST and look at all function declarations and definitions. For each declaration or definition, extract the signature (i.e., name, return type, argument types) and the alert mechanism (i.e. pre/post conditions – exceptions are not available in C), and store this for later.

Next, traverse the AST and look for scopes, function calls, `on` handler declarations or handler operators. When seeing a *scope*, create a new, nested scope for subsequent `on` handler declarations. When later existing this scope, drop all handlers registered in this scope. When seeing an *`on` handler declaration*, register its entire definition in the current scope. When seeing a *handler operator* (`<::`, expand the pattern in Figure 5.8. When seeing a *function call*, check if the signature of the callee is matched by any of the registered handlers. Check the textually closest handlers first and proceed to parent scopes. If a matching handler is found, expand the pattern in Figure 5.8.

Keep in mind that a function call is only rewritten if at least one relevant handler is found for it. Let us call the closest (and thus active) handler *current-handler*. Given a function call $f(e_1, ...)$ to function $t_0$ $f(t_1, ...)$ where $t_0$ is the return type, $f$ the name, $t_1$, ... the types of the formal arguments, and $e_0$, ... the expressions for the actual arguments, the instantiation of the template in Figure 5.8 occurs as follows: First, a local variable $r$ of type $t_0$ is declared, and will hold the eventual return value. Then, each of the actual arguments is evaluated and stored, each $e_x$ into a local variable $v_x$. Then, the expression for the `pre` condition of $f$ is evaluated on the variables $v_x$ (the expression <*precond(v0, ...)*> means that the pre condition code is expanded in-place – care is taken to avoid accidental variable capture), and if it succeeds, we must invoke the alert handler. The code for *current-handler* is expanded in place, and `use` statements in the

body are translated into assignments to $r$. The process is similar for the `post` condition and the `post` condition handler. An instantiation of this template was given in a cleaned up, human-readable form in Figure 5.6.

### 5.5.2 Implementation issues

In C, which lacks function overloading, matching a function call to the function's declaration can be done easily just by comparing names. In other languages, such as C++ and Java, overload analysis is needed to distinguish functions sharing a common name.

Although overload analysis is unnecessary for C, we still need to be able to determine the types of arbitrary expressions, in order to declare temporary variables and type-check substituted values. Support for this is lacking in Transformers, but was easily added.

A special problem occurs with function pointers, since it is in general impossible to determine statically which function will be called at runtime. In the case of dynamic loading, the function may not even be written yet. A similar problem occurs with exceptions in object oriented languages, where the preferred solution, in for example Java, is to require the exceptions of a function in a subclass to be an (improper) subset of the exceptions of the corresponding function in the superclass. This technique translates to our alerts as well: We can add a declaration about alerting to the type declaration of the function pointer, i.e. the function pointer declaration now also declares the alert mechanism for the function it will eventually point to, and type checking on the function signature, with alert declarations, must be performed when function pointers are assigned to. Arguably, this will make function pointers even more difficult to read, but these syntactical issues can be remedied by judicious use of `typedef`s. Our current implementation does not yet support this.

### 5.5.3 Compiling to Aspects

The application of alert handlers to function call sites is a separate, cross-cutting concern, and can certainly be considered an aspect in the sense used by Kiczales et al [Kiczales et al., 2001]. If we targeted AspectJ rather than C, the template in Figure 5.8 could be realized as an around advice, where the invocation of $f$ is replaced with a call to `proceed`. The `pre` and `post` condition expressions would be placed before and after the call to `proceed`, respectively, in the same fashion as now. `Use` statements in the handler body would translate into `return`s.

The full granularity of handler declarations from expression level to arbitrary function groups would be harder to capture faithfully, however. While funspaces

can be captured by normal point-cuts, by listing all function names in the point-cut, we do not see any easy and robust way of encoding point-cuts that exactly match expression level handlers.

## 5.6 Related Work

Language support for exception handling has been introduced gradually since the 1960's. Research-wise, PL/I's condition system and later CLU [Liskov et al., 1977, Liskov, 1993] and Ada's structured exception handling have perhaps been the most influential.

Hill [Hill, 1970] documents about ten different idioms for raising exceptions in languages such as Algol and Fortran, which at the time did not have any exception handling facility. He advocates disciplined control transfer over special return values.

Randell [Randell, 1975], introduces a failure recovery system inspired by "stand-by sparing", as found in hardware designs. Their technique assumes a nested, block-structured language, and provides transaction-like error recovery. Each block is a transaction, and may have one or several recovery blocks associated with it. A block has an acceptance test, which acts as a postcondition check. If a block fails, by failing its postcondition check, the program state is unrolled to the state before the block was entered, and the list of associated recovery blocks is tried in order, each time preceded by an unroll, until one of the recovery blocks passes the acceptance test. If the list is exhausted before recovery occurs, the error is passed to the enclosing block. This technique does not support raising of errors to handlers further up the call chain. MacLaren [MacLaren, 1977] critiques the design for being too complex, and encouraging bad idioms, like a global **ON** handler for file exceptions which set global error flags that must be checked after by the caller of any file operation, thus effectively degenerating to global error values.

Borgida [Borgida, 1985] discusses language features for exception handling with a focus on the interplay between exception handling and transactions found in database and information systems. He advocates the support for resumption, user-defined exception types, classification of exception types, and preventing the handler from modifying the context of the alerter. The language presented supports transactional unrolling in the case of unhandled exceptions and the capture of accountability in transactions using exceptions.

In a sufficiently reflective language, such as Oberon, exception handling may be entirely implemented by the user without extending the language, as is shown in [Hof et al., 1997]. Oberon allows reflection over stack frames using "riders".

Exceptions are thrown by invoking a rider that locates the appropriate handler in an enclosing procedure on the stack. If the found handler returns, this is taken as termination, and the stack below the handler is cleaned. If the handler invokes **Resume**, execution resumes at the point of exception.

Romanovsky and Sandén [Romanovsky and Sandén, 2001] discuss good and bad practices in exception handling, dividing the problem into bad language design and misuse due to insufficient training. They argue that languages should support two kinds of exceptions with respect to their program units (modules or packages): internal and external. External exceptions must be declared and checked, i.e. a propagation discipline must be declared and the compiler must enforce it. They argue further, based on experience with Ada, that exceptions in OO-languages must be classes, and user-definable, so that information may be passed from the exception raiser to the handler with the exception, and so that the exceptions may be classified based on type. They also argue that exceptions can aid in, rather than complicate program validation and verification. The critique of Romanovsky and Sandén about a propagation discipline was addressed by Luckham and Polak [Luckham and Polak, 1980]. They describe a language extension to Ada for specifying the propagation of exceptions. This extension has not been included in later versions Ada, however.

Cui and Gannon [Cui and Gannon, 1992] describe an alternative exception handling system for Ada than the school of Goodenough [Goodenough, 1975]. Instead of being declared as part of the control structure, as markers on the stack, exception handlers are dynamically attached to objects. When an object raises an exception, its associated handler is invoked. The authors refer to this as data-oriented exception handling. Our alert system provides no easy way to support this form of exception handling.

An argument against exception handling for embedded systems – the difficulty of predicting timing constraints in the face of exceptions – is addressed by Chapman et al [Chapman et al., 1993], where the authors presents a model for static timing analysis of exceptions in a subset of Ada83.

The Lisp condition system [Pitman, 2000] is similar to PL/I's **ON**, but supports more of the features first described by Goodenough [Goodenough, 1975], such as resumption.

Other functional languages, such as SML and Haskell, also support exceptions. Wadler [Wadler, 1995] shows how exceptions may be realized in purely functional programming languages, using monads.

Dony [Dony, 1990] describes an object-oriented exception handling for Smalltalk, where users can define new exceptions, where exception objects contain information passed from the alerter, and where different exceptions can be distinguished

and organised in a hierarchy based on their type. Like Goodenough's approach, the possible action of the handler are resumption, termination, and retry, but the choice is determined by the type of the exception object, rather than the alerting primitive. Smalltalk-80 allows the declaration of class-handlers: per-class exception handlers. These do not take handlers in the dynamic call context into account, as proposed by Goodenough; this is provided by Dony's extension.

Lippert and Lopes [Lippert and Lopes, 2000] describe how to untangle error handling code from algorithmic code using aspect-oriented programming. The solution is applied to a framework constructed using design by contract. Aspects are used to extract the contract checking code from the rest of the framework. A drawback of the proposed solution is that the contract is declared in documentation comments along with the framework (algorithmic) code, whereas the contract checking code is maintained elsewhere, thus opening up for deviations between the declared contract and the actual contract checking code.

## 5.7 Conclusion

The state of the art in alert handling provides reporting mechanisms that are both efficient and expressive. However, with the exception of the aspect-oriented approach to separating out failure behaviour [Lippert and Lopes, 2000], failure handling code is largely rigid. The alert reporting and handling are tangled, and the implementer must always choose a mechanism when implementing a function, but in doing so, also makes an implicit choice about the handling policy.

We have presented a flexible alert language extension that supports decoupling of the reporting and handling mechanisms for exceptional behaviour. The extension user-defined alert handling and reporting at a wide range of granularities. It allows the caller to declare what is normal and what is exceptional, and to declare separately the desired handling policies. This improves reuse of existing libraries and components, as policies can now be specified retroactively by the library user. We have sketched its implementation [C-Alert] based on the Transformers program transformation framework. The extension functions as a compiler extension in the form of a pre-processing step to the C compiler. The design space where the alerter and handler are decoupled is largely unexplored. This is unfortunate, since even the modest extensions we have shown to a simple, imperative language with exceptions could improve both the reuse of existing code bases and the clarity of failure handling code. Some work on this topic has been done in the context of aspect-orientation, but we believe that our alert declaration language is more precise and concise than generic join-points

and advice. One could consider our language extension as a domain-specific aspect language for error handling.

While our proposed extension has only been realized for two rather simple languages, in the imperative style, we expect it to be transportable to other languages and paradigms. The syntax should be modified to fit the conventions of the language; in this paper, we have based the syntax on C-like languages; this would be out of place in Python, for instance. A few obvious extensions may be necessary. In general, funspace patterns may need to be more like AspectJ [Kiczales et al., 2001] or AspectC++ [Spinczyk et al., 2002] point-cuts, to deal with function overloading and namespaces. In our subject language, C with exceptions, this was not needed, since we only have one global namespace and no overloading. It is important that funspaces should continue to be function (or method) groups, so that funspaces can cross-cut namespaces. This will keep the flexibility of specifying alert mechanisms and handlers across namespaces. More research is necessary to determine the best interaction between funspaces and method visibility, however.

```
alert {PrecondFailure, Whoops};
on PrecondFailure in * {fatal("Precond␣failed");}

int f(int x) alert PrecondFailure pre unless(x > 0)
               alert Whoops post(value > 10);
int ff(int a, int b)
{ on Whoops in f() {print("whoops!"); use 0;}
  return f(f(a));
}
```

```
int f(int x);

int ff(int a)
{ int r;
  if(a > 0)
  { r = f(a);
    if(r > 10) { print("whoops!"); r = 0; }
    if(r > 0)
    {  r = f(r);
      if(r > 10) { print("whoops!"); r = 0; }
    } else fatal("Precond␣failed");
  } else fatal("Precond␣failed");
  return r;
}
```

Figure 5.6: Comparison of the Alert extension to C (top), and the corresponding normal C code (bottom).

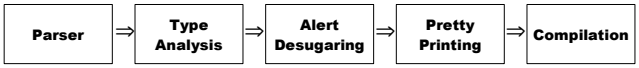



Figure 5.7: The compiler pipeline for our extended C language.

```
t0 r;
{ t1 v1 = e1; ...
 if(<precond-f(v1, ...)>) {
  <current-handler>;
 } else {
  r = f(v1, ...);
  if(<postcond-f(r)>) {<current-handler>};
} }
```

Figure 5.8: Template for desugaring function calls.

# Chapter 6

# Implementing ConceptC++

*Concepts* are a widely expected extension for C++. ConceptC++ was proposed as an extension in C++0x but has then been rejected. Though concepts are very useful when using the template mechanism of C++. But it is also a main-stream extension that proposes a syntax for axioms. There are several kind of program transformations that might need axioms: optimization, testing, verification. It is interesting for these transformation to have an unified way to express those axioms.

Experimenting with concepts was then very important. Unfortunately, it was far too new to be able to have tools for this kinds of experimentation. Not many compilers handle ConceptC++, and it is mostly experimental. In this paper we showed how we can de-sugar ConceptC++ to pure C++ with a very simple translation. This technique tries to look for a context-free transformation. In this way we reduce the cost of engineering. It is not hard to do so since we can generate template meta-programs that are handled later on by the compiler. With this technique, we cannot achieve the full ConceptC++ language. But we can approach a fairly useful version for experimentation.

This paper will be presented at SCAM'09.

# Concepts as syntactic sugar

VALENTIN DAVID MAGNE HAVEERAEN

*In the proceedings of Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09), September 20-21, 2009, Edmonton, Canada.*

**Abstract.** The coming standard for C++ will contain language extensions. It can be expected that there will be some years between the release of the new standard and the time when most compilers will be compliant, like it happened for the 1998 standard. Concepts are an extension proposed for the new standard. We show how we can translate ConceptC++ code into pure C++03 using the C++ template mechanism. Such a translation tool could be used for example to port software written using a ConceptC++ compiler to architectures having only older compilers. Or a library written using concepts could be used by a project written in pure C++. The goal of this transformation is not to provide all capabilities of ConceptC++, but with restrictions we can afford, to be as simple as possible.

## 6.1 Introduction

C++ has been standardized in 1998 then revised in 2003 [WG21, 2003]. The standard committee is now preparing a new version [Becker, 2009]. This version will be more than a simple revision, and it brings new features and reinforces existing ones [Stroustrup, 2005a]. One of these features needing improvement in C++ is generic programming. C++ provides a construct for it called *template*. It allows writing reusable code with better performance than using the object-oriented programming technique, which in C++ needs dynamic binding. For example, the Standard Template Library [Stepanov and Lee, 1994] has been built around generic programming.

**© 2009 IEEE. Reprinted, with permission, from the proceedings of Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, Concepts as syntactic sugar, Valentin David, Magne Haveeraen**

But C++ has confronted a problem of typing. A template[1] in itself cannot be type checked. Only its *instantiations* are type checked. This means, each time the template is used with concrete types as parameters, the template is type checked for these given types. Then the concrete type can be checked to have the necessary operations used inside the template. A compiled program is thus type safe as the instantiations are done at compile-time.

Programming with templates evolved to meta-programming. Which means the template level is a meta-program evaluated at compile-time. Compiling a meta-program with inappropriate template arguments leads to inconsistencies in the instantiations, and the meta-program user would end up with complicated error messages pointing to the internals of the template where a type is missing a certain operation. Figure 6.1 shows a simple template function that will generate several error message about missing functions if used for `list<int>`. Rather we want the error message to point to the invalid template arguments of the meta-program.

```
template <typename T>
T sum(const list<T>& l) {
  T s = identity_elt();
  for (typename list<T>::const_iterator i
       = l.begin(); i != l.end(); ++i)
    s = op(s, *i);
  return s;
}
```

Figure 6.1: Example of template function

Concepts are introduced in the new C++ standard to state requirements on template arguments [Siek et al., 2005]. Then meta-programs (and not only instantiations) can be type-checked, also making it possible to check template arguments against the requirements. The compiler can then produce error messages on invalid uses of a meta-program [Dos Reis and Stroustrup, 2006, Gregor et al., 2006b].

This paper is structured as follows. In the next section we present ConceptC++. Then we motivate the need for a transformation tool. In section IV we specify our requirements for a simple transformation tool. Sections V-VII describes three different transformations tools, each dealing more accurately with

[1]The syntax of C++ is deceptive and can lead us to think of a template as only its list of type arguments. But a template is actually the whole definition: a class template or a function template.

a larger part of ConceptC++. The following section describes how to add better error reporting in the translated code. Finally we sum up this work and conclude with our experience.

## 6.2 ConceptC++

A *concept* is a set of syntactic requirements on a set of types. It can require *associated types*, *functions*, *members*, *operators* or other sets of requirements (other concepts). A concept can also declare semantic requirements with *axioms*, but the compiler does not have to use them. In Figure 6.2, `Monoid` is a concept. It is a set of requirements on a type parameter `T`. The requirements here are a function called `identity_elt` and a concept, `Semigroup` on `T`. `Semigroup` defines the function `op` (and axioms on it) that is used in the axiom `Identity`.

```
concept Semigroup<typename T> {
  T op(T, T);

  axiom Associativity(T a, T b, T c) {
    op(a, op(b, c)) == op(op(a, b), c);
  }
}

concept Monoid<typename T> {
  requires Semigroup<T>;
  T identity_elt();

  axiom Identity(T a) {
    op(a, identity_elt()) == a;
    op(identity_elt(), a) == a;
  }
}
```

Figure 6.2: Example of concept definition

A *concept map* describes how a set of types fits with the requirements of a concept. A concept map is a modeling relation. For a set of argument types, it defines how they relate to the concept, if needed by explicitly declaring the associated types and functions. The concept map of Figure 6.3 defines a model for `Monoid` on type **`int`** and defines the required function `identity_elt`. Note that a concept map for `Semigroup` (and the required function `op`) also had to be defined.

```
concept_map Semigroup<int> {
  int op(int a, int b) { return a + b; }
}

concept_map Monoid<int> {
  int identity_elt() { return 0; }
}
```

Figure 6.3: Example of concept map

Concepts may provide default values for (some of) its required associated types and functions. A concept map can then use the defaults, or explicitly redefine one or more of them.

For a normal C++ template definition, *requirements* on the *type parameters* can be described using concepts. Then all required associated types and functions are accessible in the scope of the concept, and members are guaranteed to be accessible from the types they are associated to. The function template defined in Figure 6.4 requires the polymorphic type `T` to have a concept map `Monoid` defined. Then error messages due to inappropriate type arguments can be flagged as such. In our example, the concept maps from Figure 6.3 ensures that `list<int>` is a valid argument for the function template.

```
template <typename T>
  requires Monoid<T>
T sum(const list<T>& l) {
  T s = identity_elt();
  for (typename list<T>::const_iterator i
       = l.begin(); i != l.end(); ++i)
    s = op(s, *i);
  return s;
}
```

Figure 6.4: Example of constrained function template

A concept may be declared with the keyword *auto*. Then a concept map is automatically implied for all types that match its (syntactic) requirements. This is useful when needing to test the presence of specific operations for a template. Auto concepts should not be used when semantic requirements (axioms) are present. On figure 6.5, the concept *DefaultConstructible* is available for any type having a default constructor.

```
auto concept DefaultConstructible<typename T> {
  T::T();
};
```

Figure 6.5: Example of auto concept (from the standard library)

A concept can *refine* other concepts. This looks similar to inheritance in object-oriented programming. A refinement consist of extending the set of requirement with the ones from the refined concepts. It differs from requiring a concept, as we do not need a separate concept map to be defined on the refined concept, see Figure 6.6.

```
concept MonoidRefines<typename T>
  : Semigroup<T> {
  T identity_elt();

  axiom Identity(T a) {
    op(a, identity_elt()) == a;
    op(identity_elt(), a) == a;
  }
}

concept_map MonoidRefines<int> {
  int op(int a, int b) { return a + b; }
  int identity_elt() { return 0; }
}
```

Figure 6.6: Example of refinement of a concept and a concept map

## 6.3 Motivation for a ConceptC++ to C++ transformation tool

When the first C++ standard came out in 1998, it took several years for compilers to get a good level of compliance. Three years after the standard, Sutter stated "there is no C++ compiler or library today that implements the Standard perfectly" [Sutter, 2001]. GNU Compiler Collection (GCC), for example, could have been considered as compliant at a very early stage. But parsing bugs remained, such that in 2004, after 6 years, the parser had to be replaced. Microsoft on its

side, did not implement partial template specialization before the 2003 version of their compiler, yet it is considered an important feature of C++. The export feature for templates has been supported by only a few compilers, and only after a long time. Sutter argues that this feature is just not wanted [Sutter, 2003].

Thus, we can expect to wait for quite a few years before most C++ compilers support C++0x. Some compilers might rush to provide this extension, but certainly not all of them. There are good reasons we want to be able to use concepts before it is fully supported everywhere.

### 6.3.1 Software development

ConceptC++ offers more than just checking the template level of a source. Concepts can be viewed as algebraic specifications [Haveraaen, 2007, Zalewski and Schupp, 2007]. A concept declares sorts (the type arguments and the associated types), operations (functions, members, operators), and axioms. A concept map declares how concrete types model the concept. Our experience with integrating algebraic specifications in the software development process has given good results in the design of software for the PDE domain [Haveraaen et al., 1999, Haveraaen and Friis, 2009]. Gottschling developed specification of fundamental algebraic properties through concepts [Gottschling, 2006], and used this as background for the design of the numerical library *MTL4*. MTL4 is expressed in ConceptC++, but maintained in parallel versions for ConceptC++ and C++.

Algebraic specification are not limited to syntactic checking. For example Bagge et al. [2003], Tang and Järvi [2007a] used axiom declarations in C++ to perform domain-specific optimization. Another application is axiom-based testing. Instead of writing unit tests, these can be generated from the concept axioms [Bagge and Haveraaen, 2008].

It is more convenient to use the ConceptC++ syntax for such purposes than to express axioms in some *ad hoc* notation. Unfortunately, even if the ConceptC++ syntax is generally backward compatible to C++, when you start using concepts, it modifies the semantics of the code by changing the scope for name resolution, making interoperability difficult.

### 6.3.2 Shipping a generic library

We certainly would like to develop libraries, such as MTL4, using the concept feature while allowing backward compatibility with C++03. A library is a piece of code that might be reused by different users using different compilers. We cannot rely on distribution of binaries: these are not portable, and C++ generic programming does not even support a separate compilation scheme.

In this strategy, a user must be able to take the translated code from the library and incrementally add new declarations tying his code to the library code, for instance, a concept map for a user defined type written in C++03 to a library concept that already has been transformed. The tie-in code must then also be translated to end up with pure C++03 code.

## 6.4 Specification of the transformation tool

We want a transformation tool that will take ConceptC++ code and produce C++03 code as output. The tool should also leave C++03 code untouched.

The first idea would be to simply hide concept features using the preprocessor. But we will see in section 6.5 that this is not a solution. The transformation needs to duplicate code from ConceptC++ declarations to make them available as class templates and class template specializations.

To make the transformation tool as simple as, and therefore as correct as, possible, we want it to need no or very little context from analysis of ConceptC++ code. Full fledged parsing and analysis of C++ code is extremely difficult, and dealing with a C++ extension is not making this easier. So we simplify the transformation tool by pushing as much context related analysis as possible into the target C++03 code, using meta-code, like type traits or other template meta programming techniques. In this way we can forward type checking from the source ConceptC++ code to the target C++ code. Our aim is to transform each definition in our ConceptC++ program source into a set of pure C++ definitions independently from the context. This is needed if we want to support the library shipping strategy. In practice we did not find a way to resolve the context problem perfectly. But we can still reduce significantly the transformation's dependence on semantic analysis.

Note that we are looking for a "context-free" translation, i.e., the translation tool only needs to relate to the part of the parse tree it is about to transform. Parsing C++ itself is not context-free, but that problem is exterior to the transformations themselves.

The aims of a context-free translation can be summed up in two simple axioms. For a C++03 code $a$, we have $tr(a) = a$. For declarations $a$ and $b$, we have $tr(a + b) = tr(a) + tr(b)$.

Note that the first axiom induces idempotence ($tr(tr(a))$), since the output of $tr$ is C++03 code.

Unfortunately it is a hard task to check the validity of the input code. Our transformation tool is not trying to replace compilers, and the simplicity of our tool design makes it unable to do code validity checking such as type-checking.

So errors in the source ConceptC++ code may be very hard to detect unless the original source code can be checked with a proper ConceptC++ compiler.

In the next sections, we show three levels of completeness in how to transform ConceptC++ into C++. The first is the naive transformation that seems intuitive but does not work. The second level provides a much more acceptable solution. The last level, which is not completely context-free, deals with constraint-based overloading on function templates.

## 6.5 Transformation: A naive approach

Simply stripping out concept declarations and constraints does not make a ConceptC++ source become pure C++. As observed in Gregor et al. [2006b], concept constraints have two roles: they act as predicates, and they provide a scope for name resolution. If you consider the code in Figure 6.7, `identity_elt` will be resolved as `Monoid<T>::identity_elt`. Removing the concept language extensions will make it impossible to find the right `identity_elt`, as in this example, it may be defined inside a concept map that will disappear.

Simple removing of requirements is not possible, we need to provide declarations emulating the name resolution of ConceptC++.

## 6.6 Transformation: Context free approach

ConceptGCC implements concept declarations as class templates, concept maps as specialized templates and concept map templates as partial specializations [Gregor et al., 2006b]. Refinement and constraints are translated into virtual public inheritance.

A program transformation can easily use this approach. In contrast to a compiler, the program transformation should be able to generate code that can be compiled again. Because of that, the transformation will generate artifacts that a compiler does not need.

We deal with each ConceptC++ construct in turn. First, the translation of concepts and concept maps, 6.6.1 through 6.6.3. These are translated to template classes. Finally, the translation of concept constraints in templates, 6.6.4 through 6.6.5.

### 6.6.1 Concepts

A concept `A` is translated to two class templates, `A` and `A_requirements`. In a pure C++ program, the members of a concept will not be accessed, so the

```
template <typename T>
  requires Monoid<T>
T sum(list<T> l) {
  T s = identity_elt();
  for (typename list<T>::const_iterator i
     = l.begin(); i != l.end(); ++i)
    s = op(s, *i);
  return s;
}

concept_map Monoid<int> {
  void identity_elt() { return 0; }
}
concept_map Semigroup<int> {
  void op(int a, int b) { return a+b; }
}
```

↓

```
template <typename T>
T sum(const list<T>& l) {
  T s = identity_elt(); //not declared
  for (typename list<T>::const_iterator i
     = l.begin(); i != l.end(); ++i)
    s = op(s, *i); //not declared
  return s;
}
```

Figure 6.7: A naive transformation

generated class template `A` can be empty. This class will be specialized capturing dependencies between concepts.

The template `A_requirements` represents the concept's requirements, and will be filled with relevant declarations. This class will never be specialized. It will capture dependencies between concepts by inheritance.

The reason for this splitting is to be consistent with the semantics induced by the concept maps and their translation, see subsection 6.6.2.

#### Axioms

Axioms do not affect the semantic of the code, and they are treated optionally by compilers. They are even not accessible as members. We just do not translate

them.

#### Declared functions and types

Functions and types that are only declared inside a concept are there solely for checking their presence when doing concept checking. These can therefore be removed during the transformation process.

If a default definition is provided, however, the transformation is more involved. See section 6.6.1.

#### Requirements

```
concept A<typename T>
 { void foo(T); }
concept B<typename T>
 { requires A<T>; }
template <B T> void bar(T t)
 { foo(t); }
```

Figure 6.8: Influence of constraints on the scope

A concept can use other concepts to provide requirements. We will translate such requirements using inheritance, but need to be careful to get scoping and context issues right.

Consider the example on Figure 6.8. Function `foo` is accessible from `B`. We can deal with this using virtual public inheritance. We need virtual inheritance because we might have "diamond-shaped" requirements coming from multiple layers of multiple requirements.

Any concept map will depend on these requirements. It means any class template specialization corresponding to a concept map will need to inherit from requirements that are specified in the concept definition. The specialization corresponding to the concept map will inherit `B_requirements`. See Figure 6.10 for the transformation of concepts with requirements, and refer to Figure 6.9 for the concept map translation.

#### Refinements

A refinements is when a concept extends another, such that all requirements and default declarations of the extended concept becomes part of the extension. This translation is very similar to that of the requirements declaration,

```
concept_map A<B> {
  void foo() { }
}
```

↓

```
template <>
struct A<B>
: virtual public A_requirements<B> {
  static void foo() { }
};
```

Figure 6.9: Transformation of a concept map

```
concept B<typename T> {
  requires A<T>;
}
```

↓

```
template <typename T>
struct B_requirements: virtual public A<T> {};
template <typename T> struct B {};
```

Figure 6.10: Transformation of constraints as virtual public inheritance on concept maps

see Figure 6.11. The difference is that here `B_requirements` inherits from `A_requirements`, while above it inherited from `A`.

#### Default definitions

Concept definitions may define some members, *i.e.*, give default definitions. If a concept map later on does not define the required member, then the default implementations will be used. We place these definitions in the `_requirements` class, where they will be found by inheritance, if not defined in the scope of the translated concept map.

This transformation is shown on Figure 6.12. Functions need to be declared static. A static member is associated to its class and not the objects of its class. Thus a static function is called like a normal function, i.e., without prefixing it

```
concept B<typename T> : A<T> {
}
```

↓

```
template <typename T>
struct B_requirements:
  virtual public A_requirements<T> {};
template <typename T>
struct B {};
```

Figure 6.11: Transformation for refinements

with an object.

Unfortunately, it is not possible to translate default implementations on members of a type, since we cannot rewrite parameter classes (e.g., **typename** `T::type =` This gives a restriction on the flexibility of our approach.

#### Associated types

An associated type of a concept that is dependent on template parameters, once translated need to be explicitly tagged with a keyword **typename**. This keyword is used for resolving ambiguity in the source code. But the ambiguity is not present on members of concepts, but becomes ambiguous on a class. The associated type becomes a member type of a class template, and its kind (type or field) is needed for parsing in pure C++, but was not needed in ConceptC++. This may require a bit more of analysis in case the dependence on parameters is deeply hidden. For example, an intermediate type dependent on a parameter is defined. Then this type is used as parameter of a concept. The idea is that the parse tree for C++ should give enough background on the kind of each entity (is it a type name or a concept name?).

### 6.6.2 Concept maps

A concept map translates to a template specialization of the modeled concept. And by extension, a concept map template translates to a partial template specialization. Members of the concept map have to be copied to the specialization, any function definitions becoming static methods. Requirements embedded in the concept are captured by virtual public inheritance. Figure 6.9 shows this process.

```
concept A<typename T> {
  typename U = T;
  void f(T) { /* ... */ }
}
```

↓

```
template <typename T>
struct A_requirements {
  typedef U T;
  static void f(T) { /* ... */ }
};

template <typename T>
struct A {
};
```

Figure 6.12: Transformation for default definitions of concept members

Note how the two different strategies for translating concept requirements and concept refinements are compatible with their semantics and for this concept map translation.

#### Operator declarations

Operators are a bit special as they cannot be declared as static members in C++. But they are allowed as members of concept maps. Operator calls have to be translated to use functions calls which are legal as static members of classes. The problem then resides on knowing when we need to rename. To remain context-free, all calls have to be translated. Then the call has to be resolved to the right operator. This can be done with meta programming techniques using a wrapper function. The piece of code on Figure 6.13 can be used to resolve the operator. If we need to attach an operator to a new scope, we just have to instantiate with the proper `Scope` value. This value is a type that symbolizes the scope of the concept map. It can be the generated template specialization itself. Then we need to declare a static function template in the scopes that calls this generic "engine". Figure 6.14 shows how to define an operator in a given scope. Once this is done, all use of the operator can be safely transformed to a call of the wrapper (here it is `opplus`).

```
template <typename Scope,
          typename T,
          typename U>
struct opplus_tools {
  static typename
     return_of_opplus<Scope, T, U>::res
     call(T t, U u)
  {
    return t + u;
  }
};
```

Figure 6.13: Default operator resolution

### 6.6.3 Auto concepts

Since auto concepts act both as concepts and concept maps, their members can be accessed. But auto concepts are valid only if the members already are declared in the scope. So the class resulting from an auto concept should be empty.

Consider Figure 6.15. If `A<T>` for a given `T` is automatic (there is no concept map defined), it means function `foo` and method `fum` have been declared in the scopes of `A` and `T`, respectively.

That `foo` has to be resolved from the scope where `A` is declared, also mean that if we use `A` in another namespace, we need to make sure we can resolve the function there as if we were inside the namespace of `A`. One possibility could have been to define a static member in the generated class template `A`. This is not compatible with auto concepts, since `foo` can be declared after the definition of the concept. Then `foo` would not be resolved within the static function. The proper solution is that each time we have declared requirements on an auto concept, we need to insert a **`using namespace`** in the body of any restrained template.

We have a limitation here in case the source code has a **`using`** to the concept, then it might be used unqualified without having access directly to other of components its namespace. In this case, the transformation would need to look at the context, requiring full context analysis.

```
template <>
struct return_of_opplus<C<A>, A, A> {
  typedef A res;
};

template <>
struct opplus_tools<C<A>, A, A> {
  static A call(A, A)
  {
    return A();
  }
};

template <>
struct C<A> {
  template <typename T, typename U>
  static typename
    return_of_opplus<C<A>, T, U>::res
    opplus(T t, U u)
  {
    return opplus_tools<C<A>, T, U>
      ::call(t, u);
  }
};
```

Figure 6.14: Providing another implementation for an operator

### 6.6.4 Constrained class template

Class templates, with type parameters constrained by concepts, are transformed to inherit from corresponding requirements class template, such as in section 6.6.2.

This induces an unfortunate restriction on the translation. Method definitions (inside the class template) with a parameter that depends on a template parameter, cannot use elements from the virtual public inheritance. Figure 6.16 shows an example where inheritance will not work. In this example, `bar` is not declaring any parameter, so it will never look at inheritance using template parameters. If `foo` was declared inside `A<T>`, it will not be found. With GCC, it is possible to avoid this problem using the command line parameter "`-fpermissive`". But this would anyway make the generated code non-portable. The other solution is to automatically add a parameter, rename

```
auto concept A<typename T> {
  void foo();
  void T::fum();
}
```

Figure 6.15: Auto concept

the method and create a method calling this extracted method with a dummy argument.

```
template <typename T>
struct A {};

template <typename T>
struct B: public virtual A<T> {
  static void bar() {
    foo(); //foo will not be resolved A<T>::foo
  }
};
```

Figure 6.16: Non dependent method problem

### 6.6.5 Constraint function templates

Free functions cannot inherit, so we have to wrap a template function into a class template to capture concept requirements. But we cannot change the call convention, since it would break the automatic inference of type arguments on function template calls. To handle this, the translation keeps the function, but its body just calls the generated wrapped function.

In Figure 6.17, the compiler can still infer the argument type T. The template function `foo` will know this type and will pass it to the wrapper class `foo_function` which provides the original function body as a static method.

In ConceptC++ it is possible to overload functions based solely on constraint templates. Dealing with this requires some context sensitivity, but within the limits of our simple translation tool.

```
template <typename T>
  requires A<T>
void foo(T a) {
  //body...
}
```

↓

```
template <typename T>
struct foo_constrained
  : virtual public A<T> {
  static void def(T a) {
    //body...
  }
};
template <typename T>
void foo(T a) {
  return foo_constrained<T>
    ::def(a);
}
```

Figure 6.17: Constrained function templates

## 6.7 Transformation: Dealing with overloading of function templates

Constraints can be used as predicates for overloading resolution of function templates. When the overloaded function call is resolved to a function template, the requirements are tested. If they are met, the resolution is valid, if not, the compiler will look at the other declarations. To preserve this behavior, we need to create a type trait that maps a type argument to a Boolean value indicating whether there is concept map. This technique does not work for auto concepts, since we need to know when there is a concept map available. Then we need a wrapper function that tests the different predicates and finds the right version given by an integer. To do this and remain (almost) context-free, we can have such a test using template meta-programming techniques passing the template arguments to a meta program that tests the predicates and finds the right version. Then call the right function definition through a specialized class template. Figure 6.18 shows step by step how this translation is done.

This transformation is a bit complex and not easy to make to work with the

rest of the translation. It is not completely context-free as we need to know the number of functions defined. But we could fix this by defining and redefining macros for the preprocessor. We also need to make sure it is idempotent, so we need to check for use requirements in other function templates of the same name.

## 6.8 A better feedback on inconsistent uses

In the library shipping approach, where a ConceptC++ library is shipped as C++03 code, we would like to provide intelligible feedback, when the user is trying to compile his code and has errors or violations of concept requirements. There are basically two cases. The first would be where the user needs to define a concept map, but forgets it. The second is where the user provides the concept map, but does not define all the needed declarations to fulfill the concept requirements. Recall that due to the properties of the translation tool, the user can add concept maps to his C++ code, transform it, and expect it to be correct C++ code and interact correctly with the library.

If the user forgets the concept map entirely, we need the compiler to complain about this. This can be done by forbidding implicit instantiation of the class template that correspond to the concept. Figure 6.19 is a simple example that can address this problem. Here, `Foo` is a sketch of the class template generated from a concept `Foo`. When `Foo` is required somewhere, it will appear as inheritance. If `Foo` is required on a type, and there was no concept map on it, then the template will be instantiated. Then the type member `t` will be instantiated, instantiating `there_is_no_concept_map<Foo<T> >`, and try to look for member type `here`, which will not be found. Most compilers will complain they cannot find it, displaying `there_is_no_concept_map<Foo<T> >`, where `T` is expanded to the concrete type. Then the user should be able to deduce exactly what is missing.

The second case is for instantiation of an auto concept, or a concept map written by the user, but not providing all the operations needed. Siek describes how to deal with such checking in Siek and Lumsdaine [2000]. To do so, each concept needs to have a class template, containing a method that has to be instantiated (but not called). The approach in the paper demands two things: a call to the constraint methods from the other concepts (basically the constraints of the concept), and a call of each possible operation. This approach does not actually check for the return type to be the correct one. But it is possible to do so by declaring special functions taking as parameter the correct type, and passing the testing operation call as parameter to this function. However this last case

```
concept A<typename T> {}
```
→
```
template <typename T>
struct A {
  enum { test=false };
};
```

```
concept_map A<B> {}
```
→
```
template <>
struct A<B> {
  enum { test=true };
};
```

```
template <typename T>
void foo(T) {
  //body...
}
```
→
```
template <int N, typename T>
struct foo_select:
  public foo_select<N-1, T> {};

template <int A_Version>
struct foo_resolve {
};
template <>
struct foo_resolve<0> {
  template <typename T>
  struct foo_function {
    static void def(T) {
      //body...
    }
  };
};
template <typename T>
struct foo_select<0, T>:
  public foo_resolve<0> {};
```

```
template <typename T>
  requires A<T>
void foo(T) {
  //body...
}
```
→
```
template <>
struct foo_resolve<1> {
  template <typename T>
  struct foo_function
    : virtual public A<T> {
    static void def(T) { //body... }
  };
};
template <typename T>
struct foo_select<1, T>:
  public If<A<T>::test,
     foo_resolve<1>,
     foo_select<0, T> >
{};
```

*before the first use of* `foo`
→
```
template <typename T>
void foo(T a) {
  return foo_select<T,1>
   ::template foo_function<T>::def(a);
}
```

Figure 6.18: Transformation of overloaded functions

```
template <typename T>
struct there_is_no_concept_map {
};

template <typename T>
struct Foo {
  typedef typename
     there_is_no_concept_map<Foo<T> >::here t;
};
```

Figure 6.19: Forbidding implicit concept

has not yet been tested combined with the transformation.

## 6.9 Conclusion

We have described three levels of transformations from ConceptC++ to plain C++. A naive transformation, known not to work, where we would just strip away the concept extension. A more comprehensive, context free transformation, that in general gives scoped name resolution similar to ConceptC++. And a higher level where we also can handle overloading, but requiring a little context information (just the counting function declarations). We also indicated how to produce code that would give relevant error messages from C++ for invalid template arguments.

Unfortunately, ConceptC++ is not completely handled by the proposed transformation. Some points are difficult due to the restrictions on the target language of the transformation. For example, it is impossible to define type members outside of the definition of the class it belongs to. This example of restriction makes default definition of types hard to implement. Also some new features of ConceptC++ are so different from C++ that it is hard to find a transformation dealing with all cases. For example, we have shown how to simulate overloading of restrained function templates. But this example breaks when using it combined wit auto concepts. This implied that the final choice of implementation using the different techniques will never handle the whole language, but at least a major part of it.

Another problem we discussed in section 6.4 is we expect to have as input a source that is type safe. Of course the transformation will not break the type safety, but the errors will be found by the compiler after the translation. We can of course expect to use a front-end doing this. In our implementation, it did not. We can of course keep track of line information in the output source, which

would not only ease fixing type errors, but also debugging. But the error reports of the compiler will still not be perfect.

We have implemented the context free transformation tool. It was implementing using Transformers C++ [Anisko et al., 2003]. This front-end is still incomplete, but provides sufficient extensions for ConceptC++ [David, 2008]. Our tool is foremost limited by this front-end, but also constrained by the limitations of the context free translation technique itself. For instance, we cannot translate default members, nor do we have the right namespace import for a requirement on a concept that is imported from a different namespace with a "**`using`**".

We have used our tool to support concept based testing [Bagge et al., 2008]. The setup is to transform concept axioms to unit tests (still expressed in ConceptC++ due to the name resolution issues), and run these to check the implemented code. This required a ConceptC++ compiler, but the difficulty running this compiler on our preferred hardware prompted us to develop the ConceptC++ to plain C++ transformation tool described here. Our experience is that the testing cycle is much faster using our tool than the prototype ConceptC++ compiler.

Unfortunately, this is tool has not been tested on very huge projects yet. It appears that the code base using ConceptC++ is quite poor. And those starting to use it do not provide yet a full interface using concepts. Thus, the scalability is hard to evaluate. Moreover, no real coding conventions have been adopted by the community concerning concept programming. It is then hard to know what features are the most important to handle. For example, will overloading using constraints be widely used? In our implementation, we did not use this technique since it introduces impurity, when our code base did not need such a feature.

Moreover, the limitations of the front-end might become a problem. For the moment we did not have any. ConceptC++ code is usually nicer as it uses less meta-programming techniques. But it might not be working well when faced to a project using several programming paradigms.

We also see the value of our tool for maintaining libraries such as *MTL4*. MTL4 is a numerical library expressed in ConceptC++, but maintained in parallel versions for ConceptC++ and C++, since almost all users are constrained by C++03 compilers. Our tool could simplify this, by limiting maintenance to the ConceptC++ version. Then C++ releases can be generated when needed, and these releases would seamlessly integrate with the user's plain C++ code.

# Chapter 7

# ConceptC++ for categorical constructions

In the following paper we try to identify categorical construction by describing concepts as specifications. We also points some missing features that could ease considerably the use of concepts for building specifications using standard specification operations. And we propose some examples of syntax for these features.

Concepts are like algebraic specifications, it is possible to make unions of specifications. We also identify signature morphisms in different constructions and show by combining them, we can write all kind of morphisms. But the technique is not that easy. Moreover writing signature morphism implies here to claim for satisfaction.

But concepts do open C++ for another programming paradigm: algebraic specification structuring.

# Concept Morphisms in C++

VALENTIN DAVID MAGNE HAVEERAEN

*Not published*

**Abstract.** With the proposed extension of C++ with concepts, C++ is becoming an institution: Signatures (concepts and plain declarations), sentences (concept axioms), models (concept maps and plain definitions) and satisfaction relation ("oracles" relating concept axioms and models). Here we detail the relationship between ConceptC++ and institutions, showing the importance of concepts and concept morphisms.
We also investigate how ConceptC++ supports the classical institution independent structuring specification mechanisms, e.g., how to build a ring concept by combining concepts for monoid, group and distributivity. We discuss several patterns for putting concepts together, some which requires minor extensions to ConceptC++, some which work but may not be ideal for other reasons.

## 7.1 Introduction

The notion of concepts in C++ integrates a specification formalism in a mainstream language [Dos Reis and Stroustrup, 2006]. This gives extra power to the software development process, e.g., as seen in the work on extended ML [Sannella and Tarlecki, 1986, 1991].

Certain issues with concepts in C++ may hinder the exploitation of this power. To see this, we relate ConceptC++ to the notion of institution [Burstall and Goguen, 1977]. Being an institution ensures certain nice properties of a specification formalism, e.g., in the way axioms from different parts of a large specification interacts, and also opens up an important set of institution independent mechanisms for building structured specifications. These are important

**This work has been submitted for publication. Copyright may be transferred without further notice and the accepted version may then be posted by the publisher.**

when building of large specifications in a modular way, and ensures reusability of existing specifications.

When looking at the details of the relationship between ConceptC++ and institutions, we notice that this becomes a many-to-many relationship. A construct in ConceptC++ may map to more than one component of the institution, and many different ConceptC++ constructs map to the same institution component. This investigation extends that of Haveraaen [2007] and Zalewski and Schupp [2007], which each identified parts of this larger relationship between ConceptC++ and the institution notion.

Looking at this in detail, we expose a weakness in the way signature morphisms are handled in ConceptC++. Signature morphisms are crucial for forming what may be termed concept morphisms - the mapping between concepts (specifications). Such mappings define our ability to reuse existing concepts in declaring new concepts, i.e. for structuring large specifications in a modular way.

This paper next gives an overview of the concept extension of C++, before it presents institutions and the standard institution independent specification structuring mechanisms. Then it details the relationship between ConceptC++ and the institution. Section five discusses several alternatives for providing concept morphisms. Finally we conclude with discussing our results.

## 7.2 ConceptC++

A *concept* is a set of syntactic requirements on a set of types. It can require *associated types*, *functions*, *members*, *operators* or other sets of requirements (other concepts). A concept can also declare semantic requirements with *axioms*, but the compiler does not have to use them. In ConceptC++ axioms have the form of equations (expressions equated using ==, whether the type has defined such an operator or not) or conditional expressions (equations being the target of an if-statement). In figure 7.1, `Semigroup` is a concept. It is a set of requirements on a type parameter `T`. The requirements here are a function called `op` and a concept, `std::Regular` on `T`. The latter requirement states that the type has no "surprising" behavior like not being comparable, throwing exception, not allocatable, not copyable, etc. While it is not absolutely necessary, it still provides some comfort with algebraic objects.

A *concept map* describes how a set of types fits with the requirements of a concept. A concept map is a modeling relation. For a set of argument types, it defines how they relate to the concept, if needed by explicitly declaring the associated types and functions. The concept map of figure 7.2 defines a model

```
concept Semigroup<typename T> {
  requires std::Regular<T>;
  T op(T, T);

  axiom Associativity(T a, T b, T c) {
    op(a, op(b, c)) == op(op(a, b), c);
  }
}
```

Figure 7.1: Example of concept definition

for `Semigroup` providing the required function `op`. The concept map may define all the operations required by a concept as we have done here. It may also omit explicitly defining a requirement if it is available in the context where the concept map is defined.

```
concept_map Semigroup<int> {
  int op(int a, int b) { return a + b; }
}
```

Figure 7.2: Example of concept map

A concept can *refine* other concepts. This looks similar to inheritance in object-oriented programming. A refinement consist of extending the set of requirements with the ones from the refined concepts. It differs from requiring a concept, as we do not need a separate concept map to be defined on the refined concept, see figure 7.3. Note how this concept map is completely independent of the concept map in figure 7.2. If we instead of refining the concept `Semigroup` (using inheritance) had required it (using the **`requires`** statement), the concept map for semigroup would define the meaning of `op`, leaving only the definition of `identity_elt` to the second concept map. Clearly not what we would want in this case.

For a normal C++ template definition, *requirements* on the *type parameters* can be described using concepts. Then all required associated types and functions are accessible in the scope of the concept, and members are guaranteed to be accessible from the types they are associated to. The function template defined in figure 7.4 requires the polymorphic type `T` to have a concept map `Monoid` defined. Then error messages due to inappropriate type arguments can be flagged as such. In our example, the concept maps from figure 7.2 ensures

```
concept Monoid<typename T>: Semigroup<T> {
  T identity_elt();

  axiom Identity(T a) {
    op(a, identity_elt()) == a;
    op(identity_elt(), a) == a;
  }
}

concept_map Monoid<int> {
  int op(int a, int b) { return a * b; }
  int identity_elt() { return 1; }
}
```

Figure 7.3: Example of refinement of a concept and a concept map

that `list<int>` is a valid argument for the function template.

```
template <typename T>
  requires Monoid<T>
T prod(const list<T>& l) {
  T s = identity_elt();
  for (typename list<T>::const_iterator i
       = l.begin(); i != l.end(); ++i)
    s = op(s, *i);
  return s;
}
```

Figure 7.4: Example of constrained function template

## 7.3 Institutions

The institution notion introduced in Burstall and Goguen [1977] gives a precise framework for understanding the relationship between specifications and models. Here we give a condensed and slightly informal presentation of the institution notion, using a presentation style that should not be too unfamiliar for software scientists. An institution is comprised of interfaces (signatures) and renamings (morphisms) between these, formulas (axioms) and models (software libraries) with transformations consistent with the signature renamings, and

finally a satisfaction relation stating when an axiom holds for a software library.

- Interface $I$ (signature): declaration of type names and function declarations (argument and return types).

  Interface morphism $i : I_1 \to I_2$: the ability to map type names and function declarations between interfaces while keeping the structure of the declarations.

- Formulas $F(I)$ (axioms): the set of all axioms $F(I)$ for an interface $I$.

  Formula morphism $F\langle i\rangle : F(I_1) \to F(I_2)$: the systematic transformation of an axiom for $I_1$ to an axiom for $I_2$ consistent with the interface morphism.

- Models (software library) $\ell \in L(I)$: data structure definitions for type names and function definitions (algorithms) for function declarations, for the given interface $I$.

  Software morphisms $L\langle i^{\mathrm{op}}\rangle : L(I_2) \to L(I_1)$: the transformation, consistent with $i$, of a library for interface $I_2$ to a library for interface $I_1$. This morphism is in the opposite direction of the interface morphism, intuitively, the interface morphism picks out those components of the library $\ell \in L(I_2)$ that we want to reuse as a $L(I_1)$.

- Satisfaction relation: defining what it means for a software library to satisfy an axiom, i.e. when an axiom holds for a software library.

  The satisfaction relation must be consistent with the morphisms: given an interface morphism $i : I_1 \to I_2$ for an interface $I_1$ to an interface $I_2$, a software library $\ell \in L(I_2)$ for the interface $I_2$, and axiom $a \in F(I_1)$ for the interface $I_1$, then the transformed axiom $F\langle i\rangle(a)$ holds for $\ell$ if and only if the axiom $a$ holds for the transformed library $L\langle i^{\mathrm{op}}\rangle(\ell)$ of $\ell$ for the interface $I_1$. (Note how the axioms are for $I_1$, while the software library being reused is for $I_2$.)

All these morphisms must be composable, i.e. interface morphisms $i_1 : I_1 \to I_2$ and $i_2 : I_2 \to I_3$ must compose to an interface morphism $i_1; i_2 : I_1 \to I_3$, and likewise for the corresponding formula and model morphisms.

The theory of institutions enables powerful mechanisms for combining specifications. Experience has shown that a software development approach based on institutions (*property based programming* [Gottschling, 2006, Haveraaen et al., 1999]), is a very powerful approach to programming. It has the potential for significant software productivity increase, while being able to benefit from software verification and testing tools, as well as support for high level optimizations [Bagge and Haveraaen, 2008].

### 7.3.1 Standard specification building operations

A specification (presentation) is the declaration of an application programming interface (API) together with the axioms describing its behavior, corresponding to the concept notion of ConceptC++.

In the context of institutions we automatically get a collection of very useful ways of structuring specifications. These allow us to build complicated specifications from simpler ones, i.e. the modularization of large specifications. Our running example will be how to extend a semigroup to a monoid and on two a group, and equip a group with the commutative property to achieve an abelian group. More interestingly, we can combine a monoid with an abelian group, provide distributivity and achieve a ring specification.

The specification building operations that follow from having an institution are normally presented as extension, union, renaming and forgetting. These are available for any institution, and are often called *institution independent structuring mechanisms* because of this.

#### Extension

Extensions come in two forms, which often are combined: the extension of the signature, i.e. adding more declarations of types and functions, and extension of the formulas, i.e. adding more axioms. Concept refinement is such an extension mechanism. Figure 7.5 shows the extension of a Monoid specification to a specification of Group, and how **`int`** may be seen as a group. In the institution setting we are guaranteed that any type satisfying the Group specification will also satisfy the specification of Monoid in the prescribed way. This is very fundamental to the extension mechanism.

#### Union, disjoint union and amalgamated union

As we want to build a specification by reusing smaller ones, we need a notion of joining separate specifications together.

Here it is important to control how the declarations from the different concepts interact. For example, if we want to build an abelian group by combining the group specification with the specification of commutative, we want the binary operation from each of these specifications to be merged into one and the same operation. In figure 7.6 we show the union between specifications Group and Commutative to form the concept Abelian Group. This examples relies on the coincidental use of the same name for the binary operation `op` in both `Semigroup` (figure 7.1) and `AbelianGroup`.

```
concept Group<typename T>: Monoid<T> {
  T inverse(T);

  axiom Inverse(T a) {
    op(a, inverse()) == identity_elt;
    op(inverse(), a) == identity_elt;
  }
}

concept_map Group<int> {
  int op(int a, int b) { return a + b; }
  int identity_elt() { return 0; }
  int inverse(int a) { return -a; }
}
```

Figure 7.5: Example of refinement extending Monoid with an extra operation and new axioms

A plain union will consider everything with the same declaration as the same operation. A disjoint union will still keep them separate. ConceptC++ provides union (through the use of refinement) and disjoint union (through the use of **requires**). Figure 7.7 shows the disjoint union between Monoid and Abelian Group to form a Ring specification by adding extra axioms. Here the binary operations from each concept is distinguishable, which is being exploited in the axiom Distributive. The concept map for **int** being a ring relies on the earlier concept maps, of **int** being a monoid for multiplication and of **int** being an abelian group for addition, for the requirements.

The plain union is common as specification building operations in specification languages. The detailed control over which declarations are to be merged, and which are to be kept separate, when combining specifications is called amalgamated union. Amalgamated union is normally achieved by combining plain union and an extensive renaming mechanism (signature morphism and associated formula transformation).

### Renaming

Renaming is useful when we want to use an existing specification in a specific context. For instance, in the context of a ring specification, we may want to rename the group operations to `plus`, `zero` and `minus` to separate them from monoid operations renamed to `times` and `one`. This would have made the

```
concept Commutative<typename T> {
  T op(T, T);

  axiom Commutative(T a, T b) {
    op(a, b) == op(b, a);
  }
}

concept AbelianGroup<typename T>:
  Group<T>, Commutative<T> {}

concept_map AbelianGroup<int> {
  int op(int a, int b) { return a + b; }
  int identity_elt() { return 0; }
  int inverse(int a) { return -a; }
}
```

Figure 7.6: Example of union between two concepts

Distributive axioms in figure 7.7 much more readable.

Concepts readily support renaming of its parameters. These are normally restricted to type parameters, see Section 7.5 for a discussion of renaming.

#### Forgetting

In many specification formalisms, e.g., conditional equations, it is not possible to specify the properties one wants for a given API precisely enough. Extending the interface with extra types and operations may achieve the target, but then these extra declarations should be forgotten to get back the original interface.

Forgetting operation and type names is in many ways the opposite of extension, but the effect of the hidden specification still affects what has been specified.

Given a powerful renaming mechanism, forgetting is a consequence of the institution properties for signature- and formula morphisms.

## 7.4 C++ with concepts as a Software Institution

Here we investigate the relationship between ConceptC++ and the institution notion. We find that there are may be many syntactic constructs in ConceptC++ that map to a given institution component, and that some constructs in ConceptC++

```
concept Ring1<typename T> {
  requires AbelianGroup<T>
  requires Monoid<T>;

  axiom Distributive(T a, T b, T c) {
    Monoid<T>::op(a, AbelianGroup<T>::op(b,c))
    == AbelianGroup<T>::op(Monoid<T>::op(a,b),
                           Monoid<T>::op(a,c));
    Monoid<T>::op(AbelianGroup<T>::op(b,c), a)
    == AbelianGroup<T>::op(Monoid<T>::op(b,a),
                           Monoid<T>::op(c,a));
  }
}

concept_map Ring1<int> {
}
```

Figure 7.7: Example of disjoint union between two concepts and the extension with extra axioms

have many interpretations as institution components. This was also evident in the different approaches taken to understand ConceptC++ as an institution in Haveraaen [2007] and Zalewski and Schupp [2007].

Here we deal with each institution component in turn, and discuss what ConceptC++ features that can be give us such a component.

### 7.4.1 Interface / Signature

The collection of declarations for an API form the interfaces of C++: type names, function and operator declarations, classes and their methods (we will call these type and operation declarations for simplicity). In ConceptC++ the concept itself, with its parameters, type and operations declarations, forms an API declaration and also gives us a named signature for later use. Let **sig** be a function that extracts the signature part of a C++ program, class or concept declaration. Signature extraction must follow the concept building operations, s.t., **sig**(`AbelianGroup<`**`typename`**` T>`) = **sig**(`Group<`**`typename`**` T>`). Figure 7.8 shows explicitly the signatures of Abelian Group and Ring1.

A concept declaration, or any collection of templated API, normally defines a large collection of signatures, one for all its possible instantiations. So a concept**`concept`**` A<`**`typename`**` T,` **`typename`**` U>{}` defines signatures for all combinations of instantiations of `T` and `U`, e.g., $I_1 =$ {**`int, float`**} from `A<`**`int,`**

**sig**(AbelianGroup<**typename** T>) =

```
{
  typename T;
  T op(T, T);
  T identity_elt();
  T inverse(T);
}
```

**sig**(Ring1<**typename** T>) =

```
{
  typename T;
  T AbelianGroup<T>::op(T, T);
  T Monoid<T>::op(T, T);
  T AbelianGroup<T>::identity_elt();
  T Monoid<T>::identity_elt();
  T inverse(T);
}
```

Figure 7.8: Example of signature extraction from concepts

**float**>, $I_2 = \{$**char, int**$\}$ from A<**char, int**> and $I_3 = \{$**int**$\}$ from A<**int, int**>. The former two, as formal signature declarations, should be considered isomorphic (two distinct type names, ignoring the fact that some type names are key words in C++), the latter is different as it only has one distinct type name.

The simplest signature morphism we have is the template instantiation mechanism, which substitutes the actual parameters for the formal parameters in the body of a template. This is very important in the reuse of concepts using extensions, and also in the mapping from the concept declaration to its instantiation.

We will also consider the definition of entities from an interface $I_1$ using expressions over an interface $I_2$ as a signature morphism. Expressions allowed for this purpose are limited to those that can be inlined into ConceptC++ axioms. This boils down to **typedef** for type declarations and non-recursive return expressions for operation declarations. We combine interface morphisms $i : I \to I'$ and $i' : I' \to I''$ to one signature morphism $(i; i') : I \to I''$ by inlining transformation that fully expands the expressions of $i'$ inside the expressions of $i$, so that no trace of entities from $I'$ remains in the result.

Identifying interface morphisms in plain program code is difficult, since there is no sharp distinction between what should be considered interface $I$ and interface $I'$ when trying to identify the signature morphism $i : I \to I'$. However, if the code is templated, this becomes possible, if it is templated with explicit

requirements this is straight forward.

Consider a group of constrained template definitions. Then the signature of the constraints constitute the signature $I'$. The types and operations defined constitute the signature $I$. The defining expressions give the signature morphism $i : I \to I'$, provided the necessary syntactic restrictions are met.

Concept maps are syntactically very explicit about requirements and target signature (that of the target concept). We will therefore provide an overloaded version of the **sig** function, one which extracts signature morphisms from concept maps. This extraction can also be applied to segments of program code provided the necessary source and target interfaces are identified. In all cases, the expressions being used are limited to the ones constituting signature morphisms.

The constrained partial concept map definition presented in figure 7.9 gives us a signature morphism from $\mathbf{sig}(A)$ to $\mathbf{sig}(B)$.

```
template <typename T>
requires B<T>
concept_map A<T, int> {
  T foo(int i) { return B<T>::bar; }
}
```

Figure 7.9: Example of morphism from $\mathbf{sig}(A)$ to $\mathbf{sig}(B)$

### 7.4.2 Formulas

The ConceptC++ formulas are the axioms embedded in concept declarations. Let **ax** extract these axioms. For a concept `A<T,U>`, then $\mathbf{ax}(\texttt{A<T,U>}) \subseteq F(\mathbf{sig}(\texttt{A<T,U>}))$ as required.

Given an axiom $\varphi \in F(I)$ and a signature morphism $i : I \to I'$, we get a corresponding $I'$ axiom by fully inlining the definitions of $i$ in $\varphi$, yielding an axiom $F\langle i \rangle(\varphi) \in F(I')$, as required.

Unfortunately ConceptC++ does not really provide notation for doing this transformation, see the discussion in Section 7.5.

### 7.4.3 Models

Classical C++ definitions are models (software libraries). A C++ program is a list of definition, which means that at each point of the list we have a model containing the list of types or functions previously defined. At each of these points, we can

define a concept map. It will capture the model of the current context and make sure it provides the required types and functions for the signature of the concept map. For example, if we introduce a line **concept_map** A<**int**>{}, inside our program, we capture the part of the model that the signature A<**int**> requires. The body of the concept map can as well define required definitions. Note that these definitions are not limited in the same way as signature morphisms are. Here any appropriate C++ code can be used.

Given a model $\ell' \in L(I')$ and a signature morphism $i : I \to I'$. We get a reduct $L\langle i^{\mathrm{op}}\rangle$ by fully inlining $\ell'$ inside the expressions of $i$. Then $i;\ell'$ is a model, defining each declaration of $I$.

### 7.4.4 Satisfaction relation

Given a model $\ell \in L(\Sigma)$ and an axiom $\varphi \in F(\Sigma)$, and think of the axiom as an assertion on its free variables (the variables of the axiom). We define that $\ell$ satisfies $\varphi$, if the assertion holds for all possible arguments for the free variables of the assertion.

Now, given a signature morphism $i : I \to I'$, an axiom $\varphi \in F(I)$ and a model $\ell' \in L(I')$. Then we must prove that $L\langle i^{\mathrm{op}}\rangle(\ell')$ holds for $\varphi$ if, and only if, $\ell'$ holds for $F\langle i\rangle(\varphi)$. By definition, $L\langle i^{\mathrm{op}}\rangle(\ell') = i;\ell'$ (fully inlined), and $F\langle i\rangle(\varphi)$ is fully inlining the definitions of $i$ in $\varphi$. But then obviously, $i;\ell'$ holds for $\varphi$ exactly when $\ell'$ holds for $F\langle i\rangle(\varphi)$.

## 7.5 Putting concepts together to build specifications

Although ConceptC++ has all the structure needed to become an institution, it lacks notation for the formula morphisms. Signature morphisms are needed to reach amalgamated union constructions.

Here we investigate and propose several strategies for getting the necessary concept morphisms:

- Amalgamation by replacing a declaration with its definition inside concepts
- Exploting the template parameter substitution mechanism to control amalgamation
- An explicit startegy for writing signature morphisms on concepts

Some of these are patterns on existing ConceptC++ constructs. Others will require extensions to the language.

### 7.5.1 Amalgamation by renaming inside signatures

Here the idea is to allow signature morphisms directly inside the concept declarations. This can control the granularity of renaming, s.t. we achieve the amalgamated union.

As an example, we want to build the specification of a ring using specifications of monoids and abelian groups.

We first specify monoids. We take the definition of `Monoid` from figure 7.3 and `AbelianGroup` from figure 7.6. We also introduce a separate specification of distributive.

```
concept Distributive<typename T, typename U> {
  T mul(U, T);
  T mul(T, U);
  T plus(T, T);

  axiom Distributive(U a, T b, T c) {
    mul(a, plus(b, c)) == plus(mul(a, b), mul(a, c));
    mul(plus(b, c), a) == plus(mul(b, a), mul(c, a));
  }
}
```

Figure 7.10: Distributive axioms on two typed multiplication

Now `Monoid` and `AbelianGroup` defines the same operations, but in our ring, the need to be dissociated. We need to rename them. The specification should like figure 7.11.

```
concept Ring2<typename T>:
  Distributive<T,T> {
  requires Monoid<T>;
  requires AbelianGroup<T>;
  rename T Monoid<T>::op(T a, T b) {
    return Distributive<T,T>::mul(a, b); }
  rename T AbelianGroup<T>::op(T a, T b) {
    return Distributive<T,T>::plus(a, b); }
}
```

Figure 7.11: Definition of a ring by renaming operations

Keyword `rename` would indicate that the operation has to be renamed, i.e. that the original operation is not available in the exported signature. Reusing

the keyword **inline** could be also consider, since it is already available in C++. But it has a different semantic. Unfortunately such syntax for concept definition is not available in ConceptC++.

### 7.5.2 Parameter substitution

We said in section 7.4.1 that parameter substitution gives us signature morphisms. The problem is that the parameters are only on sorts.

In the previous example we wanted to rename the multiplication and addition operations because both `AbelianGroup` and `Monoid` defined these operations as a same name `op`. With an union, we would get only one operation. But a ring needs two distinct operations. An abelian group specification could be defined not using a monoid, but this would not be a case a re-usability. Having a signature renaming sorts (types) is not enough.

There is a work around for this problem, which is widely used in C++, specially for template meta-programming technique. It is possible to define a type with a default constructor and an operation. This operation is usually the operator "parenthesis", but it could be any method or overloaded function. This way, an operation can be associated to a sort.

In figure 7.12, the different specifications are re-factored so that a ring can be composed using parameter substitution morphisms.

This seems to be encouraged style of writing concepts. But using function object can seem unnatural for the neophyte. Moreover, you cannot just provide a signature morphism if you discover your specification was not designed to be reused. But rather, you have to re-factor your concepts.

There is a way to know what part of the signature should appear in the parameters, is to ask the question: *What is not automatically deducible from other sorts?* But this question might not be always easy to answer. Moreover, we might want to split sorts whereas we can only merge them now.

The operation "parenthesis" in C++ is nice for keeping the same syntax of axioms. But we can use the same technique in more general languages. The only need is function overloading. Figure 7.13 defines `Semigroup` without using this operator.

The axioms are not natural, and we see the interest of using the operator "parenthesis" of C++.

### 7.5.3 Providing signature morphisms

Instead of re-factoring the specification, we now look for a way to write signature morphisms while keep intact the concept definitions of figures 7.1, 7.3, 7.5, and

7.6.

Distributivity on left and right of the monoid operation over the abelian group operation can be defined with the specification on figure 7.14

An union would have `op` from monoid and abelian group as the same operation. Moreover the two operations need to match the name for the distributivity specifications. We write first a signature morphism on monoid to rename `op` to `mul` like on figure 7.15.

Then we rename `op` to `plus` in the abelian group specification. This is depicted by figure 7.16

The wanted specification is then the union of our four specifications. Figure 7.17 specifies it by refinement.

We note that if we are given a ring model, they should as well behave like monoid, abelian group, etc.

**Axioms and signature morphisms**

Using signature such morphisms, a problem of readability of axioms rises.

A specification is a pair of signature and set of axioms. Then a model is said to satisfy a specification if all the axioms are satisfied. A concept should certainly be specification, as it defines a signature and a set of axioms. Unfortunately this is not exactly what we get.

The reason is that a signature morphism is provided by a concept map. But a concept map is a contract where the developer claims his model holds for a concept. And all the axioms are then reachable in the requirements. On a constrained partial concept map, if the constraints are met, then we claim axioms hold for the source of the signature morphism.

This is unfortunately unnatural. For example, figure 7.18 defines a signature morphism from a semigroup signature to a monoid signature.

As you see, there is no indication on the concept `Monoid` that it should hold for associativity. But this axiom is inherited since we claimed our `Semigroup` holds for any `Monoid`. It means the virtual set of axioms a concept has may grow without changing the definition of the concept or its requirements or refinements.

It should be expected to insert the signature morphism from `Semigroup` to `Monoid` on the definition of the concept of `Monoid`. A way to do this is naturally the refinement. Even though refinement works in such a simple case, it is not powerful enough to provide morphism that for example rename functions. If concept map definitions were allowed inside concept definitions, we would be able to write more complex signature morphisms, as powerful as the constrained partial concept map technique, with the difference that the signature morphism would be seen from the concept definition. And then the set of all axioms of

the specification could be deduced by looking at the concept definition. Such a concept definition on `Monoid` would look like figure 7.19.

Unfortunately,this kind of thing cannot be written in ConceptC++. The interesting thing is that this way to define signature morphisms is nearly identical to the "natural" way we wanted to write our specification in section 7.5.1.

#### Resolving ambiguities

The method of using user defined signature morphisms has a problem. There is an ambiguity as a ring can be seen as a monoid in two different ways. The problem is that name a model is identified with its type parameter. But there are here two different models with the same types. To make a difference at least one type as to be wrapped. Figure 7.20 defines a wrapper.

`Mul<T>` is not equivalent to a `T`, but has a different name. We can now rewrite the signature morphism from `Monoid<Mul<T> >` to `MonoidMul` as figure 7.21 does.

There is another way to address the problem of ambiguity. We introduced wrapped types to make differences between model. This can be pushed further. It is possible name all our morphisms and models. The type parameters are used to name models. An unique dummy type can be introduced to name each of our unique models. On figure 7.22, the concept definitions are written so that we can name our models.

With a class template signature morphisms can be named. This look like the wrapping technique. Figure 7.23 identifies the signature morphism between *Semigroup* and *Monoid* with class template `Phi<ModelName>`.

To create a model, a dummy type as to be created like shown on figure 7.24.

`Phi<MIntPlus>` will now be a model for semigroup. This model can be aliased with a **`typedef`**:

```
typedef Phi<MIntPlus> SGIntPlus;
```

The main problem of this strategy is that it might seem unnatural to create dummy types for naming our models and morphisms. The second problem is that, like with our third strategy, signature morphisms using concept maps make specifications inherit automatically from axioms from other concepts.

## 7.6 Discussion and Conclusion

Institutions are a stamp of good quality for relating specifications (concepts) and models (C++ code), Signature morphisms are a crucial component. Having an institutions gives us many useful ways of building modular specifications. These

are well known in algebraic specification languages. Extended ML [Sannella and Tarlecki, 1986, 1991], is an example of extension of a (nearly) main stream programming language, Standard ML, with specification constructions. The interesting aspect is that signature morphisms are described as parametrized functors between models as we do in this paper. The "parametrized" corresponds to the template, the functor corresponds to the concept map definition with a requirement on another concept.

In this paper, we have seen three different ways of building colimits (amalgamated unions) of specifications.

A direct approach was to allow individual signature morphisms directly in the concept declaration, e.g., by an interpretation of the keyword `rename` or **`inline`**. This would seem natural for normal programmers and people used to algebraic specifications, but would require extensions to the proposed ConceptC++ standard.

Another approach is to use the template mechanism to also rename functions and not only types. This is achieved through the function object pattern in C++, where special classes are equipped with the "paranthesis" operator to allow object instances to behave as functions. This technique seems natural for a trained C++ developer used with bizarre techniques of template meta programming, but is not usual for normal people used to algebraic specifications. Moreover, this method may require substantial code refactoring. It seems to be the adopted by the programmers who initially investiageted the ConceptC++ possibilities. For example MTL4 uses this approach.

We also investigated several strategies for explicitly writing and managing signature morphisms. The nice side is that you might never need refactoring your code as you are free to modify the behavior of your specifications each time you have to re-use them. Unfortunately, it can also change the semantic of your specifications. The implication here is that it is impossible to prove correctness of a program incrementally. Only after the full program text is known, will it be possible to reason about its components.

This problem could be allivated if we integrated the associated signature morphisms, in the form of concept maps, directly in the concept declarations. Again this seems to be a very natural way of building amalgamated unions for the flexible reuse of specifciations.

We also noticed that a standard type parameter list causes problem of ambiguity. This can be resolve by systematically having an additional type parameter for uniquely identifying each instance of a concept. But this approach is not natural as it induces extra notational overhead. An alternative approach where the whole parameter list was reduced to the identifying type turned out to be a

much more natural approach to writing specifications.

Summing up: modular reuse of concepts when building large concept specifications is very important. In the current proposal or ConceptC++, this can be done either relying on functon objects, probably inducing heavy refactoring of code, or using a signature morphism metodology that breaks the modularity of corretness proofs. As an alernaive wehave identified two syntactic extensions which both seem to work naturally with the ConceptC++ notation and semantics, and which both seem natural for the user.

```
concept Semigroup<typename T, typename Op> {
  T operator()(Op, T, T);

  //axioms...
}

concept Monoid<typename T, typename Op>:
  Semigroup<T, Op> {
  T identity_elt();

  //axioms...
}

concept Abelian<typename T, typename Op> {
  T operator()(Op, T, T);

  //axioms...
}

concept Group<typename T, typename Op>:
  Monoid<T, Op> {
  T inverse(T);

  //axioms...
}

concept AbelianGroup<typename T, typename Op>:
  Group<T, Op>, Abelian<T, Op> {}

concept Ring3<typename T,
              typename Plus, typename Mul>:
  Monoid<T, Mul>, AbelianGroup<T, Plus> {
  axiom Distributivity(Mul mop, Plus pop,
                       T a, T b, T c) {
    mop(a,pop(b,c)) == pop(mop(a,b),mop(a,c));
    mop(pop(b,c),a) == pop(mop(b,a),mop(c,a));
  }
}
```

Figure 7.12: Construction of a ring using parameter substitution. Some axioms have been omitted for conciseness.

```
concept Semigroup<typename T, typename Op> {
  T op(Op, T, T);

  axiom Associativity(Op o, T a, T b, T c) {
    op(o, a, op(o, b, c)) == op(o, op(o, a, b), c);
  }
}
```

Figure 7.13: Example of semigroup specification re-factored without using operator “parenthesis”

```
concept Distributive<typename T> {
  T mul(T, T);
  T plus(T, T);

  axiom Distributivity(T a, T b, T c) {
    mul(a, plus(b, c))
      == plus(mul(a, b), mul(a, c));
    mul(plus(b, c), a)
      == plus(mul(b, a), mul(c, a));
  }
}
```

Figure 7.14: Concept definition for distributivity

```
concept MonoidMul<typename T> {
  T mul(T, T);
  T one();
}

template <typename T>
requires MonoidMul<T>
concept_map Monoid<T> {
  T op(T a, T b) { return MonoidMul<T>::mul(a, b); }
  T identity_elt() { return MonoidMul<T>::one(); }
}
```

Figure 7.15: Signature morphism renaming the operations of *Monoid*

```
concept AbelianGroupPlus<typename T> {
  T plus(T, T);
  T inverse(T);
  T zero();
}

template <typename T>
requires AbelianGroupPlus<T>
concept_map AbelianGroup<T> {
  T op(T a, T b) {
    return AbelianGroupPlus<T>::plus(a, b);
  }
  T inverse(T a) {
    return AbelianGroupPlus<T>::inverse(a);
  }
  T identity_elt() {
    return AbelianGroupPlus<T>::zero();
  }
}
```

Figure 7.16: Signature morphism renaming the operations of *AbelianGroup*

```
concept Ring4<typename T>
  : MonoidMul<T>, AbelianGroupPlus<T>,
    Distributive<T> {}
```

Figure 7.17: Construction of a ring using renamed specifications

```
concept Semigroup<typename T> {
  T op(T, T);

  axiom Associativity(T a, T b, T c) {
    op(a, op(b, c)) == op(op(a, b), c);
  }
}

concept Monoid<typename T> {
  T identity_elt();

  axiom Identity(T a) {
    op(a, identity_elt()) == a;
    op(identity_elt(), a) == a;
  }
}

template <typename T>
requires Monoid<T>
concept_map Semigroup<T> {
  T op(T a, T b) { return Monoid<T>::op(a, b); }
}
```

Figure 7.18: Defining a signature morphism makes the source inherits from axioms

```
concept Monoid<typename T> {
  T identity_elt();

  axiom Identity(T a) {
    op(a, identity_elt()) == a;
    op(identity_elt(), a) == a;
  }

  concept_map Semigroup<T> {
    T op(T a, T b) { return Monoid<T>::op(a, b); }
  }
}
```

Figure 7.19: A concept map definition in a concept would imply a signature morphism

```
template <typename T>
requires std::CopyConstructible<T>
struct Mul {
  private:
    T value;
  public:
    Mul(T v): value(v);
    operator T() const { return value; }
};
```

Figure 7.20: Wrapping a type

```
template <typename T>
requires MonoidMul<T>
concept_map Monoid<Mul<T> > {
  T op(T a, T b) {
    return T(MonoidMul<Mul<T> >::mul(a, b));
  }
  T identity_elt() {
    return T(MonoidMul<Mul<T> >::one());
  }
}
```

Figure 7.21: Disambiguating the signature morphism from *Monoid* to *MonidMul*

```
concept Semigroup<typename ModelName> {
  typename T;
  T op(T, T);

  axiom Associativity(T a, T b, T c) {
    op(op(a, b), c) == op(a, op(b, c));
  }
}

concept Monoid<typename ModelName> {
  typename T;
  T op(T, T);
  T identity_elt();

  axiom Identity(T a) {
    op(a, identity_elt()) == a;
    op(identity_elt(), a) == a;
  }
}
```

Figure 7.22: Naming models

```
template <typename ModelName>
struct Phi;

template <typename ModelName>
requires Monoid<ModelName>
concept_map Semigroup<Phi<ModelName> > {
  typedef Monoid<Name>::T T;
  T op(T a, T b) {
   return Monoid<Name>::op(a, b);
  }
}
```

Figure 7.23: Naming a signature morphism

```
struct MIntPlus;

concept_map Monoid<MIntPlus> {
  typedef int T;
  int op(int a, int b) { return a+b; }
  int identity_elt() { return 0; }
}
```

Figure 7.24: When defining a model, a dummy type is created

# Chapter 8

# Reusing concepts for testing

The concept extension of C++ contains a syntax for axioms. Those axioms could be reused for several application. One of those is testing.

Testing using algebraic specification is well-known. The interesting aspect with ConceptC++ is that it is intended to be a main-stream extension. In chapter 6, we have shown a way to de-sugar ConceptC++ into C++. In the following paper, we show how we extract concept definitions to generate test oracles and test drivers.

This transformation does not transform the source but rather generates another product from the source code which is a testing program. The generated program is still ConceptC++ but can be de-sugared using the transformation presented in chapter 6.

This paper will be presented at GPCE'09.

# Testing with Concepts and Axioms in C++

ANYA HELENE BAGGE VALENTIN DAVID MAGNE HAVERAAEN

*In the proceedings of Eighth International Conference on Generative Programming and Component Engineering (GPCE'09), October 4-5, 2009, Denver, Colorado, USA.*

**Abstract.** Modern development practices encourage extensive testing of code while it is still under development, using unit tests to check individual code units in isolation. Such tests are typically case-based, checking a likely error scenario or an error that has previously been identified and fixed. Coming up with good test cases is challenging, and focusing on individual tests can distract from creating tests that cover the full functionality.

Axioms, known from program specification, allow for an alternative way of generating test cases, where the intended functionality is described as rules or equations that can be checked automatically. Axioms are proposed as part of the *concept* feature of the upcoming C++0x standard.

In this paper, we describe how tests may be generated automatically from axioms in C++ concepts, and supplied with appropriate test data to form effective automated unit tests.

## 8.1 Introduction

Modern software engineering practises encourage the use of unit testing to increase software reliability. Test-driven development (TDD) [Beck, 2002] dictates that software should be extended by writing tests for a new feature *first*, before implementing the feature. The tests provide a specification of the behaviour of the new feature, and provide an easy way to check the implementation throughout development and refactoring.

**© 2009 Association for Computing Machinery, Inc. Reprinted by permission from the proceedings of the 8th international Conference on Generative Programming and Component Engineering, Testing with Concepts and Axioms in C++, Anya Helene Bagge, Valentin David, and Magne Haveraaen**

Less extreme methods call for tests for all program units, and for regression tests to be written to ward off the reappearance of known bugs. Such methods may be practised rigorously, or in an ad hoc manner. Common to all is that they rely on the programmer to invent good test cases that cover both likely, unlikely and even 'impossible' errors. The programmer must also be careful that the tests exercise the full expected feature set, or the implementation will seem OK when all it does is implement the bare minimum to pass the tests (as is common in TDD, where the tests are the actual specification of expected behaviour).

### 8.1.1 Axiom-Based Testing

We suggest writing tests based on *axioms* that formally specify expected behaviour, rather than relying on ad hoc test cases. Axiom-based testing was introduced in the early eighties in the DAISTS [Gannon et al., 1981] system, which used formal algebraic specifications as a basis for unit testing. In DAISTS, a test consists of axioms in the form of conditional equations, which serve as a test oracle, an implementation with an equality operator; and a set of test data. A simple coverage analysis is done of the test runs to ensure that all the axioms and program code are exercised by the tests.

ASTOOT [Doong and Frankl, 1994] applied the ideas of axiom-based testing to object-orientation, with automated testing for Eiffel. Axioms were specified in an OO-like style, rather than the functional notation used in DAISTS.

The Daistish system [Hughes and Stotts, 1996] brought these ideas to C++. Unlike ASTOOT, Daistish used a functional notation for axioms, giving a notational gap between the conditional equational form of the axioms and the methods of C++.

Traditional unit testing, as popularised by agile methods in the last decades, is practically oriented, and does not rely on formal methods. Mainstream software engineers have focused on development methods like TDD and extreme programming [Beck, 1998], while much formal methods research has focused on formal specification and verification – which have been difficult to apply to mainstream languages and mainstream development.

Research on axiom-based testing has continued, however, and axioms are have been introduced as part of the new *concept* proposal for the upcoming C++ standard [Becker, 2009, Gregor et al., 2006a] – giving a mainstream language built-in syntactic support for axioms. The CASCAT system [Yu et al., 2008] provides a tool for testing Java components based on algebraic specification. Axiom-based testing has been employed in the Sophus numerical C++ library [Haveraaen and Brkic, 2005], and also in the JAX [Stotts et al., 2002] (Java Axioms) testing approach and JAxT [Haveraaen and Kalleberg, 2008] tool.

Axiom-like features have also been added to recent versions of JUnit [Saff, 2007]. Gaudel and Le Gall [Gaudel and Gall, 2008] provide a survey of the use of algebraic specification in testing.

Axiom-based testing from concepts has two main parts that instrument the implementation being tested:

- axioms, in the form of conditional equations, and
- suitable test data points.

Running an axiom-based test consists of evaluating the condition and (if it succeeds) the two sides of the equation using the test data, and comparing the results, typically with the help of the equality operator. For example, to test the following commutativity axiom $x + y = y + x$, we may substitute $4$ for $x$ and $5$ for $y$, evaluate $4 + 5$ and $5 + 4$, and then verify that $9 = 9$. A good test data set for this case would also include negative numbers and zero.

If the results are to be reliable, the axiom must correctly express the desired feature. Earlier it was considered crucial that the code for the equality operator had to be correct [Gaudel, 1995]. We have discussed this previously [Haveraaen and Brkic, 2005], concluding that with testing the equivalence and congruence properties of the equality operator, it can be treated alongside any other function being tested. Another problem appears if the equality operator used in a concept axiom is not implemented. This is known as the oracle problem, and can be handled by techniques based on behavioural equivalence [Gaudel, 1995, Chen et al., 1998], i.e., two values are considered equal if they cannot be distinguished by any operation in the system. The ASTOOT [Doong and Frankl, 1994] system is based on behavioural equivalence, though in practise the user must still define equivalence, either through an axiom, or by an equals operator in the implementation. Chen et al. [Chen et al., 1998] describe a system for testing object-oriented programs, and provide a technique for determining behavioural equivalence based on white-box heuristics. In this paper, though, we will assume that an equality operator has been implemented for every type that occurs in the left- or right-hand side of an axiom.

Concepts and axioms are still a work in progress as far as C++ standardisation is concerned[1]. Previous work on C++ axioms has mainly focused on their use for optimisation [Bagge and Haveraaen, 2008, Tang and Järvi, 2007b]. Our contributions in this article include:

- a technique for using C++ axioms for testing, and

[1] And, in fact, have recently been dropped from the final proposal.

- a tool to support this technique.

The rest of the paper is organised as follows. In the next two sections we introduce C++ concepts and axioms, and show how to generate test oracles and test code from them. In Section 8.4, we discuss how to generate test data, both random and user-selected. We finish off with a discussion and conclusion in Sections 8.5 and 8.6.

## 8.2 Concepts

*Concepts* [Gregor et al., 2006a, Becker, 2009] allow restrictions to be placed on template arguments. A concept describes a specification for types. It lists the members (functions, associated types, etc.) that are required for some types to model the concept, and the axioms that apply to those members. For example, the following *Monoid*[2] concept requires the existence of an `identity_element` and an operator, and gives an *Identity* axiom (adapted from [Becker, 2009]):

```
concept Monoid<typename T>
    : Semigroup<T> {
  T op(T, T);
  T identity_element();
  axiom Identity(T x) {
    op(x, identity_element()) == x;
    op(identity_element(), x) == x;
} }
```

Axioms are simple conditional equations (or inequalities), universally quantified over the axiom parameters. Multiple equations may be given inside an axiom – they are combined by logical *and*. More complicated axioms, e.g., with existential quantification, cannot be expressed directly. The sides of the equations are full C++ expressions, allowing use of things like the comma operator and calls to any accessible function.

To state that a set of types model a concept, we use a *concept map*. The concept map can specify a mapping between the implementation names (from the class) and the names used in the concept, and can also be used to add extra code necessary to model the concept. Any functions not mentioned explicitly in the concept map is taken from the context – in many cases the body of a concept map is quite short, or empty. In the concept map below, we state that the `FiniteInt` class of bounded integers models the Monoid concept, and give an operator that returns the addition of elements and an `identity_element` function that returns the `FiniteInt::zero` identity element.

[2] A *monoid* is an algebraic class with an operator $\oplus$ and an identity element $e$, such that $x \oplus e = e \oplus x = x$. For example, $\langle \texttt{int}, +, 0 \rangle$ and $\langle \texttt{int}, *, 1 \rangle$ are monoids.

```
template<int size>
concept_map Monoid<FiniteInt<size> >{
  FiniteInt<size> op(const FiniteInt<size>& a,
                     const FiniteInt<size>& b) {
    return a+b;
  }
  FiniteInt<size> identity_element() {
    return FiniteInt<size>(0);
} }
```

Without the concept map body, we would have to provide `op` and `identity_element` for `FiniteInt`s directly.

Concepts may also be declared `auto`, in which case an implicit concept map is provided for any set of types that have the relevant functions declared. We feel it is best to avoid axioms in auto concepts – since they may end up specifying behaviour for functions without the programmer being aware of it (though, a few standard cases like having equality, comparison or assignment operators can probably safely be made auto). We will therefore only generate tests for the cases where the programmer has explicitly used a concept map to declare that the implementation models a concept.

## 8.3 From Axioms to Test Code

There are two steps involved in generating tests from concepts. First, we generate a *test oracle* for each axiom in each concept. The test oracle is a function having the same parameters as an axiom, and returning true or false depending on whether the axiom holds for the given arguments.

For example, consider the *Indexable* concept in Figure 8.1, intended for data structures such as arrays. It has the usual indexing operators you would expect, and an axiom *ArrayEqual*. The axiom can be transformed into callable code by creating a normal C++ template class for the concept (`Indexable_oracle`), and making the axiom a boolean function within that class – see Figure 8.2.

The second step is to generate test cases for each type that models a concept. This is done by finding all the concept maps within the program, and generating code for each of them. The test case will use data iterators (see Section 8.4) to iterate through a set of data values for each argument to the axioms, and then call the test oracle for each combination of data values. Success or failure of the oracle test is then reported to the testing framework.

For example, consider an `ArrayFI` class – an array indexed by finite (bounded) integers. A simplified version of the class is shown in Figure 8.4. It is supplied

```
concept Indexable<typename A, typename I,
                  typename E>
  : std::EqualityComparable<A,A>,
    std::EqualityComparable<E,E> {
  requires SameShape<A, I>;
  const E& operator[](const A&, const I&);
  E& operator[](A&, const I&);

  axiom ArrayEqual(A a, A b, I i) {
    if (a == b)
      a[i] == b[i];
  }
}
```

Figure 8.1: The concept *Indexable* has indexing operators and an axiom *ArrayEqual* that states that two if Indexables are equal, then their elements are equal. `A` is an indexable type, `I` is the index type, and `E` is the element type. `A` and `I` are required to be of the same shape, i.e., the values of type `I` are the allowable indices for the type `A`. *SameShape* is a trivial concept used to state that the indexable and index type are of compatible shapes/dimensions.

```
template <typename A,typename I,typename E>
requires Indexable<A, I, E>
struct Indexable_oracle
{
  static bool ArrayEqual(A a, A b, I i)
  {
    if (a == b)
      if (!(a[i] == b[i]))
        return false;
    return true;
  }
};
```

Figure 8.2: Oracle code from the ArrayEqual axiom. The oracle returns immediatly upon failure, otherwise we continue, as there may be more than one equation in the axiom.

```
template <int size,typename E>
requires Indexable<ArrayFI<size, E>, FiniteInt<size>, E>
struct Indexable_testCase<ArrayFI<size, E>, FiniteInt<size>, E>
{
  static void ArrayEqual() {
    typedef HasDataSet<ArrayFI<size, E>>::dataset_type dt_0;
    dt_0 b_0 = HasDataSet<ArrayFI<size, E>>::get_dataset();
    for (DataSet<dt_0>::iterator_type a_0 = DataSet<dt_0>:: begin(b_0)
       ; a_0 != DataSet<dt_0>::end(b_0); ++a_0) {
      typedef HasDataSet<ArrayFI<size, E>>::dataset_type dt_1;
      dt_1 d_0 = HasDataSet<ArrayFI<size, E>>::get_dataset();
      for (DataSet<dt_1>::iterator_type c_0 = DataSet<dt_1>:: begin(d_0)
         ; c_0 != DataSet<dt1>::end(d_0); ++c_0) {
        typedef HasDataSet<FiniteInt<size>>::dataset_type dt_2;
        dt_2 f_0 = HasDataSet<FiniteInt<size>>::get_dataset();
        for (DataSet<dt_2>::iterator_type e_0 = DataSet<dt_2>:: begin(f_0)
           ; e_0 != DataSet<dt_2>::end(f_0); ++e_0)
        check(Indexable_oracle<ArrayFI<size, E>,
                          FiniteInt<size>, E>::ArrayEqual(*a_0,*c_0, *e_0),
              "Indexable", "ArrayEqual");
} } } };
```

Figure 8.3: Concrete test code generated from a concept map. *HasDataSet* is used to select an appropriate data set for each data type. `check` is a hook for reporting results to a testing framework.

with two concept maps, relating the implementation to the *SameShape* and *Indexable* concepts. The first stating that any `ArrayFI` of size `size` has the same shape as a `FiniteInt` of size `size` – this is needed to fulfil the *SameShape* requirement of the *Indexable* concept. The second states that `ArrayFI` is Indexable with index type `FiniteInt` and element type `E`. Note that the concept maps are templated, working on any integer `size` and element type `E`.

The test case (seen in Figure 8.3) consists of an `Indexable_testCase` class specialised for `ArrayFI<size, E>`, `FiniteInt<size>` and `E`. The class contains a test function, `ArrayEqual`, which iterates over the data generators and calls the generic test oracle derived from the axiom. The two outer loops generate arrays (`*a_0` and `*c_0`), while the inner loop generates indexes (`*d_0`). The test oracle (from Figure 8.2) will check that the array code behaves as expected for an Indexable structure. The `HasDataSet` provides a mapping from a type to a data generator for that type (reasonable defaults for this are generated automatically – see Section 8.4).

```
template<int size, typename E>
class ArrayFI {
private:
  E data[size];
public:
  E& operator[](const FiniteInt<size>& i) {
    return data[i];
  }
  bool operator==(const ArrayFI& a){
    for(int i = 0; i<size; ++i)
      if (data[i] != a.data[i])
        return false;
    return true;
  }
  int getSize() const {
    return size;
  }
};

// Any ArrayFI has the same shape as
// a FiniteInt index type if the sizes match
template<int size, typename E>
concept_map SameShape<ArrayFI<size, E>,
                      FiniteInt<size> > { }

// ArrayFI<size,E> is Indexable, with index
// type FiniteInt<size> and element type E
template<int size, typename E>
concept_map Indexable<ArrayFI<size, E>,
                      FiniteInt<size>, E> { }
```

Figure 8.4: The *ArrayFI* class, parameterised with a size and an element type.

### 8.3.1 Reusable Tests

A convenient effect of having concepts and their axioms separate from the classes that implement them is that they can be freely reused for testing new types that model the same concepts. If you already have a *Stack* concept with carefully selected axioms, you get the tests for free when you implement a new stack class.

Having libraries of standard concepts for things such as algebraic classes [Gottschling, 2006] (including *monoid*, *ring*, *group* and others that apply to numeric data types), containers (indexable, searchable, sorted, ...) as well as common type behaviors [Gregor and Lumsdaine, 2008] (*CopyAssignable*, *EqualityComparable*, ...) cuts down on the work needed to implement tests. A well thought-out library is also far less likely to have flawed or too-weak axioms than axioms or tests written by a programmer in the middle of a busy project.

### 8.3.2 Concept Combinations

Some combinations of classes can create interesting interactions between concepts. For example, the `FiniteInt` type we used in the implementation of `ArrayFI` satisfies the *Monoid* concept from Section 8.2 (as well as several other algebraic concepts that are too lengthy to include in this paper). If we extend our `ArrayFI` with element-wise operations, an instance `ArrayFI<FiniteInt>` can 'inherit' the *Monoid* concept from the `FiniteInt`. For this to work, we need to provide a concept map

```
template<typename A>
  requires DefaultIndexable<A>,
           Monoid<DefaultIndexable<A>
                  ::element_type>,
           std::CopyConstructible<A>
concept_map Monoid<A> {
  A op(const A& a, const A& b) {
    return Shape<A>::map(
       Monoid<DefaultIndexable<A>
              ::element_type>::op,
       a, b);
  }
  A identity_element() {
    return Shape<A>::build(
       Monoid<DefaultIndexable<A>
              ::element_type>
       ::identity_element());
  }
}
```

The *Indexable* concept may be used in several different ways on the same array type with different index and element types. As we want the compilation process to automatically deduce which way to index our data structure, we need to provide a default pair of index type and element type to each *Indexable* through the following concept *DefaultIndexable*:

```
concept DefaultIndexable<typename A> {
  typename index_type;
  typename element_type;
  requires Indexable<A, index_type,
                     element_type>;
}
```

Then, for example, `ArrayFI` would only need a small concept map like the one below to inherit all the axioms.

```
template <int size, typename E>
concept_map DefaultIndexable<
               ArrayFI<size, E>> {
  typedef FiniteInt<size>
    index_type;
  typedef E element_type;
}
```

Based on the above concepts and the concept maps, an `ArrayFI<size, FiniteInt>` would have test code for the ArrayEqual axiom (instantiated from the template code in Figure 8.3), and for the `Monoid::Identity` axiom. And, as `ArrayFI<size, FiniteInt>` is itself a *Monoid*, we can use it as the element type for a new *Indexable Monoid* `ArrayFI<size1, ArrayFI<size2, FiniteInt> >`, and so on[3]. Such constructions are important in some problem domains [Haveraaen et al., 2005] and allow us to do some simple integration testing with axioms as well.

### 8.3.3 Test Drivers / Suites

So far we have generated test oracles from axioms, and test cases that generate test data and call the oracles. To actually perform the testing, we need to call the test cases as well. There are three ways to do this: we may call the code manually, we may generate code that calls all known test functions, or we may use a combined approach.

[3]Which raises the question – how many of these do we generate code for? None and all. Since the generated test code uses templates, the basic test case functions also handle the nested combinations. Only the basic variants are called automatically by our tool's code, though.

By default, our tool will generate a `main` function filled with calls to all non-template test functions. Guessing at sensible template parameters is difficult in the case of unconstrained template parameters and when there is a large or infinite number of choices (as in the case of the nested arrays above). We therefore rely on the user to choose which templated tests to run, as explained in Section 8.4.

If we want fully automatic test program generation, we could analyse existing application code and find suitable template instantiation arguments there. Or, in cases where template parameters are constrained by concepts, we could generate calls with all classes that fulfil the concept constraint (with a cut-off in place to avoid infinite nesting). This would allow quite exhaustive exercising of code, including combinations that a programmer would likely never think of.

Even if the tool does not automatically generate full test suites, it could help the programmer by generating code templates. With integration into an IDE – such as Eclipse – test suite building can be done in a guided manner.

### 8.3.4 Axioms for Object-Oriented Code

The axiom examples we have used so far have mostly been in a functional style where results are returned and there are no side-effects on arguments. Realistic C++ code will often be written in a more object-oriented style.

Object-oriented style favors side-effects on the first argument. To capture side-effects in concepts, some functions will have to have reference type arguments. If the first argument is a reference, then the function can be defined in the concept map as a method defined in a class. If the first argument is not a reference, or it is a `const` reference, then the function is defined as `const` method. Non-`const` methods – methods that may change the object – is the norm in C++ programming, which means that function declarations in concepts will often have reference parameters. The side effect of those functions have to be captured somehow by the axiom. The comma operator can be useful for testing side effects. An example is the following axiom:

```
axiom CopyPreservation(T x, U y) {
  (x = y, x) == y;
}
```

This axiom states that after assigning `y` to `x`, the value of `x` should be equal to `y`. The comma operator has the effect of first assigning `y` to `x`, and then yielding the value of `x`.

Figure 8.5 shows the traditional bounded stack example also used for DAISTS [Gannon et al., 1981], Daistish [Hughes and Stotts, 1996] and JAX [Stotts et al., 2002]. The *BoundedStack* concept is written in a functional syntactic style,

```
concept BoundedStack<typename S> {
  requires std::DefaultConstructible<S>,
    std::EqualityComparable<S>;
  std::EqualityComparable E;
  E top(S);
  E pop(S&);
  void push(S&, E);
  bool full(S);
  bool empty(S);

  axiom PushTop(S s, E e) {
    if(!full(s))
      (push(s,e), top(s)) == e; }
  axiom PushPop(S s, E e) {
    if(!full(s))
      (push(s,e), pop(s)) == e;
    if(full(s))
      (push(s,e), pop(s), s) == s; }
  axiom Empty1() {
    empty(S()) == true; }
  axiom Empty2(S s, E e) {
   if(!full(s))
    (push(s, e), empty(s)) == false; }
  axiom Equal1() {
    S() == S(); }
  axiom Equal2(S s, E e) {
   if(!full(s))
    (push(s, e), s) != S();  }
  axiom Equal3(S s1, S s2) {
   if(!empty(s1) && (s1 == s2))
      s1.top() == s2.top();
   if(!empty(s1) && (s1 == s2))
      (pop(s1), s1) == (pop(s2), s2);
  }
}
```

Figure 8.5: An example of a bounded stack concept capturing side effect of OO programming style, with a selection of axioms. The comma operator (`,`) is used to first evaluate a call for the side effect (left side), then choosing the value we're interested in (right side). `S()` constructs a new stack.

```
...
bool PushFull_help(S s, E e) {
  try { push(s, e); return false; }
  catch(...) { return true; }
}
axiom PushFull(S s, E e) {
  if(full(s))
    PushFull_help(s, e) == true;
}
```

Figure 8.6: Checking for exceptions. The function `PushFull_help` will return true if pushing onto a full stack throws an exception, and false otherwise.

but the reference of the first parameter on `pop` and `push` captures the object-oriented style. These two stack operations modify the current object, rather than return a new modified stack.

In our stack axioms, we have intentionally not specified what happens when we attempt to push onto a full stack or pop an empty stack. In a traditional bounded stack, pushing onto a full stack has no effect. By leaving this behaviour undefined, we leave the door open for alternative solutions (handled by non-axiom test cases, for example).

However, if we wish to specify that an exception should be thrown when attempting to push onto a full stack, we would need a small helper function to do the push, catch the exception and return true or false – see Figure 8.6. With some small changes [Bagge and Haveraaen, 2008] to the proposed C++ syntax, we could avoid the use of the helper function.

This state-modifying style of axioms has some consequences for test code generation, since the test oracles will modify the test data. For this reason, the test oracles avoid reference arguments, ensuring that the data is copied into the oracle function. This may not be sufficient for all data structures, though. We are still unsure of the best way to handle this, as we would like to keep data generation as simple and efficient as possible. Fortunately the `const`/non-`const` status of parameters will give a clue as to when this may be a problem – for example, the equals operator is safe, since *EqualityComparable* specifies that it has `const` arguments. We could then try to force copying of test data which is passed as non-`const` arguments in axioms. Alternatively, one could simply expect the test driver to generate fresh data for every oracle invocation.

## 8.4 Generating Test Data

Creating a test oracle from the concept axioms and a concept map is straightforward, as described in the previous section. Such a test oracle will normally have parameter variables (free variables) that need to be instantiated by suitable values in order to actually perform testing.

We have three cases to consider when we want to provide data for a free variable:

- The parameter has a known, primitive C++ type.
- The parameter has a known, user-defined type. In this exposition we will not investigate the issues arising if the known type can be subclassed.
- The parameter type is a template argument to the test oracle. In this case, the template may have additional constraints, e.g., that a parameter models a given concept, see Section 8.3.3.

For the last case we will rely on concept maps to identify candidate types. Though some authors [Claessen and Hughes, 2000] claim that fixing the test data set for one such candidate will be sufficient, we believe test data sets should exercise several of these in order to check that the stated requirements are sufficient constraints on the template arguments.

We provide test data through associating test data generators with each class. For the primitive types, we can use a random generator library, to obtain an arbitrarily large selection of test data. User defined classes should provide a test data generation interface, allowing our testing tool to feed generated test data to the test oracles. A test data generator for a class template may call upon the data generators for the argument classes.

For a known type, whether primitive or user-defined, we see several strategies for providing test data.

1. User selected data sets.
2. Randomly chosen generator terms.
3. Randomly chosen data structure values.

The first is the classical approach to testing and the one (implicitly) favored by test driven development. Here the tester decides, e.g., that integer values -1, 0, 1 and 3 are of prime importance, or that stacks `S()`, `S().push(1)` and `S().push(1).pop()` are specifically important. Such selected data sets are useful for regression testing, where specific data sets that have exposed problems

in the past are rechecked with each revision of the code. The data sets can also be targeted for other purposes, e.g., path coverage of the implemented algorithms.

The second is favored by Claessen & Hughes in their QuickCheck system [Claessen and Hughes, 2000] and by Prasetya et al. for their Java-based testing system [Prasetya et al., 2007]. The idea is to let random expressions or sequences of (public) methods compute data values of the appropriate type. By choosing a suitable enumeration of terms this will always be possible and give good data coverage. For example, testing integer-like types (with axioms such as associativity, commutativity, distributivity) we may use expressions $0, 0+1, (-1+0)*2, \ldots$ and for stacks sequences like `S s(); s.push(1+-2); s.push(3|4); s.pop();` may be used.

The third approach requires the tester to have access to the data representation (data field attributes) of a type. For primitive types such as floats, this means setting the bit patterns of a floating point number directly. For a user-defined class this implies that each data field is given a random value of the appropriate type, subject to the constraints of the implementation. For instance, having a rational number class where we represent rational numbers as pairs of integers (a nominator and a denominator, the denominator different from zero), we may choose random pairs of integers for the attributes, discarding any pair where the denominator part would be equal to 0. Such direct setting of attribute values may give access to a larger range of test values than allowed by method 2, and is needed if all or some of the data fields are publicly available. Setting data attributes directly requires a filtering mechanism that identifies all bad data combinations, i.e., a complete data (class) invariant. If the data invariant has narrow requirements on the data, e.g., that the stack has a length field required to be equal to the length of the linked list representing the data on the stack, independently generating random integers and random linked lists will probably turn up too few good combinations for this technique to be worthwhile.

Harvesting the data produced by an application program is related to the second method, in that it provides values computed by the public methods of the classes, though harvesting ensures a statistical distribution of data much closer to those that appear in practice. One way of harvesting application data would be to insert the test oracles directly as assertions into an application, using the available data values as parameter arguments to the oracle. This would only be safe for stateless data types or copy-assignable data types, otherwise we risk that the oracle itself modifies the state of the application.

Currently, random test data generation seems to be favored by the literature [Gutjahr, 1999, Hamlet and Taylor, 1990, Hamlet, 1994]. Studies of testing efficiency seem to indicate that random testing outperforms most other test set

designs. For any fixed data set size, a carefully chosen data set will normally be better than a random data set, but a slightly larger, often cited as 20% larger, random data set is often just as good [Hamlet, 1994]. Random data generation offers an easy route to expand the data set to any reasonable size.

Similarly to the data invariant, a conditional axiom itself represents a filtering mechanism. A conditional axiom contains an if-statement, and only those data combinations that satisfy the condition will really be tried. Assume that we want to test the transitivity axiom for equality on a user-defined rational number type.

```
if (a == b && b == c) a == c;
```

With the representation of rationals as pairs of integers sketched above, we may compute the equality of $\frac{n_1}{d_1}$ and $\frac{n_2}{d_2}$ by the Boolean expression `n1*d2 == n2*d1` involving integer equality. Choosing arbitrary combinations of integers for nominator and denominators, chances are rather slim we ever will get to the truth part in the transitivity axiom. As in QuickCheck we will provide a warning in such cases, encouraging the user to provide data sets where a significant amount of data reaches the body of the condition. On the other hand, only choosing obviously equal nominator and denominator pairs, skews the data set towards trivially satisfying an axiom, and not providing good tests for the algorithms in general.

Claessen & Hughes also point out that different uses of a data type may benefit from different data distributions. The observation being that the data set of integers which best checks that the integers form a monoid, may not be the ideal data set for array sizes when generating finite array test sets. We see this observation on targeted generation of data sets as very important, and expect the locality we have by associating the data generators with each class will provide this flexibility.

Once the test oracles and the test data machinery are in place, it is easy to run the tests by iterating through the corresponding data set for each of the free variables of each test oracle. However, this easily leads to a combinatorial explosion in the testing size. A test set of 100 elements is quite reasonable, but when we test axioms with several free variables this may become a problem. Take the transitivity axiom. It has three free variables, hence we will test it for one million elements altogether. This may be OK for integers, but what about one million finite arrays? We can deal with this by providing the data generators with a parameter related to the number of arguments in an axiom. Our test generator tool can then fill in this parameter automatically based on the number of free variables in an axiom.

### 8.4.1 Associating a Data Set with a Type

Any type that is part of a universal quantification on an axiom needs to have a test data set associated with it. Our tool expects the user to configure the data generation with a concept map, where relevant types model a concept *HasDataSet*.

For any type, the user must then provide a concept map for *HasDataSet* with a function `get_dataset`, which provides the iterators needed to obtain values of the type. The `for`-loops in Figure 8.3 show how `get_dataset` is used to obtain test data. The exact mechanics of iterators and data source (predefined values, random or generated by some other scheme) is up to the user, but the library provided with the testing tool provides a general implementation which can be used as a basis for generating predefined and random values.

## 8.5 Discussion

There is no reason to believe that writing axioms (or test cases) is any less error-prone than programming in general. Failure of a test can just as well indicate a problem with the axioms or the equals operator as a problem in the implementation. It is important to be aware of this while programming, so that bug-hunting is not exclusively focused on implementation code. The same issue arises with hand-written tests, though, so this is not specific to axiom-based testing. Also, since axioms have a different form than implementation code (equation versus algorithm), it is unlikely that a bug in an axiom and in the implementation will 'cover' for each other so that neither are detected. It is still possible, though; having several axioms covering related behaviour will make this less likely.

Building libraries of well-tested concepts with axioms will increase confidence in the completeness and correctness of the axioms, and reduces the training needed to make effective use of axioms. Not everyone can be expected to know all the laws governing integer arithmetic – but using an existing axiom library and simply stating that "my class should behave like an integer" is easy.

### 8.5.1 Equality Testing

Axiom-based testing (at least with equations) relies on a correct implementation of equality. In many cases, problems with equality will be uncovered in testing, but it is possible to write an implementation of equality that tries to hide most errors – for example, by simply returning true for all arguments (which may be

detected when testing inequalities, unless a != operator has been provided with the same problem).

We expect the equals operator to be a *congruence relation* – an equality relation that is preserved by all functions. This means that it has the usual reflexivity, symmetry and transitivity expected of an equivalence relation, with the additional requirement that all equal objects are treated the same by all functions, i.e. $f(a) = f(b)$ if $a = b$ for all $a$, $b$, and $f$.[4] A straightforward bitwise comparison of two objects will often lack this property. In some cases, such as with floating-point numbers, a usable equals operator will not be truly transitive (due to a small amount of 'fuzz' when comparing, to cover up round-off errors) – this has little impact on our use, however.

The *EqualityComparable* concept in the standard library provides axioms for the equivalence relation of the equality operator and also ensures that inequality operator is the negation of the equality.

It may not always be desirable that the equality operator is a congruence. In the cases we want this property, the relevant axioms should be tool generated, since they will involve every method belonging to the class being tested.

A 'bad' equality operator, returning arbitrary results, will almost certainly be caught during testing since it is basically tested by every axiom in the system relevant for the particular type. Trivial cases like equality always returning true is easily caught by testing based on equality axioms, while more subtle bugs may only show up in general testing, and will be more difficult to trace to the equality operator.

Note that having an equality operator is not strictly necessary. Any type that is *EqualityComparable* is observable in our test oracles, *i.e.*, can be tested on equality. But any type that can be projected on an observable type becomes observable. A projection or context is a term with placeholder for a variable [Zhu, 2003]. This kind of test oracle generation has not been developed in our tool yet and we for the moment require tested types to be *EqualityComparable*.

Note, though, that even if equality is not generally available for a type, it can be provided in a concept map, thus making it available in any template context where the type is constrained to *EqualityComparable*.

### 8.5.2 Algebraic Axioms and Imperative Code

As discussed in Section 8.3.4, a particular problem occurs for code written in an object-oriented or imperative style, relying on side-effects on arguments. Although this is a poor fit for algebraic-style axioms, side-effects can be captured

[4] Informally speaking, since C++ functions may have side-effects or rely on global data.

by using the comma operator. Another issue is that the concept itself must specify whether side-effects occur or not, through the use of non-`const` reference arguments. If an implementation has chosen a different approach, a mapping between the two styles may be given in a concept map, possibly at the expense of an extra temporary. A solution to this problem is provided by *mutification* [Bagge and Haveraaen, 2009], which automatically maps between algebraic and imperative/OO-style code.

In the ASTOOT [Doong and Frankl, 1994] system, algebraic specification of object-oriented programs is done in the LOBAS formalism which supports OO syntax. Each axiom relates object states or values that are computed through a sequence of method calls; optionally, observer functions may be called at the end each sequence to inspect the objects. The system is purely algebraic, allowing no side-effects in operations, except for modifying object state in methods – though a relaxation of this is described by Doong and Frankl [Doong, 1993, Doong and Frankl, 1991]. ASTOOT will automatically generate test drivers from class interfaces, and also generates test cases from a LOBAS algebraic specification. Automated tests can be augmented by manual test generation.

As the C++ axiom proposal allows arbitrary expressions, the ASTOOT / LOBAS-style can easily be used with C++ axioms; though, without disciplined use within same restrictions, there is a danger that side-effects will interfere with testing, as discussed in Section 8.3.4.

The ideas of ASTOOT have been developed further by Chen et al., and applied to axiom-based testing of object-oriented code at the level of class clusters and components [Chen et al., 1998, 2001].

### 8.5.3 Axiom Selection and Algebraic Specification

Early work by Liskov and Zilles [Liskov and Zilles, 1975] discuss techniques for formal specification of abstract data types. They point out that specification should be done by relating the various operations of the abstract data type, rather than directly specifying the input / output of each operation. The latter leads to over-specification, providing many unnecessary details and hiding the essential properties of the data type – for example, by enforcing some order on the elements of an unordered set. Specifying operations in terms of each other avoids bias towards particular representations or implementations. In traditional unit testing, there is always a temptation to over-specify by focusing on testing the input and output of every operation, though a disciplined developer can still avoid over-specification.

In the context of C++ concepts, the concept is separate from the implementation and should avoid putting undue constraints on how the concept may be im-

plemented. Hence, axiom expressions should be limited to using the operations provided in the concept (together with C++'s primitive operations – on booleans, for example – these can be considered implicitly defined in every concept).

Among the techniques discussed by Liskov and Zilles, algebraic specification [Goguen et al., 1978, Guttag and Horning, 1978, Guttag et al., 1978, Liskov and Zilles, 1975] shows the most promise in terms of usability and in avoiding over-specification. An algebraic specification consists of a syntax description and a set of axioms; this maps to the C++ idea of concepts, which provide axioms together with a syntax description in the form of associated types and operations.

To ensure that the behaviour of the abstract data type is fully specified (or *sufficiently complete*) one can divide the operations into *constructors* (the set of which can generate all possible values), *transformers* (which can be defined in terms of constructors) and *observers* (which yield values of another type). Left-hand sides for the axioms of a sufficiently complete specification can then be constructed from the combination of each constructor with every non-constructor. Further guidelines for constructing specifications are discussed by Guttag [Guttag, 1980] and Antoy [Antoy, 1989].

Many of the existing axiom based testing approaches, such as JAX and Daistish, rely on sufficiently complete specifications, provided by complete axiomatisations or initial specifications. This gives extra properties on which to base tools. For example, the approach of Antoy and Hamlet [Antoy and Hamlet, 2000] uses initial specifications, which are evaluated alongside the implementation, as a *direct implementation* [Guttag et al., 1978] of the specification. All objects in the system contain both a concrete value and an abstract value (in the form of a normalised term over constructors in the specification), and the equations from the specification can be evaluated by treating them as rewrite rules on the abstract value terms. A *representation mapping* translates between the abstractions of the specification and the concrete data structures of the implementation. Self-checking functions are made by doing an additional abstract evaluation according to the specification, and – using the representation mapping – comparing the result of normal execution and evaluating the specification. In this way, a whole program can be described and evaluated in two distinct ways – using program code and algebraic specification – providing good protection against programming errors. This is also the disadvantage of the approach – the implementation work must basically be done twice. The overhead of the abstract evaluation and comparison can probably be lowered by running the testing code in a separate thread on a multicore system.

Axioms written in C++ concepts will normally be loose and incomplete, making many of these testing techniques void. The approach described in this paper

will work equally well with an incomplete specification (though, it will of course not be able to test unspecified behaviour). Our experience with developing and testing Sophus [Haveraaen and Brkic, 2005, Haveraaen et al., 2005] shows that such axioms are very useful.

### 8.5.4 Experiences with Axiom-Based Testing

There is currently no large body of code around that uses C++ axioms, since the standard proposal is not yet finished and compiler support is still not mature.[5] A version of the Matrix Template Library [Siek and Lumsdaine, 1998] (MTL) with concepts and axioms is in development and we plan to apply our tool to it as soon as it is ready.

We have experience with axiom-based testing from the Sophus numerical software library [Haveraaen and Brkic, 2005]. This predates C++ axioms, so the tests were written by hand, based on a formal algebraic specification. In our experience, the tests have been useful in uncovering flaws in both the implementation and the specification, though we expect to be able to do more rigorous testing with tool support.

The JAxT tool [Haveraaen and Kalleberg, 2008, Kalleberg, 2007] provides axiom-based testing for Java, by generating tests from algebraic specifications. The axioms are written as static methods and are related to implementation classes through inheritance and interfaces. For any class with axioms, the JAxT tool will generate code that calls the associated axioms. A team of undergraduate students successfully wrote JAxT axioms for parts of the Java collection classes, discovering some weaknesses in the interface specifications in the process [Masood et al., 2009].

The JAX [Stotts et al., 2002] method of combining axioms with the JUnit [JUnit.org, Louridas, 2005] testing framework has provided some valuable insight into the usefulness of axiom-based testing. The JAX developers conducted several informal trials where programmers wrote code and tests using basic JUnit test cases and axiom testing, and found that the axioms uncovered a number of errors that the basic test cases did not detect.

Initial experiences with DAISTS [Gannon et al., 1981] were positive and indicated that it helped users to develop effective tests, avoid weak tests, and the use of insufficient test data. With Daistish [Hughes and Stotts, 1996], the authors did trials similar to those done with JAX, with programming teams reporting that their axioms found errors in code that had already been subjected to traditional unit testing. Testing also uncovered numerous incomplete and erroneous

[5] The prototype ConceptGCC compiler works well in some cases, but is not complete yet.

axioms – the Daistish team note that this is to be expected since the programmers were students learning algebraic specification. This is probably a factor, but some axiom errors can be expected even from trained programmers.

Further experiences and case studies are summarised by Gaudel and Le Gall [Gaudel and Gall, 2008].

### 8.5.5 Tool Implementation

Our implementation is based on the Transformers C++ parsing toolkit [Borghi et al., 2006, Tra, 2006] and the Stratego program transformation language [Bravenboer et al., 2006b]. We have extended Transformers with the new syntax for concepts and axioms, and written a tool, `extract-tests`, that reads C++ with concepts and generates testing code from the concepts and concept maps in the code [Bagge et al., 2008].

As part of our concepts extension to Transformers, we also have an embedding of the Concept C++ grammar into Stratego, so that Stratego transformation rules can be written using concrete C++ syntax. This makes it easy to modify the code templates for the generated code, for instance, changing the test oracles to report success / failure to a testing framework. As an example we use a backend for test oracles that instead of returning a boolean, throws an exception for the CUTE library [Sommerlad, 2009] with the line number of the axiom, so we get test results reported within the Eclipse IDE.

Together with the tool, we have a utility library with basic data generation support, and hooks into a testing framework. This library provides a concept for test data generators. Each type to be tested is expected to have an associated data generator specified through a concept map. This allows the user to specify which generator to use, to create any new kind of generator, and finally to combine streams of generated data.

Since compiling Concept C++ is usually slow, and since generating code directly for pure C++ is complex, the tool is delivered with a Concept C++ to C++ tool translation. Though this tool is not complete, it can still give a sufficient translation to be able to work on a big part of Concept C++ with a standard pure C++ compiler.

### 8.5.6 Future Work

We have identified several areas for improvement throughout this paper. Areas of particular research interest are:

- Perform proper trials to gauge the effectiveness of axiom-based testing and its impact on development.

- Testing of multi-threaded applications is notoriously difficult [Sen, 2007], and it would be interesting to see if axiom-based testing could be applied here.

- As discussed in Section 8.4, there are many open issues with data generation. These will likely only be resolved once we apply the method to realistic-sized projects (like MTL).

There are also much engineering work to be done (in no particular order):

- A library of common concepts with axioms should be written. There has been some work on this already [Gottschling, 2006]. Such concepts should eventually make their way into the C++ standard, for consistency and interoperability.

- Our tool is still experimental, and would need many improvements to be ready for production use. In particular, the underlying framework needs to be developed to handle the kind full-featured C++ code found in mainstream application.

- The tool should be extended with the ability to generate meta-axioms for testing, e.g., congruence axioms for the equality operator or axioms checking the preservation of class invariants in all methods.

- Generate oracles that can test equality on observable types that have no direct equality comparison operator.

## 8.6 Conclusion

The use of axioms and "informal formal methods" has seen a surge in popularity recently. We have presented a method for doing axiom-based testing in the context of proposed concept and axiom features for C++, along with a tool to make generation of such tests automatic.

Both the C++ standard, and programming tools such as compilers are still in development and should be considered 'unstable'. However, our initial experiments with simple test cases show promise, and experiences with axiom-based testing from other languages (both our own and others') encourage us to push forward with tool development and larger-scale experiments.

# Chapter 9

# Conclusion

## 9.1 Tool building

C++ is a challenging language to when writing programming tools by program transformation. The cost of maintenance and development is often very high. Usually, modifications of the language have to be integrated into the compiler. People push the modifications into the standard through the committee so that all compilers integrate the new functionality. This is what happened with C++ 0x. However, experimentation by modifying a compiler implies that the language construct will be handled by only one compiler.

A general purpose language can be extended by providing libraries, but some domain-specific extensions of a language beyond libraries can be required.

C++ proposes a very wide panel of language features such as object-orientation, polymorphism, genericity, etc. Template meta-programming can also help the user with extending the language, However, some features are inaccessible, since template meta-programming does not provide ability to manipulate programs over the whole language, but just expression templates. There are extra tools that are needed but cannot be provided: correctness checking, loop manipulation, declarations, etc.

We have experimented with language features of C++ to enhance software engineering. The transformations we have used for the dissertations are alert handling, testing, concepts and signature morphisms.

There are sufficient tools for C++ analysis. For example, Rose [Davis and Quinlan, 1998] proposes a solution. Though program transformation of C++ is still very hard when the language is modified. This has to do with syntactic

change or with big semantic differences. As stated in the introduction, ways to face this problem are diverse:

- Modify a compiler front-end (ConceptGCC, CC++, ...).
- Use tricks for reusing unused code for other purpose (Codeboost).
- Use pragmas (OpenMP, as well as some transformations with Rose).

In general, a language extension does not survive if there is no maintenance. And in practice, this maintenance has to be done by the compiler developer teams.

Another approach is being minimalistic. In our approach, the only thing we need is a program transformation system that works for C++ and that is simple to work with and is maintained. While we are far from surpassing Rose or Pivot, we provide language extensions that are not touching the base tools.

What often kills language dialects is the cost of maintenance. The few dialects that survived are those which have been adopted by compilers.

The program transformation system we need has a reduced cost of maintenance for extensions. Libraries for example provide an interface that is supposed not to change. Eventually if a very new approach of the library is needed, the older versions are maintained (Qt approach). Or in another approach, the library never changes old functions and only add new functions (DirectX approach). In program transformation for C++, we want the same, but the syntax definition should be a part of the interface. This is what this dissertation proposed.

### 9.1.1 Results for different kinds of extensions

In chapter 5, the transformation for producing pure C is not very complex, but we need type inference for creating temporary variables. For experimentation we start with a `typeof` extension, but not all compilers accept it. Nevertheless type inference in C is not that complex.

In chapter 6, we extend the syntax but do not care about the semantics while transforming. To do this, we try to find a systematic, context-free transformation. It was possible thanks to template meta-programming. We use then the transformation as a simple macro-expansion. It asks for more work on the design, but needs very little implementation work.

Chapter 6 also show as an example how to transform ConceptC++ to C++. We also see that it is not always possible to make a purely context-free transformation. Even for ConceptC++ which seems at first to be good for this purpose, we finally stops on some technical details making it impossible to make a complete

ConceptC++ to C++ compiler. Though it is possible to design an extension that is context-free if we are able to produce a systematic transformation to C++ using template meta-programming. Then the cost of maintenance is very little since the code is very small. The difference with directly do meta-programming is not only the syntax, we also force the translation of code without extension. For instance, in the chapter, we translate the code.

In chapter 8 we do not extend the syntax. We actually produce ConceptC++ from ConceptC++, but we change the semantics. In the problem posed by that chapter, we see we do not need much effort for designing the transformation as well.

## 9.2 Remarks on the extensions and transformations

### 9.2.1 Alerts

The transformation for alert handling has been provided for C. We do not have an actual implementation for C++, but the transformation looks very similar. Though, there are more restrictions on alert declarations due to overloading that need to be considered.

### 9.2.2 Concepts

#### Translation

The main change that the transformation computes is on the lookup. The transformation does not itself check the validity of the code, but what is generated will be checked by the C++ compiler.

Though, the transformation proposed is not complete and does not succeed to provide all the features of the last proposition of concept to the standard. This we try to keep the transformation context-free, but we succeed in covering a major part.

#### Testing

The transformation proposed for testing using ConceptC++ is complete as language extension. However, it is not complete as a testing environment. As it is, we need types to have an equality operator to test its terms, but it is possible to build equality operators from other expression for testing [Zhu, 2003]. This might involve language analysis, but does not directly affect the extension.

Another missing piece is the selection of data. When the number of quantified variables for an axiom grows, the number of tests to call explodes exponentially.

The current implementation loops on the set of data, but the sets of data are provided by a library and not by the transformation. The transformation certainly does not need much modification, but a big effort is needed on providing filters for selecting the right data to provide to the axioms.

## 9.3 Contribution

Extensibility of language is not new. It has not been a problem for many languages. Program transformation of C++ without extensibility, while painful, is still possible to do in an acceptable amount of time.

However, for C++, extensibility has not been answered properly. This dissertation gives an approach. It is hard to see what other language requires such an approach to be applied since most of other languages do not have as much complexity. Though C++ has different dialects, and also different extensions (often abandoned).

On ConceptC++, our work presents some ideas on how a few modifications can help us to use concepts for software building as institutions with usual specification operations.

### 9.3.1 Language design

In the development of testing tools for ConceptC++, we come to the conclusion that we need an equality operator to be defined. Eventually this operator could be automatically built by testing equality of returned values of public method. All in all, we need this operator to be present. It is possible to propose a way to test ConceptC++ without experimenting, but it is not only less convincing, but incomplete. The question now is how do we experiment transformations on ConceptC++ when no front-end is already doing it? In experimenting language design ideas, the approach of extensible language definition has been very helpful.

### 9.3.2 Software

Our contribution on the development has been optimization and bug fixing front-end. This needs of maintenance appeared with the development of the tools presented and specially for the development of the ConceptC++ extension. While the base syntax definition has not been considerably modified (which is good to know), the engine required optimization in complexity of time and memory usage.

## 9.4 Direction for future work

Writing a front-end or analysis by hand has a large advantage: execution speed. The current approach has some difficulties on this side: parsing all possibilities in case of ambiguity, multi-pass evaluation of attribute grammar (AG), parse trees are very general and memory consuming. The two first problems are linked to the fact that parsing of C++ is not linear, but it will be interesting to see how optimization could be done on this point.

The use of the approach will make the tool more robust. Trying more and more extensions is important.

The last thing to be improved is the problem of multiplication of semantic analysis orthogonally to the growing number of extensions. Since the tools do not provide proper semantic analysis, the problem does not occur yet.

# Acronyms

**AG** attribute grammar

**ANTLR** ANother Tool for Language Recognition

**API** application programming interface

**ASF** Algebraic Specification Formalism

**ASF+SDF** Algebraic Specification Formalism + Syntax Definition Formalism

**AST** abstract syntax tree

`apt` Annotation Processing Tool

**BNF** Backus-Naur form

**BSD** Berkeley Software Distribution

**CDT** C/C++ development tools

**DSL** domain-specific language

**EDG** Edison Design Group

**EGCS** Experimental/Enhanced GNU Compiler System

**FOG** Flexible Object Generator

**IPR** Internal Program Representation

**GCC** GNU Compiler Collection

**GPL** GNU General Public License

**GPP** Generic Pretty Printer

**GPU** Graphics Processing Unit

**GLR** Generalized LR

**IDE** integrated development environment

**KDE** K Desktop Environment

**LGPL** GNU Lesser General Public License

**LALR** lookahead LR parser

**MOP** metaobject protocol

**PT** parse tree

**SDF** Syntax Definition Formalism

**SGLR** Scannerless Generalized LR

**SQL** Structured Query Language

**TXL** Turing eXtender Language

**UUAG** Utrecht University Attribute Grammar system

**XML** eXtensible Markup Language

**XR** eXternal Program Representation

# Bibliography

Robert L. Akers, Ira D. Baxter, Michael Mehlich, Brian J. Ellis, and Kenn R. Luecke. Reengineering C++ component models via automatic program transformation. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2474-5. doi: http://dx.doi.org/10.1109/WCRE.2005.25.

Robert Anisko, Valentin David, and Clement Vasseur. Transformers: A C++ program transformation framework. Technical Report 0310, EPITA/LRDE, 2003.

G. Antoniol, M. Di Penta, G. Masone, and U. Villano. Compiler hacking for source code analysis. *Software Quality Control*, 12(4):383–406, 2004. ISSN 0963-9314. doi: http://dx.doi.org/10.1023/B:SQJO.0000039794.29432.7e.

Sergio Antoy. Systematic design of algebraic specifications. In *IWSSD '89: Proceedings of the 5th international workshop on Software specification and design*, pages 278–280, New York, NY, USA, 1989. ACM. ISBN 0-89791-305-1. doi: http://doi.acm.org/10.1145/75199.75241.

Sergio Antoy and Richard G. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Trans. Software Eng.*, 26(1):55–69, 2000.

Arthur Baars, Doaitse Swierstra, and Andres Löh. Utrecht University Attribute Grammar System, 1999. `http://www.cs.uu.nl/wiki/Center/AttributeGrammarSystem`.

Anya Helene Bagge and Magne Haveraaen. Interfacing concepts: Why declaration style shouldn't matter. In Torbjörn Ekman and Jurgen J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools*

*and Applications (LDTA '09)*, Electronic Notes in Theoretical Computer Science, York, UK, March 2009. Elsevier.

Anya Helene Bagge and Magne Haveraaen. Axiom-based transformations: Optimisation and testing. In Electronic Notes in Theoretical Computer Science, editor, *Proceedings of the 8th Workshop on Language Description, Tools and Applications (LDTA'08)*, Budapest, Hungary, April 2008. Elsevier.

Anya Helene Bagge, Valentin David, and Magne Haveraaen. Axiom-based testing for C++. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 721–722, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-220-7. doi: http://doi.acm.org/10.1145/1449814.1449829.

Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 65–74, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

J Barnes. *High Integrity Ada. The SPARK Approach*. Addison-Wesley, 1997.

Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms: Program transformations for practical scalable software evolution. In *International Conference on Software Engineering (ICSE)*, pages 625–634. IEEE Computer Society, 2004.

Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2002. ISBN 0321146530.

Kent Beck. Extreme programming: A humanistic discipline of software development. In *Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering (FASE'98)*, pages 1–6. Springer Berlin / Heidelberg, 1998.

Pete Becker. Working draft, standard for programming language C++. Technical Report N2798, ISO/IEC, 2009.

Jean-Philippe Bernardy, Patrik Jansson, Marcin Zalewski, Sibylle Schupp, and Andreas Priesnitz. A comparison of c++ concepts and haskell type classes. In *WGP '08: Proceedings of the ACM SIGPLAN workshop on Generic*

*programming*, pages 37–48, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-060-9. doi: http://doi.acm.org/10.1145/1411318.1411324.

Alexandre Borghi, Valentin David, and Akim Demaille. C-Transformers — A framework to write C program transformations. *ACM Crossroads*, 12(3), Spring 2006. `http://www.acm.org/crossroads/xrds12-3/contractc.html`.

Alexander Borgida. Language features for flexible handling of exceptions in information systems. *ACM Trans. Database Syst.*, 10(4):565–603, 1985. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/4879.4995.

Marat Boshernitsan. *Program manipulation via interactive transformations*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2006. Adviser-Graham, Susan L.

Mark van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

Mark van den Brand, A. van Deursen, J. Heering, H. de Jonge, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction 2001 (CC'2001)*, volume 2027 of *LNCS*, pages 365 – 370. Springer-Verlag, 2001.

Mark van den Brand, Steven Klusener, Leon Moonen, and Jurgen J. Vinju. Generalized parsing and term rewriting: Semantics driven disambiguation. In B.R. Bryant and J. Saraiva, editors, *Third Workshop on Language Descriptions, Tools and Applications*, volume 82 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.

Mark G. J. van den Brand, Jeroen Scheerder, Jurgen Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.

Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and*

*Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

Martin Bravenboer, René de Groot, and Eelco Visser. MetaBorg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In R. Lämmel and J. Saraiva, editors, *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, volume 4143 of *Lecture Notes in Computer Science*, pages 297–311, Braga, Portugal, 2006a. Springer Verlag.

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.16: Components for transformation systems. In Frank Tip and John Hatcliff, editors, *PEPM'06: Workshop on Partial Evaluation and Program Manipulation*. ACM Press, January 2006b.

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 2008. (To appear).

Nicolas Burrus, Alexandre Duret-Lutz, Thierry Géraud, David Lesage, and Raphaël Poss. A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In *Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL)*, Anaheim, CA, USA, October 2003.

Rod M. Burstall and Joseph A. Goguen. Putting theories together to make specifications. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058, 1977.

C-Alert. Alert extension for C. URL `http://www.codeboost.org/alert/`.

S. Callanan, D. J. Dean, and E. Zadok. Extending GCC with modular GIMPLE optimizations. In *Proceedings of the 2007 GCC Developers' Summit*, Ottawa, Canada, July 2007.

Robert D. Cameron and M. Robert Ito. Grammar-based definition of metaprogramming systems. *ACM Trans. Program. Lang. Syst.*, 6(1):20–54, 1984. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357233.357235.

Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

K. Mani Chandy and Carl Kesselman. Cc++: a declarative concurrent object-oriented programming notation. In *Research directions in concurrent object-oriented programming*, pages 281–313. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-01139-5.

R. Chapman, A. Wellings, and A. Burns. Worst-case timing analysis of exception handling in Ada. In L. Collingbourne, editor, *Ada: Towards Maturity. Proceedings of the 1993 AdaUK conference*, pages 148–164. IOS Press: London, 1993.

Huo Yan Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 7(3):250–295, 1998. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/287000.287004.

Huo Yan Chen, T. H. Tse, and Tsong Yueh Chen. Taccle: a methodology for object-oriented software testing at the class and cluster levels. *ACM Trans. Softw. Eng. Methodol.*, 10(1):56–109, 2001. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/366378.366380.

Shigeru Chiba. A metaobject protocol for C++. *SIGPLAN Not.*, 30(10):285–299, 1995. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/217839.217868.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: http://doi.acm.org/10.1145/351240.351266.

M.L. Collard, H.H. Kagdi, and J.I. Maletic. An xml-based lightweight C++ fact extractor. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 134, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1883-4.

Jonathan Corbet. GCC unplugged. *Linux Weekly News*, November 2007. URL `http://lwn.net/Articles/259157/`.

James R. Cordy. TXL - a language for programming language tools and applications. In *Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA'04)*, 2004.

Lawrence Crowl and Thorsten Ottosen. Proposal to add contract programming to C++ (revision 3). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1866.html`, August 2005.

Qian Cui and John Gannon. Data-oriented exception handling. *IEEE Trans. Softw. Eng.*, 18(5):393–401, 1992. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.135772.

Krzysztof Czarnecki, John T. OrsquoDonnell, Jörg Striegnitz3, and Walid Taha4. DSL implementation in MetaOCaml, Template Haskell, and C++. In *Domain-Specific Program Generation*, 2004.

L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

Valentin David. Preparing for C++0x. In *OOPSLA '08: Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, New York, NY, USA, 2008. ACM.

Kei Davis and Dan Quinlan. Rose: An optimizing code transformer for c++ object-oriented array class libraries. In *Workshop on Parallel Object-Oriented Scientific Computing (POOSC'98), at 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium*, volume 1543 of *Lecture Notes in Computer Science*. Springer Verlag, July 1998.

Thomas R. Dean, Andrew J. Malton, and Ric Holt. Union schemas as a basis for a C++ extractor. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 59, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1303-4.

Christophe Dony. Exception handling and object-oriented programming: Towards a synthesis. *SIGPLAN Not.*, 25(10):322–330, 1990. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/97946.97984.

Roong-Ko Doong. *An approach to testing object-oriented programs*. PhD thesis, Polytechnic University, Brooklyn, NY, USA, 1993.

Roong-Ko Doong and Phyllis G. Frankl. Case studies on testing object-oriented programs. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 165–177, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-449-X. doi: http://doi.acm.org/10.1145/120807.120822.

Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2):101–130, 1994. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/192218.192221.

Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: http://doi.acm.org/10.1145/1111037.1111064.

Alexandre Duret-Lutz. Olena: a component-based platform for image processing, mixing generic, generative and OO programming. In *Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE)—Young Researchers Workshop; published in "Net.ObjectDays2000"*, pages 653–659, Erfurt, Germany, October 2000. ISBN 3-89683-932-2.

Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. *SIGPLAN Not.*, 42(10):1–18, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1297105.1297029.

EPITA Research and Developpement Laboratory (LRDE). The Olena image processing library. `http://olena.lrde.epita.fr`, 2009.

John Gannon, Paul McMullin, and Richard Hamlet. Data abstraction, implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357139.357140.

Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1995. ISBN 3-540-59293-8.

Marie-Claude Gaudel and Pascale Le Gall. Testing data types implementations from algebraic specifications. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 209–239. Springer, 2008. doi: 10.1007/978-3-540-78917-8.

Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice Hall, 1978.

David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/103162.103163.

John B. Goodenough. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/361227.361230.

James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, USA, 1996. ISBN 0-201-63451-1.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java$^{TM}$ Language Specification, The (3rd Edition)*. Addison-Wesley Professional, 2005. ISBN 0321246780.

Peter Gottschling. Fundamental algebraic concepts in concept-enabled C++. Technical Report 638, Indiana University, 2006.

Douglas Gregor and Andrew Lumsdaine. Core concepts for the C++0x standard library (revision 2). Technical Report N2621=08-0131, JTC1/SC22/WG21 – The C++ Standards Committee, May 2008.

Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 291–310, New York, NY, USA, 2006a. ACM. ISBN 1-59593-348-4. doi: http://doi.acm.org/10.1145/1167473.1167499.

Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Gabriel Dos Reis, Bjarne Stroustrup, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'06)*, October 2006b.

Walter J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, 1999. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/32.815325.

John V. Guttag. Notes on type abstraction (version 2). *IEEE Trans. Softw. Eng.*, 6(1):13–23, 1980. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1980.230209.

John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Inf.*, 10:27–52, 1978.

John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract data types and software validation. *Commun. ACM*, 21(12):1048–1064, 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359657.359666.

Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. *IEEE Trans. Softw. Eng.*, 16(12):1402–1411, 1990. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.62448.

Richard Hamlet. Random testing. In J Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

Magne Haveraaen. Institutions, property-aware programming and testing. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 21–30, New York, NY, USA, 2007. ACM. ISBN 978-1-60558-086-9. doi: http://doi.acm.org/10.1145/1512762.1512765.

Magne Haveraaen and Enida Brkic. Structured testing in Sophus. In Eivind Coward, editor, *Norsk informatikkonferanse NIK'2005*, pages 43–54. Tapir akademisk forlag, Trondheim, Norway, 2005.

Magne Haveraaen and Helmer Andr'e Friis. Coordinate-free numerics: all your variation points for free? *Int. J. Comput. Sci. Eng.*, page to appear, 2009. ISSN 1742-7185.

Magne Haveraaen and Karl Trygve Kalleberg. JAxT and JDI: the simplicity of JUnit applied to axioms and data invariants. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 731–732, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-220-7. doi: http://doi.acm.org/10.1145/1449814.1449834.

Magne Haveraaen, Helmer André Friis, and Tor Arne Johansen. Formal software engineering for computational modelling. *Nord. J. Comput.*, 6(3): 241–270, 1999.

Magne Haveraaen, Helmer André Friis, and Hans Munthe-Kaas. Computable scalar fields: a basis for PDE software. *Journal of Logic and Algebraic Programming*, 65(1):36–49, September-October 2005. doi: 10.1016/j.jlap.2004.12.001.

Mark Hennessy, Brian A. Malloy, and James F. Power. gccxfront: Exploiting gcc as a front end for program comprehension tools via xml/xslt. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program*

*Comprehension*, page 298, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1883-4.

I.D. Hill. Faults in function, in Algol and Fortran. *Comp.J.*, 14(3), 1970.

Markus Hof, Hanspeter Mössenböck, and Peter Pirkelbauer. Zero-overhead exeption handling using metaprogramming. In *SOFSEM '97: Proceedings of the 24th Seminar on Current Trends in Theory and Practice of Informatics*, pages 423–431, London, UK, 1997. Springer-Verlag. ISBN 3-540-63774-5.

Merlin Hughes and David Stotts. Daistish: systematic algebraic testing for OO programs in the presence of side-effects. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 53–61, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-787-1. doi: http://doi.acm.org/10.1145/229000.226301.

ISE Software. Building bug-free O-O software: An introduction to design by contract(TM). `http://archive.eiffel.com/doc/manuals/technology/contract/page.html`, 1993.

Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.562936. `http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html`.

Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-880446-00-6.

Merijn de Jonge and Joost Visser. Grammars as contracts. In Greg Butler and Stan Jarzabek, editors, *Generative and Component-Based Software Engineering, Second International Symposion, GCSE 2000*, volume 2177 of *Lecture Notes in Computer Science*, pages 85–99, Erfurt, Germany, October 2001. Springer.

Merijn de Jonge, Eelco Visser, and Joost Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand and D. Parigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, April 2001.

Merjin de Jonge. A pretty-printer for every occasion. In Ian Ferguson, Jonathan Gray, and Louise Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*, pages 68–77. University of Wollongong, Australia, June 2000.

JUnit.org. Junit. URL `http://junit.org/`.

Karl Trygve Kalleberg. *Abstractions for Language-Independent Program Transformations*, chapter 11. University of Bergen, Norway, Postboks 7800, 5020 Bergen, Norway, June 2007. ISBN 978-82-308-0441-4.

Michael Karasick. The architecture of montana: an open and extensible programming environment with an incremental c++ compiler. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 131–142, New York, NY, USA, 1998. ACM. ISBN 1-58113-108-9. doi: http://doi.acm.org/10.1145/288195.288284.

Carl Kesselman. High performance parallel and distributed computation in compositional CC++. *SIGAPP Appl. Comput. Rev.*, 4(1):24–26, 1996. ISSN 1559-6915. doi: http://doi.acm.org/10.1145/240732.240741.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and Willian G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001: Object-Oriented Programming: 15th European Conference*, volume 2072 of *LNCS*, pages 327–353. Springer-Verlag, June 2001. ISBN 3-540-42206-4.

Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy, October 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

Donald E. Knuth. Semantics of context-free languages. *Journal of Mathematical System Theory*, pages 127–145, 1968.

Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. *ICSE*, 00:418, 2000. doi: http://doi.ieeecomputersociety.org/10.1109/ICSE.2000.10148.

Barbara Liskov. A history of CLU. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 133–147, New York,

NY, USA, 1993. ACM Press. ISBN 0-89791-570-4. doi: http://doi.acm.org/10.1145/154766.155367.

Barbara Liskov and Stephen Zilles. Specification techniques for data abstractions. In *Proceedings of the international conference on Reliable software*, pages 72–87, New York, NY, USA, 1975. ACM. doi: http://doi.acm.org/10.1145/800027.808426.

Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564–576, 1977. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359763.359789.

Sylvain Lombardy, Raphaël Poss, Yann Régis-Gianas, and Jacques Sakarovitch. Introducing Vaucanson. In Springer-Verlag, editor, *Proceedings of Implementation and Application of Automata, 8th International Conference (CIAA)*, volume 2759 of *Lecture Notes in Computer Science Series*, pages 96–107, Santa Barbara, CA, USA, July 2003.

R.J. Lorimer. Apt: Compile-time annotation processing with java. *Javalobby*, 2005.

Panagiotis Louridas. JUnit: Unit testing and coding in tandem. *IEEE Softw.*, 22 (4):12–15, 2005. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2005.100.

LRDE — EPITA Research and Developpement Laboratory. Transformers home page, 2005. `http://transformers.lrde.epita.fr`.

David C. Luckham and W. Polak. Ada exception handling: an axiomatic approach. *ACM Trans. Program. Lang. Syst.*, 2(2):225–233, 1980. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357094.357100.

M. Donald MacLaren. Exception handling in PL/I. In *Proceedings of an ACM conference on Language design for reliable software*, pages 101–104. ACM Press, 1977. doi: http://doi.acm.org/10.1145/800022.808316.

Brian A. Malloy, Tanton H. Gibbs, and James F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Softw. Pract. Exper.*, 33(1):19–39, 2003. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.493.

Madhia Masood, Erlend Birkenes, Karl Trygve Kalleberg, Magne Haveraaen, and Anya Helene Bagge. Axiom-based testing of Java collections with JAxT.

Technical Report 388, Department of Informatics, University of Bergen, P.O.Box 7803, N-5020 Bergen, Norway, August 2009.

Scott McPeak. *Elkhound: A Fast, Practical GLR Parser Generator*, Dec 2002. http://www.cs.berkeley.edu/~smcpeak/elkhound/.

Bertrand Meyer. *Eiffel: The language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN 0-13-247925-7.

Bertrand Meyer. *Object-Oriented Software Construction, 2nd ed.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997. ISBN 0136291554.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997. ISBN 0262631814.

George C. Necula. CIL home page, 2005. http://sourceforge.net/projects/cil.

George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC'02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4.

Karina Olmos and Eelco Visser. Strategies for source-to-source constant propagation. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies (WRS'02)*, volume 70 of *Electronic Notes in Theoretical Computer Science*, page 20, Copenhagen, Denmark, July 2002. Elsevier Science Publishers.

POSIX.1. *The Single UNIX® Specification, Version 3 / IEEE Std 1003.1-2001*. The Open Group, 2004. URL http://www.unix.org/version3/online.html.

openmp.org. The OpenMP API specification for parallel programming. URL http://www.openmp.org/.

Yoann Padioleau. Parsing c/c++ code without pre-processing. In *CC '09: Proceedings of the 18th International Conference on Compiler Construction*, pages 109–125, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00721-7. doi: http://dx.doi.org/10.1007/978-3-642-00722-4_9.

Kent M. Pitman. Condition handling in the Lisp language family. In *Advances in Exception Handling Techniques*, pages 39–59. Springer-Verlag, 2000.

Wishnu Prasetya, Tanja Vos, and Arthur Baars. Trace-based reflexive testing of OO programs. Technical Report UU-CS-2007-037, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2007.

B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM Press. doi: http://doi.acm.org/10.1145/800027.808467.

Alexander Romanovsky and Bo Sandén. Except for exception handling ... *Ada Lett.*, XXI(3):19–25, 2001. ISSN 1094-3641. doi: http://doi.acm.org/10.1145/568671.568678.

David Saff. Theory-infected: or how I learned to stop worrying and love universal quantification. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 846–847, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-865-7. doi: http://doi.acm.org/10.1145/1297846.1297919.

Donald Sannella and Andrzej Tarlecki. Extended ml: an institution-independent framework for formal program development. In *Proc. Workshop on Category Theory and Computer Programming*, pages 364–389. Springer, 1986.

Donald Sannella and Andrzej Tarlecki. Extended ml: Past, present and future. In *Proc. 7th Workshop on Specification of Abstract Data Types, Wusterhausen. Springer LNCS 534*, pages 297–322. Springer-Verlag, 1991.

Markus Schordan and Daniel Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, August 2003.

Koushik Sen. Effective random testing of concurrent programs. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 323–332, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: http://doi.acm.org/10.1145/1321631.1321679.

Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko JÃ¤rvi, and Andrew Lumsdaine. Concepts for C++0x. Technical Report N1758, ISO/IEC, 2005.

Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A unifying framework for numerical linear algebra. In *ECOOP '98: Workshop on Object-Oriented Technology*, pages 466–467, London, UK, 1998. Springer-Verlag. ISBN 3-540-65460-7.

Jeremy G. Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, 2000.

Brian Cantwell Smith. *Procedural reflection in programming languages*. PhD thesis, Massachusetts Institute of Technology, 1982.

Peter Sommerlad. *C++ Unit Testing Easier*, 2009. URL `http://r2.ifs.hsr.ch/cute`.

Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Know.-Based Syst.*, 20(7):636–651, 2007. ISSN 0950-7051. doi: http://dx.doi.org/10.1016/j.knosys.2007.05.004.

Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002.

Alexander Stepanov and Meng Lee. The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project, 1994.

P. David Stotts, Mark Lindsey, and Angus Antley. An informal formal method for systematic JUnit test case generation. In Don Wells and Laurie A. Williams, editors, *XP/Agile Universe*, volume 2418 of *Lecture Notes in Computer Science*, pages 131–143. Springer, 2002. ISBN 3-540-44024-0.

stratego-language.org. Stratego/XT. URL `http://www.stratego-language.org/`.

Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, third edition, 1997.

Bjarne Stroustrup. The design of C++0x. *C/C++ Users Journal*, May 2005a.

Bjarne Stroustrup. A rationale for semantically enhanced library languages. In *Proceedings of the First International Workshop on Library-Centric Software Design (LCSD05)*, 2005b.

Bjarne Stroustrup and Gabriel Dos Reis. Supporting sell for high-performance computing. In *In LCPC 05, volume 4339 of LNCS*, pages 458–465. Springer-Verlag, 2005.

Herb Sutter. C++ conformance roundup. *C/C++ Users Journal*, 19(4), April 2001. ISSN 1075-2838.

Herb Sutter. Why we can't afford export. Technical Report N1426, ISO/IEC, 2003.

S. Tucker Taft, Robert A. Duff, Randall Brukardt, and Erhard Plödereder, editors. *Consolidated Ada Reference Manual. Language and Standard Libraries, International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1*, volume 2219 of *Lecture Notes in Computer Science*. Springer, 2001. ISBN 3-540-43038-5.

Xiaolong Tang and Jaakko Järvi. Concept-based optimization. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 97–108, New York, NY, USA, 2007a. ACM. ISBN 978-1-60558-086-9. doi: http://doi.acm.org/10.1145/1512762.1512772.

Xiaolong Tang and Jaakko Järvi. Concept-based optimization. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 97–108, New York, NY, USA, 2007b. ACM. ISBN 978-1-60558-086-9. doi: http://doi.acm.org/10.1145/1512762.1512772.

VAUCANSON group. VAUCANSON home page, 2007. `http://vaucanson.lrde.epita.fr/`.

Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985. ISBN 0898382025.

*Transformers*. The Transformers Group, LRDE, EPITA, 2006. URL `http://www.lrde.epita.fr/cgi-bin/twiki/view/Transformers/Transformers`.

Hoang Truong. Guaranteeing resource bounds for component software. In *FMOODS'05*, volume 3535 of *LNCS*, pages 179–194. Springer, 2005. ISBN 3-540-26181-8.

Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/567097.567099.

M.G.J. van den Brand and P. Klint. Aterms for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology*, 49(1):55–64, 2007.

Guido van Rossum and Fred L. Drake, Jr. *Python Language Reference Manual*. Network Theory Ltd, Bristol, UK, 2003. ISBN 0-9541617-8-5.

E. Van Wyk, D. Bodin, L. Krishnan, and J. Gao. Silver: an extensible attribute grammar system. *Scinece of Computer Programming*, 2009. Accpeted, In Press.

Clément Vasseur. Semantics driven disambiguation: a comparison of different approaches. Technical report, LRDE, 2004.

Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. ISSN 1040-6042. Reprinted in C++ Gems, ed. Stanley Lippman.

Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

Eelco Visser. Meta-programming with concrete object syntax. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002a. Springer-Verlag.

Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, June 2004.

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

Eelco Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, July 1997b.

Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997c.

Eelco Visser. A bootstrapped compiler for strategies (extended abstract). In B. Gramlich, H. Kirchner, and F. Pfenning, editors, *Strategies in Automated Deduction (STRATEGIES'99)*, pages 73–83, Trento, Italy, July 5 1999.

Eelco Visser. Tiger in Stratego, 2002b. `http://www.program-transformation.org/Tiger/WebHome`.

D. G. Waddington and B. Yao. High fidelity C++ code transformation. In *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, Electronic Notes in Theoretical Computer Science, Edinburgh University, UK, April 3 2005.

Daniel Waddington and Bin Yao. High-fidelity C/C++ code transformation. *Sci. Comput. Program.*, 68(2):64–78, 2007. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2006.04.010.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: http://doi.acm.org/10.1145/75277.75283.

Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52. Springer, 1995.

WG14. *ISO/IEC 9899:1999 (E). Programming languages — C*, 1999.

WG21. *ISO/IEC 14882:2003: Programming languages — C++*, 2003.

Edward D. Willink. *Meta-Compilation for C++*. PhD thesis, University of Surrey, UK, 2001.

Bin Yao, W. Mielke, S. Kennedy, and R. Buskens. C macro handling in automated source code transformation systems. In *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, pages 68–69, Sept. 2006. doi: 10.1109/ICSM.2006.16.

Bo Yu, Liang Kong, Yufeng Zhang, and Hong Zhu. Testing java components based on algebraic specifications. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 190–199, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3127-4. doi: http://dx.doi.org/10.1109/ICST.2008.39.

Marcin Zalewski. *Generic Programming with Concepts*. PhD thesis, University of Göteborg, Sweden, 2008.

Marcin Zalewski and Sibylle Schupp. C++ concepts as institutions: a specification view on concepts. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 76–87, New York, NY, USA, 2007. ACM. ISBN 978-1-60558-086-9. doi: http://doi.acm.org/10.1145/1512762.1512770.

Hong Zhu. A note on test oracles and semantics of algebraic specifications. In *QSIC '03: Proceedings of the Third International Conference on Quality Software*, pages 91–98, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2015-4.

Mikal Ziane. Towards tool support for design patterns using program transformations. In *Langages et Modèles à Objets (LMO)*, volume 7, pages 199–214, Le Croisic, January 2001. Hermès Science Publications.