PROGRAMMING LANGUAGE EVOLUTION AND SOURCE CODE REJUVENATION

A Dissertation

by

PETER MATHIAS PIRKELBAUER

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2010

Major Subject: Computer Science

PROGRAMMING LANGUAGE EVOLUTION AND SOURCE CODE REJUVENATION

A Dissertation

by

PETER MATHIAS PIRKELBAUER

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Bjarne Stroustrup
Committee Members,	Gabriel Dos Reis
	Lawrence Rauchwerger
	Jaakko Järvi
	Weiping Shi
Head of Department,	Valerie Taylor

December 2010

Major Subject: Computer Science

ABSTRACT

Programming Language Evolution

and Source Code Rejuvenation.

(December 2010)

Peter Mathias Pirkelbauer, Dipl.-Ing., Johannes-Kepler Universität Linz, Austria;

M.B.A., Texas A&M University

Chair of Advisory Committee: Dr. Bjarne Stroustrup

Programmers rely on programming idioms, design patterns, and workaround techniques to express fundamental design not directly supported by the language. Evolving languages often address frequently encountered problems by adding language and library support to subsequent releases. By using new features, programmers can express their intent more directly. As new concerns, such as parallelism or security, arise, early idioms and language facilities can become serious liabilities. Modern code sometimes benefits from optimization techniques not feasible for code that uses less expressive constructs. Manual source code migration is expensive, time-consuming, and prone to errors.

This dissertation discusses the introduction of new language features and libraries, exemplified by open-methods and a non-blocking growable array library. We describe the relationship of open-methods to various alternative implementation techniques. The benefits of open-methods materialize in simpler code, better performance, and similar memory footprint when compared to using alternative implementation techniques.

Based on these findings, we develop the notion of *source code rejuvenation*, the automated migration of legacy code. Source code rejuvenation leverages enhanced program language and library facilities by finding and replacing coding patterns that can be expressed through higher-level software abstractions. Raising the level of abstraction improves code quality by lowering software entropy. In conjunction with extensions to programming languages, source code rejuvenation offers an evolutionary trajectory towards more reliable, more secure, and better performing code.

We describe the tools that allow us efficient implementations of code rejuvenations. The Pivot source-to-source translation infrastructure and its traversal mechanism forms the core of our machinery. In order to free programmers from representation details, we use a light-weight pattern matching generator that turns a C++ like input language into pattern matching code. The generated code integrates seamlessly with the rest of the analysis framework.

We utilize the framework to build analysis systems that find common workaround techniques for designated language extensions of C++0x (e.g., initializer lists). Moreover, we describe a novel system (TACE — template analysis and concept extraction) for the analysis of uninstantiated template code. Our tool automatically extracts requirements from the body of template functions. TACE helps programmers understand the requirements that their code de facto imposes on arguments and compare those de facto requirements to formal and informal specifications.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER I

INTRODUCTION

Popular programming languages evolve over time. One driver of evolution is a desire to simplify the use of these languages in real life projects. For example, in the early 1970s the C programming language was developed as a system programming language for the UNIX operating system on the PDP-11 [152]. With the proliferation of UNIX to other platforms, C evolved to reflect concerns, such as portability and type safety. Later Stroustrup enhanced C with higher level abstractions to simplify the development of a distributed and modularized UNIX kernel [170]. Compared to C, ISO C++ [97] directly supports the program design with classes, dynamic dispatch, templates, exception handling, and more [170]. Abstraction mechanisms present in C++ can be compiled to efficient machine code on many architectures. This makes C++ suitable for software development of embedded systems, desktop computers, and mainframe architectures. C++'s proliferation and success is a constant source of ideas for enhancements and extensions. The ISO C++ standards committee has released a draft of the next revision of the C++ standard, commonly referred to as C++0x [173, 27]. C++0x will address a number of modeling problems (e.g., object initialization) by providing better language and library support. Until compiler and library implementations of C++0x become widely available, C++ programmers solve these problems through programming idioms and workaround techniques. These "solutions" are typically more involved than they would be in C++0x and can easily become another source of errors and maintenance problems.

This dissertation follows the style of Science of Computer Programming.

This dissertation draws its examples from C++0x's proposed extensions (as of May 2009) to the language and its standardized libraries [25], but the topic is equally valid for other widely used and evolving languages, such as Python [186], C# [68], and Java [14]. For example, Java is currently undergoing its sixth major revision since its first release in 1995. Extensions under consideration for Java 7 include support for closures, null-safe method invocations, extended catch clauses to catch and rethrow groups of exceptions, type inference for generics and others [127].

I.A. What Is Source Code Rejuvenation?

Source code rejuvenation is a source-to-source transformation that replaces deprecated language features and idioms with modern code. Old code typically contains outdated idioms, elaborate and often complex coding patterns, deprecated language features (or data structures). The rejuvenated code is more concise, safer, and uses higher level abstractions. What we call outdated idioms (and patterns) are techniques often developed in response to the lack of direct language support. When programming languages and techniques evolve, these coding styles become legacy code, as programmers will express new code in terms of new language features. This leads to a mix of coding styles, which complicates a programmer's understanding of source code and can cause maintenance problems. Furthermore, teaching and learning can be greatly simplified by eliminating outdated language features and idioms.

Source code rejuvenation is a *unidirectional process* that detects coding techniques expressed in terms of lower-level language and converts them into code using higher-level abstractions. High-level abstractions make information explicit to programmers and compilers that would otherwise remain buried in more involved code. We aim to automate many forms of code rejuvenation and to provide program assistance for cases where human intervention is necessary. In other words, our aim is nothing less than to reverse (some forms of) (software) entropy!

Preserving behavioral equivalence between code transformations is necessary to claim correctness. In the context of source code rejuvenation, a strict interpretation of behavior preservation would disallow meaningful transformations (e.g., see the initializer list example §I.B.1). We therefore argue that a valid source code rejuvenation preserves or improves a program's behavior. In addition, when a rejuvenation tool detects a potential problem but does not have sufficient information to guarantee a correct code transformation, it can point the programmer to potential trouble spots and suggest rejuvenation. For example, a tool can propose the use of the C++0x's array class instead of C style arrays. A C++0x array object passed as a function argument does not decay to a pointer. The argument retains its size information, which allows a rejuvenation tool to suggest bounds checking of data accesses in functions that take arrays as parameters.

I.A.1. Applications

Source code rejuvenation is an enabling technology and tool support for source code rejuvenation leverages the new languages capabilities in several aspects:

Source Code Migration: Upgrading to the next iteration of a language can invalidate existing code. For example, a language can choose to improve static type safety by tightening the type checking rules. As a result, formerly valid code produces pesty error or warning messages. An example is Java's introduction of generics. Starting with Java 5 the compiler warns about the unparametrized use of Java's container classes.

Even with source code remaining valid, automated source code migration makes the transition to new language versions smoother. For example, programmers would not need to understand and maintain source files that use various workaround techniques instead of (later added) language constructs. For example, a project might use template based libraries (e.g., Standard Template Library (STL) [15], STAPL [13]) where some were developed for C++03 and others for C++0x. In such a situation, programmers are required to understand both.

Education: Integration with a smart IDE enables "live" suggestions that can replace workarounds/idioms with new language constructs, thereby educating programmers on how to better use available language and library constructs.

Optimization: The detection of workarounds and idioms can contribute a significant factor to both the development cost of a compiler and the runtime, as the detection and transformation requires time. Compiler vendors are often reluctant to add optimizations for every possible scenario. The introduction of new language constructs can enable more and better optimizations (e.g., `const_expr` lets the compiler evaluate expressions at compile time [27]). Automated source code migration that performs a one-time source code transformation to utilize the new language support enables optimizations that might be forgone otherwise.

I.B. Case Study

In this section, we demonstrate source code rejuvenation with examples taken from C++0x, namely initializer lists and concept extraction.

I.B.1. Initializer Lists

In current C++, the initialization of a container (or any other object) with an arbitrary number of different values is cumbersome. When needed, programmers deal with the problem by employing different initialization idioms.

Consider initializing a `vector` of `int` with three constant elements (e.g., 1, 2, 3). Techniques to achieve this include writing three consecutive `push_back` operations, and copying constants from an array of `int`. We can "initialize" through a series of `push_back()`s:

```
// using namespace std;
vector<int> vec;
// three consecutive push_backs
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
```

Alternatively, we can initialize an array and use that to initialize the vector:

```
// copying from an array
int a[] = {1, 2, 3};
vector<int> vec(a,a+sizeof(a)/sizeof(int));
```

These are just the two simplest examples of such workarounds observed in real code. Although the described initialization techniques look trivial, it is easy to accidentally write erroneous or non-optimal code. For example, in the first example the `vector` resizes its internal data structure whenever the allocated memory capacity is insufficient to store a new value; in some situations that may be a performance problem. The second example is simply a technique that people often get wrong (e.g. by using the wrong array type or by specifying the wrong size for the vector). Other workarounds tend to be longer, more complicated, and more-error prone. Rejuvenating the code to use C++0x's initializer list construction [3][120] automatically remedies this problem.

```
// rejuvenated source code in C++0x
vector<int> vec = {1, 2, 3};
```

In C++0x, the list of values (`1, 2, 3`) becomes an initializer list. Initializer list constructors take a list of values as argument and construct an initial object state. As a result, the rejuvenated source code is more concise. It needs only one line of code

(LOC), when compared to two and four LOCs needed by the workaround techniques. The rejuvenated source code is more uniform: every workaround is replaced by the same construct. In this particular case, we gain the additional effect that the rejuvenated code code style is analogous to C style array initialization (compare the definition of the array `a` in the code snippet with the workaround examples).

I.C. Statement of Contribution

The goal of my research is to support the development of real-world applications. This support can be provided by many means and includes improvements to programming languages, provision of specialized libraries, enhancements of programming techniques, experiments with analysis and design techniques, and implementation of development and analysis tools.

The ubiquitous multi-core architectures offer performance benefits to concurrent applications. The development of concurrent software is tedious and involves many challenges that are not present in single-threaded code. The use of non-blocking data structures avoids some of these problems. We — Damian Dechev, Bjarne Stroustrup, and I — have developed a non-blocking implementation of a resizable array (similar to STL vector) (§VII.A.3, [57]). The design, implementation, and test development is joint work together with Damian Dechev. Our initial implementation requires users to follow certain coding guidelines in order to avoid ABA problems [57]. Bjarne Stroustrup suggested that these guidelines can easily be followed when the program code can be partitioned into phases (e.g., production phase and consumption phase of a vector's elements).

Our subsequent work emphasizes the notion of phases to guarantee the safe use of the non-blocking array. Together with Damian Dechev, I worked out a checking mechanism to statically extract array operations from a program phase. The use of the array is safe, if the phase only contains an ABA safe mix of operations. An approach of this idea was implemented by Damian Dechev [56].

Yuriy Solodkyy, Bjarne Stroustrup, and I have extended the C++ programming language with support for open-methods (Chapter IV, [145, 146, 147]). Open-methods provide runtime dispatch where the dispatch decision is based on the dynamic type of one or *more* arguments. Moreover, open-methods separate the definition of classes from the declaration of virtual functions. Under my lead, Yuriy Solodkyy and I (and in an earlier stage together with Nan Zhang) defined the dispatch semantics and extended the Itanium object model. This object model was initially developed for the Itanium architecture. It is used by C++ compilers on various architectures. Yuriy Solodkyy suggested the incorporation of return type information into the ambiguity resolution mechanism. Bjarne Stroustrup suggested an ambiguity resolution policy for programs with dynamically linked libraries. I extended the EDG C++ frontend, code lowering, and built-in functions (such as `dynamic_cast`) to compile open-methods (and other code) according to the modified object model. Together with Yuriy Solodkyy, I implemented the prototype linker.

The design and implementation of open-methods reflects our view, that this language feature is located at the intersection of object-oriented programming (OOP) and generic programming (GP). Subsequently, open-methods proved to be a natural complement of runtime concepts, a programming model developed by Sean Parent and Mat Marcus at Adobe Systems [136, 114]. During an internship at Adobe, I extended their implementation with a mechanism (implemented in ISO C++) that allows for open dispatch, reminiscent to open-methods (§V.A.5, [143]). By replacing the dispatch mechanism with an implementation based on C++ with open-methods, I simplified the design and improved dispatch performance (§V.A.6, [144]).

My work on library design, programming languages and techniques prompted the pertinent question: how can legacy code benefit from current developments? Consequently, my recent work emphasizes on source code rejuvenation [142] — the detection of “older”, lower level code that can be (automatically) replaced with more “modern”, higher level abstractions, such as new language features and libraries. Based on the Pivot, a source-to-source translation framework designed by Dos Reis and Stroustrup, I developed source code rejuvenations tools. TACE, a tool to analyze C++ generic functions, helps programmers understand the requirements their code imposes on template arguments (Chapter VIII). Another tool extends the Pivot with pattern matching. Pattern matching proved useful for the recovery of simple coding patterns from C++ source code. For example, I have applied this tool to recover C++0x’s initializer lists (Chapter VI).

CHAPTER II

LITERATURE REVIEW

In this section, we present prior work in areas that are related to the topic of this dissertation. This includes work on software rejuvenation, re-engineering of high level design abstractions from source code, tools that ease the implementation of source code analysis, and work that relates to the open multi-methods language extension. The literature review is organized as follows: tool support, corpus analysis of software, rejuvenation projects, refactoring, and projects relating to multiple dispatch.

II.A. Tool Support

Source to source translations frameworks have recently received great interest in both research and industry. We present alternative tools that are used to analyze C and C++ source code and tools that help programmers write queries and transformations against source code. In contrast to most tools presented in this section, the Pivot §III.A (the infrastructure we used for this dissertation) uses industrial strength compiler frontends to handle the complete ISO C++ standard and exposes all semantic details to the analysis writer.

Rose [149, 148, 194] is a compiler framework developed by Dan Quinlan's group at the Lawrence Livermore National Laboratory. It provides a source-to-source translation framework for C++ and Fortran programs. Rose's framework offers complete type and control flow information to application programmers. Rose emphasizes the re-generation of code close to its original form. The differences between Rose's C++ representation and the Pivot are subtle. A conversion between the two representations is effortlessly possible. Rose has been applied for a number of domains, such as bug finding (see §II.B), communication trace extraction [148], style guide checking, analysis and transformations of generic programs [194], and others.

Elsa [118, 117] is Scott McPeak's C/C++ parser and intermediate program representation. Elsa is generated by the Elkhound parser generator from a GLR grammar [86]. The internal tree representation is generated by an abstract tree grammar generator. Elsa is able to parse most of GCC 3.4's header files, except for files that rely on template template parameters (e.g., valarray) [117]. Oink enhances Elsa by offering a dataflow framework to analysis writers. OINK is used for finding uses of untrusted data [46]. The Mozilla development team and various research groups at UC Berkeley use OINK for source-to-source translations of C++ code.

The Design Maintenance System (DMS) [23, 24] is a commercial analysis and transformation system which supports multiple input languages. Analysis and transformations tasks are implemented using parallel attribute grammars or in the special purpose language Parlanse. Parlanse supports fine grain parallelism and exhibits characteristics of functional and procedural languages. According to Semantic Design, DMS is very stable and used to evolve multiple large scale projects, to detect clones in source code [24], to reengineer C++ components, to implement test coverage and profiling tools.

Stratego/XT [36] is a system that aims for providing modular and reusable traversal strategies and rewrite rules. At Stratego's core is an integrated functional language and a non-linear pattern matcher. Stratego distinguishes between concrete syntax (source language) and abstract syntax (term representation of the AST). Patterns can be defined in both concrete and abstract syntax. Stratego uses GLR frontends to parse a number of input languages. Due to the lack of type information, the current C++ frontend (Transformers) is not able to resolve all ambiguities currectly. Stratego/XT has been the transformation engine for diverse projects, including program optimizations through domain specific transformations (e.g., CodeBoost [17, 16]), software renovation projects (e.g., COBOL [154]), software configuration, compilation through transformation projects (e.g., The Dryad Java compiler [104]), etc.

Columbus/Can [19, 70] is FrontEndART's commercial C++ analysis framework for source code quality assurance and reverse engineering. Columbus/Can extracts schema representations from C++ source files. The tool is available for Windows and has been applied for several empirical studies in the domains of software metrics, fact extraction (see §II.B), and design pattern recognition in C++ code (see §II.B).

srcML [113, 50] is Maletic et al's tool that adds an XML like structure on top of source code. A major goal of srcML is to provide a representation that is easier to work with than a regular abstract syntax tree, and that retains all syntactic information that is present in source code, including comments, whitespace characters, and macros. As an XML based tool, srcML offers both a document representation, preserving the original formatting of code, and a data representation, exposing many details that are typically present in an AST. Since srcML stops at a specific level of semantic analysis, it is able to represent incomplete and erroneous source code. srcML has been used for program comprehension, recovering class associations and relationships for representation in UML [180], analysis of template code [181], etc.

TXL [52, 53] is language that was specifically developed to facilitate language experiments and transformations. TXL's logic is built around rules, first order functions, and a term rewrite mechanism. TXL tightly integrates traversal, non-linear pattern specification, and term rewriting mechanism. TXL is grammar based, users can extend base grammars with customized rules. Its LL parser handles ambiguities by interpreting alternatives rules in source order and backtracking in case a grammar rule cannot be successfully parsed. Among grammars for other languages, a C++95 grammar is available for download. TXL has been successfully applied in diverse domains — from the transformation of legacy code to avoid Y2K problems in a large scale project to the recognition of hand-written mathematical formulas.

Metal [45] is a language to write static analysis for C programs. Metal's grammar is a superset of C and its programs are called checkers/extensions. Programmers specify patterns for which they can use typed "hole" variables. Hole variables can match any source constructs that are valid in the context. Metal extends the type systems with metal types (e.g.: `any_expr`, `any_scalar`, `any_arguments`, ...). Metal patterns are defined together with a state machine. A matching pattern can trigger a state transition and execute a specified operation. Metal checkers use a depth first traversal of the control flow graph and terminate when all paths have been traversed and a fixed point in the analysis is reached. Among others, Metal has been applied to find imbalances of disabled interrupts and potential null pointer accesses.

Tom [20] extends numerous programming languages (e.g., C/C++, C#, Java, etc.) with a pattern matching and rewrite mechanism for tree structures. Patterns can be defined for built-in types, such as strings, and for user defined structures. Strategies, such as top-down, innermost first, define the order in which rules are applied. With Gom, the Java distribution provides an abstract syntax definition that generates tree structures. TOM has been applied for defining XML queries, source code transformations, and the implementation of rule based systems.

II.B. Software Analysis

Corpus analysis of software has been applied to numerous domains, including optimizations, fact extraction, software design recovery, bug finding, and others. The studied projects have in common that they (statically) recover higher level abstractions or defective conditions from source code. Similar techniques can be applied for re-engineering of language features from workaround techniques.

Algorithm recognition and replacement is Metzger and Wen's approach [121] towards automated, architecture specific program optimization. Their approach matches computational kernels against algorithms specified in a repository. By replacing detected algorithms with calls to software libraries, legacy code can benefit from, for example, highly optimized library implementations (e.g., exploiting parallelism). A major concern of their work is implementation variability. They address this problem by converting programs into an internal canonical form. This process normalizes the represented program by lowering control structures (e.g., loops, conditions), applying compiler optimization techniques (e.g., common subexpression elimination, unreachable code elimination, various loop transformations, constant propagation, etc.), and ordering expression trees (to deal with commutative operators). Algorithm recognition applies pattern matching to compare the canonical representation against the stored instances in the repository. The semantics of nested loops is verified by applying a constraint solver on extracted induction variable constraints. Metzger and Wen used their system to optimize scientific codes on high performance machines.

Design pattern recognition are an integral part of assisted software documentation and code comprehension frameworks. Moreover, Agerbo and Cornils [5] have argued for language extensions that support programming with well known design patterns.

Ferenc et al [70] classify patterns according to the addressed concern. Creational patterns deal with object creation, structural patterns address composition of classes (e.g., inheritance hierarchies), and behavioral patterns describe the interaction of objects. The authors use Columbus/Can to mine design patterns from C++ open source projects (StarOffice Calc, StarOffice Writer, Jikes, and Leda). They report recognizing more than 1200 design patterns in about 3 million lines of code (involving about 14000 classes).

Shi and Olsson [158] present PINOT, a static analysis tool that discovers all

GoF [79] design patterns in the structural and behavior driven category (e.g., visitor, singleton, and flyweight pattern) from Java source code. PINOT uses structural information, such as declarations, associations, and delegations, to limit the search space to likely candidates. PINOT uses limited control flow and data flow analysis to detect pattern invariants and abstract from concrete implementations. PINOT was tested on various industrial frameworks and applications, including Java AWT 1.3, JHotDraw 6.0, Java Swing 1.4, Apache ANT 1.6.

If a design pattern is expressible as language feature, gathering empirical data can be useful for language designers. Muschevici et al [131] studied programs that utilize dynamic dispatch. The article introduces a language independent model for describing multiple dispatch, and defines six metrics on generic functions (i.e. in C++ an open-method family or a virtual function and its overriders) that measure aspects, such as the number of arguments used for dynamic dispatch, the number of overriders in a multi-method family, etc. Using these metrics, the article analyzes nine applications—mostly compilers—written in six different languages: CLOS, Dylan, Cecil, MultiJava, Diesel [43], and Nice [32]. Their results show that 13%-32% of generic functions utilize the dynamic type of a single argument, while 2.7%-6.5% of them utilize the dynamic type of multiple arguments. The remaining 65%-93% generic functions have a single concrete method, and therefore are not considered to use the dynamic types of their arguments. In addition, the study reports that 2%-20% of generic functions had two and 3%-6% had three concrete function implementations. The numbers decreases rapidly for functions with more concrete overriders. Since the multiple dispatch semantics presented in this dissertation is in line with the model defined by Muschevici et al, we expect similar results for C++ with open-methods.

Micro patterns in Java code [81] describes the finding of low level implementation techniques in source code. In contrast to design patterns, micro patterns have a formal definition and can be automatically recognized. Examples for micro patterns include restricted inheritance, emulation of the procedural and functional paradigm in an object-oriented language, or data encapsulation. The goal of this work is the definition of a micro pattern catalog indicative of good coding practices. The paper gives a catalog of 27 patterns. The corpus analysis of 70000 Java classes reveals that a large portion of code (75%) can be described by micro patterns. The authors argue that the maturity of micropatterns makes them candidates for future programming language extensions.

Bug pattern recognition: Hovemeyer and Pugh [92] use simple (formal) methods to discover recurring bugs in code. Their system FindBugs checks Java code for 17 bug patterns (e.g, null pointer dereferences, superfluous null pointer checking, tacitly swallowing exceptions, double checked locking, etc.). They classify their analysis according to the methods applied. The applied methods include querying structural information (i.e., class hierarchies), linear code scan driving a state machine, control flow, and data flow. FindBugs aims at finding potential bugs with a false positive rate less than 50%. They tested their tool against several programs used in production code. On the tested software the tool reports between 1 and 3 bugs per thousand lines of code.

Quinlan et al [150] model bug patterns recognizable by structural and control flow data. In their studies, they compare bug pattern specifications based on binary decision diagrams (BDD) [99] and Datalog [39] with implementations written directly against Rose's AST. The Datalog implementation requires storing the intermediate program representation into a relational database. Then they employ `BDDBDDB` [107] as query solver. The authors implemented the following bug pattern analysis: `switch` statements without `default` branch, static member construction (posing portability hazards), and null pointer dereferences. Their research reports implementations against the AST to feel more "natural", while the declarative nature of Datalog cuts the length of implementation significantly. Conversely, the paper reports that direct approach shows good runtime performance, while the execution time of the Datalog approach is not yet well understood.

Martin et al's program query language (PQL) [115] uses a mix of static analysis and instrumentation to find bug conditions in software. Queries define patterns in a source code like style. Patterns can be combined using various operators, including operators for sequences, partial sequences, alternatives, and others. A static analyzer reports code that violates patterns. For cases where the static analyzer cannot determine a definitive result, the code is instrumented to perform the analysis at run time. The static analysis engine offers sophisticated points-to information. The implementation of the static analyzer uses `BDDBDDB` and Datalog. The authors utilized PQL to detect bug patterns, such as the violation of dynamic type constraints, security flaws allowing SQL injection attacks, call sequences violating required order of invocations, and lapsed listeners (leading to memory leaks). The authors tested their approach against six industrial projects (i.e., Eclipse, webgoat, personalblog, road2hibernate, snipsnap, and roller) and found more than 200 formerly unreported errors (the majority in Eclipse).

Bessey et al [29] describe the experience of turning an academic project into a commercial bug finding tool (Coverity). The paper provides anecdotal evidence of the challenges a static analysis tool faces when applied in industrial settings. A practical tool needs to be able to handle about 1400 LOCS per minute in order to enable nightly checks against larger code sizes. The acceptance of such a tool depends on its ability to correctly recognize important bugs and at the same time report only a limited number of false positives. Technical, organizational, and individual reasons cause additional challenges in the real world. Technical problems are posed by the plethora of systems, build processes, language dialects, and use of dated tools. Organizational problems are related to intellectual property issues (code causing problems cannot be shared with the tool developer) and misaligned incentive structures (rewards are based on bug counts, hampering the introduction of new tool versions that could identify more bugs; rewards are given for successful builds or meeting milestones on time, but not for correct code). Individual reasons deal with tool acceptance by programmers that do not understand the bug reports. Such programmers are more likely to misclassify a reported problem that is based on complex analysis as false positive.

Thompson and Brat [184] utilize a mixed set of tools for software analysis. Their tools include a static analyzer, symbolic execution engine, and a model checker. Each tool in their tool chain is built on top of the LLVM compiler framework [110]. LLVM generates low-level (e.g., templates are instantiated, inheritance hierarchies are flattened, overloaded function calls are resolved, etc.), but typed intermediary code for its own virtual machine. By integrating multiple approaches, Thompson and Brat combine the advantages of each approach. Their toolchain uses static analysis to indicate regions that the model checker will need to check, thereby reducing the effective state space (partial order reduction). In addition, the model checker provides more specific results for alias analysis, which can be fed back to the static analysis tool. Failure conditions that the model checker can find are modeled as assertions. The symbolic computation engine drives the generation of interesting test cases, that can be fed into the static analyzer and the model checker. Preliminary tests with NASA's On-board Abort Execution (OAE) indicate that the framework is able to handle real-world flight code.

Venet [188] discusses abstract interpretation as a technique for static analysis and formal verification. The key element in Venet's work is the decomposition of analysis into modular entities. For each module, the interface and the environment are formally defined. The author uses a domain specific language to model interaction between modules and their environment. This model together with the module code is fed into a static analyzer. The framework was applied to check a dynamic buffer module, which is an integral part of all OpenSSH components, against buffer overflow problems. The test verified that all pointer indirections within the buffer module are safe.

Loop classification [78] uses XQuery [40] to recognize monotone loops in Java software. A monotone loop is defined as a loop where the iteration variable is guaranteed to progress towards the limit of the termination condition in every iteration of the loop. Loops where the analyzer cannot guarantee the monotonicity criterion are reported for human inspection (or forwarded to more sophisticated analysis tools). With relatively simple formal methods, the authors show that in practical applications about 80% of loops terminate.

II.C. Rejuvenation

The MoDisco project [67] is part of the Eclipse framework. MoDisco aims at providing a framework that extracts information from existing programs, synthesizes this information, and derives upgrade information. MoDisco is a model driven framework that relies on meta-models of existing software, discoverers that can create these models from systems, and tool support that help understand and transform complex models.

Generifying Java is described by Von Dincklage and Diwan [60]. They present a tool Ilwith, a system for the Java programming language that identifies classes that could be rewritten to take advantage of Java Generics (introduced in Java 1.5). Ilwith is based on jimple, the Soot's optimization framework's [185] high level program representation. Ilwith operates in two phases, first identifying parametrizable classes and their type constraints, and second converting the identified classes and their clients. The tool uses a number of heuristics in order not to "overgenerify" (as compared to human code maintenance). Ilwith was tested against data structures from the Java 1.4 and Antlr [139] class libraries. The authors report their tool effectively handles the tested libraries, but classes that incorrectly use weak typing.

Extracting Concepts from C++ template functions has been studied by Sutton and Maletic [181]. Their work describes matching requirements against a set of predefined concepts. The approach is based on formal concept analysis and finds combinations of multiple concepts, if needed, to cover the concept requirements of a template functions. In order to avoid complications from "the reality of C++ programming" [181], the authors validate their approach with a number of hand implemented STL algorithms, for which they obtain results that closely match the C++ standard specification. The authors discuss how small changes in the implementation can lead to small variations in their identified concepts.

Leveraging paradigm shifts in legacy code requires automated source code rejuvenation tools. Siff [162, 164, 163] discusses source code renovation techniques in the context of migrating source code from C to C++. Techniques to recover object-oriented polymorphism include analysis of casts and structural subtyping relationships in C programs. The author tested the approach against several projects (i.e., bash, binutils, gcc, ijpeg, perl vortex, xemacs, phone). The work reports the number of total casts, pointer casts, upcasts, downcasts, and mismatched casts (neither up or down, for example, from `void*`). For bash, which had 76 000 LOC, the authors reports 309 type pair-specific casts, of which were 65 type pair-specific pointer casts. The pointer casts were composed of 50 up- and four downcasts, and eleven mismatches. Another renovation discusses the conversion of regular C functions into C++ template functions. Siff starts by replacing type annotations of variable and parameter declarations with free type variables. Then, he applies type inference methods and predefined functional constraints to solve type relationships among type variables. In order to not over generalize, the author applies heuristics to find types that will not be templatized. The reported results indicate that for the tested code about half of the concrete types can be replaced by template arguments. Finally, Siff also discusses the problem of modularizing source code. His work utilizes concept analysis [106]; functions are modeled as objects, and the author describes analysis with various attribute models (e.g., a function uses a specific field of a struct, a function has a parameter of a specific type). The dissertation presents an algorithm that finds possible concept partitions. The tool depends on user input to limit solutions to partitions with specific properties (e.g., granularity).

Harsu [88] presents cross language (from PL/M to C++) code modernizations. She discusses finding abstractions of object oriented code in a lower level language. The key element is programming idioms that mimic language features for modeling subclass relationships and object oriented polymorphism. In the course of migrating the code to C++ these idioms are recovered and replaced by an object-oriented implementation. The author provides a tool that can identify code in procedural languages that can better be modeled using object oriented concepts. For example, the tool identifies selection statements that can be replaced by virtual functions. She tested the approach against real world code from the telecommunication sector and found six procedures (out of eleven) where late binding improves software design. Harsu also addresses some problems emerging from conditional compilation.

Replacing the C preprocessor: McCloskey and Brewer [116] analyze the practical use of the C preprocessor. They discern important use cases and use cases that are error prone. Consequently, the authors define ASTEC, a replacement for the C preprocessor, supporting the important use cases. ASTEC is simpler to analyze, in particular for source to source transformation software. The authors also offer an automated translation tool (Macroscope) that derives its information from expanded macro in a concrete AST of the primary source language (e.g., C, C++). For complex and irregular preprocessor macros, Macroscope allows programmers to specify hints. Macroscope generates warnings for translations that are considered imprecise (potentially due to an error in the original macro specification). The authors tested their tools against four stable open source products (i.e., gzip 1.2.4, rcs 5.7, OpenSSH 3.9p1, Linux 2.6.10). The tests did not reveal any errors in the tested code base.

II.D. Refactoring

The term refactoring is derived from the mathematical term "factoring" and refers to finding multiple occurrences of similar code and factoring it into a single reusable function, thereby simplifying code comprehension and future maintenance tasks [119]. The meaning of refactoring has evolved and broadened. Opdyke and Johnson [134] define refactoring as an automatic and behavior preserving code transformations that improves source code that was subject to gradual structural deterioration over its life-time. Essentially, refactorings improve the design of existing code [73, 105].

Traditionally, refactoring techniques have been applied in the context of object-oriented software development. Automated refactoring simplifies modifications of a class, a class hierarchy, or several interacting classes [134]. More recently, refactoring techniques have been developed to support programs written in other programming styles (i.e., functional programming [108]).

Refactorings capture maintenance tasks that occur repeatedly. Opdyke [133] studied recurring design changes (e.g., component extraction and class (interface) unification). Refactoring is a computer assisted process that guarantees correctness, thereby enabling programmers to maintain and develop software more efficiently. In particular, evolutionary (or agile) software development methodologies [2], where rewriting and restructuring source code frequently is an inherent part of the development process of feature extensions, benefit from refactoring tools.

"Anti-patterns" [37] and "code smells" [73] are indicators of design deficiencies. Anti-patterns are initially structured solutions that turn out to be more troublesome than anticipated. Examples for anti-patterns include the use of exception-handling for normal control-flow transfer, ignoring exceptions and errors, magic strings, and classes that require their client-interaction to occur in a particular sequence. Source code that is considered structurally inadequate is said to suffer from code smell. Examples for "code smell" include repeated similar code, long and confusing functions (or methods), overuse of type tests and type casts. The detection of code smell can be partially automated [138] and it assists programmers in finding potentially troublesome source locations. Refactoring of anti-patterns and "code smells" to more structured solutions improves safety and maintainability.

Refactoring does not emphasize a particular goal or direction of source code modification — e.g., refactoring supports class generalization and class specification [133], refactoring can reorganize source code towards patterns and away from patterns (in case a pattern is unsuitable) [105].

Refactoring strictly preserves the observable behavior of the program. The term "observable behavior", however, is not well defined [119]. What observable behavior exactly requires (e.g., function call trace, performance, ...) remains unclear. Refactoring does not eliminate bugs, but can make bugs easier to spot and fix.

II.D.1. Source Code Rejuvenation Versus Refactoring

Table I summarizes characteristics of source code rejuvenation and refactoring. Both are examples of source code analysis and transformations that operate on the source level of applications. Refactoring is concerned to support software development with tools that simplify routine tasks, while source code rejuvenation is concerned with a one-time software migration. Both are examples of source code analysis and transformation. Source code rejuvenation gathers information that might be dispersed in the source of involved workaround techniques and makes the information explicit to compilers and programmers. Refactoring emphasizes the preservation of behavior, while source code rejuvenation allows for and encourages behavior improving modifications.

Table I. Source code rejuvenation vs. refactoring

	Code Rejuvenation	**Refactoring**
Transformation	Source-to-source	Source-to-source
Behavior	improving	preserving
Directed	Raises the level of abstraction	No
Drivers	Language and library evolution	Feature extensions Design changes
Indicators	Workaround idioms and techniques	Code smells Anti-patterns
Applications	One-time source code migration	Recurring maintenance tasks

We might consider code rejuvenation a "subspecies" of refactoring (or vise versa), but that would miss an important point. The driving motivation or code rejuvenation is language and library evolution rather than the gradual improvement of design within a program. Once a rejuvenation tool has been configured, it can be applied to a wide range of programs with no other similarities than that they were written in an earlier language dialect or style.

We are aware that refactoring has been used to describe semantic preserving code transformations that migrate code to use new frameworks (e.g., Tip et al. [18], Tansey and Tilevich [182], and others). The difference between language evolution related code transformations and refactoring is subtle but important. We prefer and suggest the term “source code rejuvenation” for describing one-time and directed source code transformations that discover and eliminate outdated workaround techniques, idioms, and uses of old libraries. The key benefit of source code rejuvenation is that it raises the level of abstraction in source code. Subsequently, more (higher-level) information is available to code maintainers, static analysis tools, and compilers.

II.E. Multiple-Dispatch Extensions

Programming languages can support multi-methods either through built-in facilities, pre-processors, or library extensions. Tighter language integration enjoys a much broader design space for type checking, ambiguity handling, and optimizations compared to libraries. In this section, we will first review both library and non-library approaches for C++ and then give a brief overview of multi-methods in other languages.

Preprocessor based implementations: A preprocessor reads some source code and transforms it into source code that can be read by the next stage in the build process. A general disadvantage of providing language extensions through preprocessors is that the build process becomes fractured and hard to analyze. The preprocessor transforms specific parts of a program, but has typically incomplete semantic information of the source. Likewise, the resulting code lacks the high level abstraction. In the long run, preprocessor solutions tend to become hard to maintain.

Cmm [166] is a preprocessor based implementation for an open-method C++ extension. Cmm takes a translation unit and generates C++ dispatch code from it. It is available in two versions. One uses RTTI to recover the dynamic type of objects to identify the best overrider. The other achieves constant time dispatch by relying on a virtual function overridden in each class. Dispatch ambiguities are signaled by throwing runtime exceptions. Cmm allows dynamically linked libraries to register and unregister their open-methods at load and unload time. In addition to open-method dispatch, Cmm also provides call-site virtual dispatch. Call site virtual dispatch delays the binding to regular overloaded functions, if one of their actual arguments is preceded by the `virtual` keyword.

```
void foo(A&);
void foo(B&); // B derives from A
// call site virtual dispatch
foo(virtual x); // which foo gets invoked depends on the dynamic type of x
```

Cmm does not provide special support for repeated inheritance, and therefore its dispatch technique does not entirely conform to virtual function semantics.

DoubleCpp [30] is another preprocessor based approach for multi-methods dispatching on two virtual parameters. It essentially translates these multi-methods into the visitor pattern. For doing so, DoubleCpp requires access to the files containing the class definitions in order to add the appropriate accept and visit methods. DoubleCpp, unlike other visitor-based approaches, reports potential ambiguities.

Libraries can provide provide multi-method abstractions when the host language lacks direct language support (e.g, Loki [9] for C++, or Python [186]). In general, libraries providing multi-methods, such as Loki, or the runtime concept library §V.A, lack semantic information of the class hierarchy that is available to compilers. Consequently, users have to provide this information in form of type lists, intrusive modeling techniques, or initialization to the library implementation. This extra effort makes libraries prone to programming errors. The lack of semantic information can also result in less than optimal performance §V.A.7.

Loki [9], based on Alexandrescu's template programming library with the same name, provides several different dispatchers that balance between speed, flexibility, and code verbosity. Currently, it supports multi-methods with two arguments only, except for the constant-time dispatcher that allows more arguments. The static dispatcher provides call resolution based on overload resolution rules, but requires manual linearization of the class hierarchy in order to uncover the most derived type of an object first. All other dispatchers do not consider hierarchical relations and effectively require explicit resolution of all possible cases.

Related language extension: Accessory functions [71, 195] allow open-method dispatch based on a single virtual argument and discuss ideas to extend the mechanism for multiple dispatch. The compilation model they describe uses, like our approach, a compiler and linker cooperation to perform ambiguity resolution and dispatch-table generation. However, the accessory functions are integrated into the regular v-tables of their receiver types, which requires the linker to not only generate the dispatch table but also to recompute and resolve the v-table index of any other virtual member function. Neither paper provides a detailed discussion of the intricacies when multiple inheritance is involved. The authors do not refer to a model implementation to which we could compare our approach.

OOLANG [135] is a language developed for the Apemille SPMD supercomputer. OOLANG features a C++ like object model and supports open-methods. The work gives special attention to the handling of covariant return types. OOLANG's system differs from our implementation in the handling of repeated inheritance. Classes that repeatedly inherit from a baseclass must define an overrider for each open-method that uses the baseclass as the type for a virtual parameter. Furthermore OOLANG does not use covariant return type information for ambiguity resolution.

Alternative implementation technique: In §IV.F, we present a dispatch table based implementation of multi-methods. Chambers and Chen [44] generalize multiple dispatch towards predicate-based dispatch. Their paper presents a lookup DAG based implementation of multiple dispatch. Their approach trades off smaller binary size for somewhat less efficient dispatch.

CHAPTER III

TOOL SUPPORT

It is crucial for the acceptance of source code rejuvenation software that the results are accurate and can be produced in a timely manner. Consequently, such software requires a source-to-source transformation framework that is general, can represent the complete source language, and can handle real-world code. In addition, the development of rejuvenation tools should allow programmers to write their analysis with an effort that is proportional to the analysis task. The graph in Fig. 1 sketches the architecture of our rejuvenation tool chain.

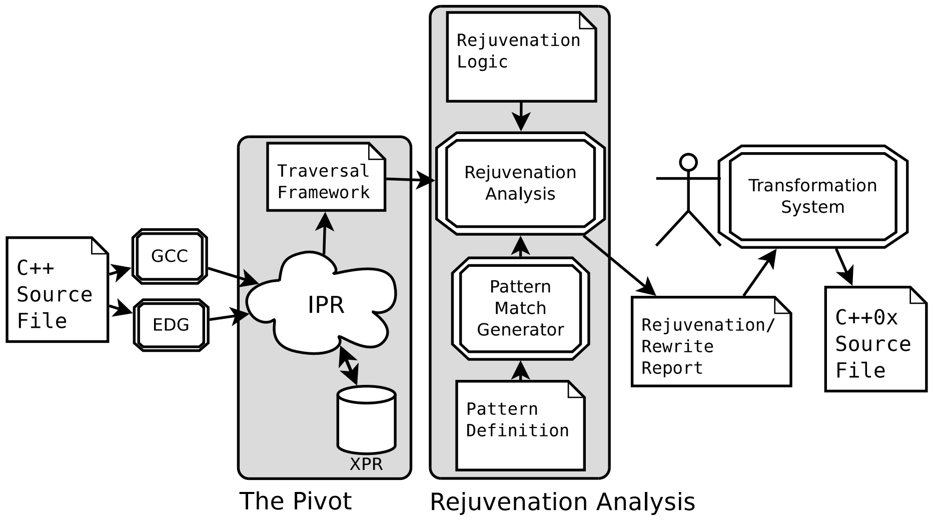

Fig. 1. Rejuvenation tool architecture

Our tool chain is based on the Pivot [177, 64], a compiler-independent platform for static analysis and semantics-based transformation of the complete ISO C++ programming language and some advanced language features included in C++0x. The Pivot generates an in-memory representation from C++ source files. The rejuvenation analysis traverses this intermediate representation and searches for interesting patterns. These patterns are defined in a C++ like language. A generator converts these definitions into C++ code that performs the matching. The rejuvenation logic determines whether the combination of recognized patterns indicate some workaround code. The analysis prints out a rejuvenation (or rewrite) report. In order to migrate the code automatically the report can be fed into a rewrite system. It is desirable to generalize the rejuvenation logic to recognize also code that resembles workaround patterns, but where the semantic equality cannot be guaranteed (e.g., if some code uses virtual function calls). Such cases require human inspection before the code can be rewritten.

The Pivot framework offers the following components. §III.A describes the core component which consists of the compiler frontend, the internal representation (IPR) and the external representation (XPR). The support libraries (§III.B) help programmers with common tasks that are not directly expressible with the core component. This includes the provision of meta-information, a class that allows writing template code against the IPR nodes, support to traverse the abstract syntax tree (AST), and a library that compares nodes for equality. §III.C describes a tool, that generates query code based on a pattern specification.

Using the framework reduces the amount of code programmers write by hand. For example, the traversal library factors 600 lines of common code; in our initializer list recovery project (see Chapter VI), the use of the pattern matcher reduces the size of code by a factor of 10.

III.A. Pivot

The Pivot, designed by Dos Reis and Stroustrup, is a static analysis and transformation framework for C++ software. It is designed for generality and emphasizes the retention of original source code. The Pivot preserves high-level abstractions that are present in source code, such as class hierarchies, type information, and template code.

The Pivot utilizes industrial strength compiler frontends (i.e., EDG and GCC) to parse C++ source code. The source code is converted into IPR. IPR makes semantic relations that are implicitly part of source codes (e.g., names) explicit. For example, code can have multiple entries with the same name (e.g, variables functions, classes, etc.). IPR resolves these ambiguities and generates separate objects for different source code entities. These IPR objects contain information, such as type and scope data, which simplifies the writing of source code analysis. The provision of semantic data distinguishes industrial-strength frontend based tools from simpler parser and text based tools.

In §II.A, we have presented some alternative tools but only few of them can handle the full ISO C++ standard, offer information on the same level of detail, or provide an easy to use intermediate representation. The Pivot also features an external program representation that can be used for language experiments or as persistent storage format in between two analysis runs.

III.A.1. Internal Program Representation

IPR represents C++ programs at a high-level that preserves many details present in source code. IPR is not a special purpose representation but rather aims at generality, completeness, and simplicity. The information retained in IPR is suitable for a wide variety of use cases.

IPR uses an object-oriented class hierarchy for representing C++ source code. It can represent a superset of C++, including some programs using language features of the next revision of the ISO standard. It is extensible and currently supports a number of C++0x's language extensions (e.g., decltype, for range loops, ...). IPR is a fully typed intermediate representation of C++ source code. It represents uninstantiated, specialized, and instantiated templates. Calls to overloaded functions (in non-template and instantiated code) are resolved in IPR. The level of these details makes the development of industrial-strength tools difficult [29]. Hence, the Pivot relies on industrial strength C++ frontends (i.e., EDG's C++ frontend [69], GCC [76]).

IPR's preservation of high level information (e.g., comments, uninstantiated templates, ...) enables analysis of template based libraries, for example, the deduction of concepts from templates or the rejuvenation and regeneration of source code. IPR's representation eliminates C++'s syntactic irregularities. For example, there is only one kind of object initialization (as opposed to using constructor or assignment syntax in a declaration) and the receiver argument of a member function call becomes part of the regular argument list. IPR does not require the source code to be error free. It can represent non-compilable programs as long as the source code is syntactically correct.

We stress that the IPR is fully typed. Type information is necessary for the implementation of source code rejuvenations that are type sensitive. Interesting rejuvenation analyses and transformations depend on type information. For example, concept extraction (Chapter VIII) distinguishes operations that are template argument dependent from operations that are not. Likewise, the implementation to migrate source code to use initializer lists (Chapter VI) is limited to container types that support initializer-list constructors (i.e., STL containers).

III.A.1.a. Implementation

The design of IPR's class hierarchy cleanly separates non-modifiable interface classes from implementation classes. Since this dissertation treats corpus analysis of software, this section focuses on presenting the AST interface. Each node type is part of one out of six categories: declarations (derived from `Decl`), statements (`Stmt`), expressions (`Expr`), names (`Name`), types (`Type`), and other information, such as Linkage, Annotations, Comments (`Node`). In total, IPR has about 160 interface classes. (With "interface class" we mean a type that is recoverable with IPR's visitor.) Fig. 2 gives an overview of IPR's class hierarchy. The various categories are drawn with different shades.

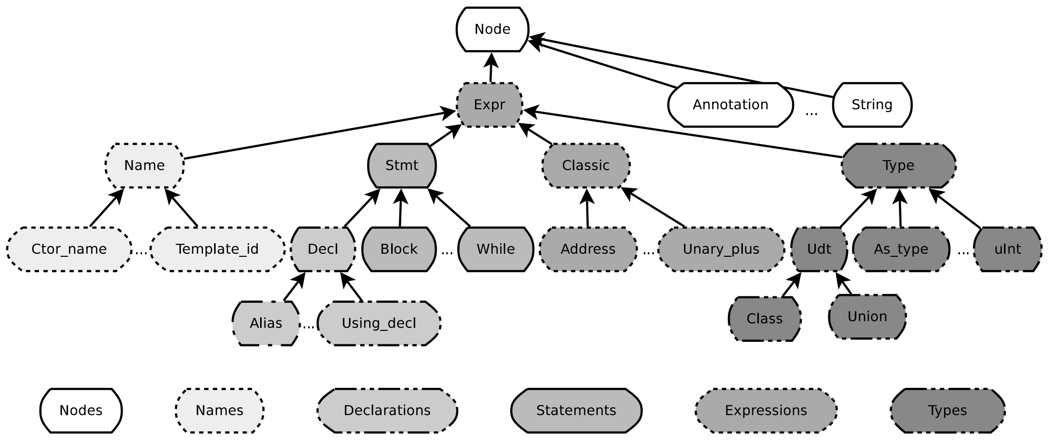

Fig. 2. IPR class hierarchy

The design and implementation of IPR uses template classes extensively. For example, any class with a single major subtree is derived from the template class `Unary`.

```
template<class Cat, class Operand = Expr&>
struct Unary : Cat
{
  typedef Operand Arg_type;
  virtual Operand& operand() = 0;
};
```

The two template arguments describe the category of the node (the template argument `Cat`) and the node type of the indirection `Operand`. The category describes the group to which a node class belongs. The category is a base class that factors common data access functions. The operand defines the type of the child. Examples for node-types that derive from `Unary` include all unary operators (e.g., the complement operator `~`, the negation operator `!`, ...) referring to `Expr`, comments and identifiers referring to `String`, expression statements referring to `Stmt`, and others. The following code shows two classes, `Comment` and `Not`, that derive from the unary template, and one class `Classic`, which is a base class of `Not`.

```
// class subsuming expression operators
struct Classic : Expr {
  virtual bool has_impl_decl() const = 0;
  virtual const ipr::Decl& impl_decl() const = 0;
  // ...
};
// class, representing the ! − operation
struct Not : Unary<Category<not_cat, Classic> > {};
// class representing comments in code
struct Comment : Unary<Category<comment_cat, Node>, const String&>
{
  Arg_type text() const { return operand(); }
};
```

The class `Not` inherits from `Unary`. `Unary` inherits from its first template argument (i.e., `Category<not_cat, Classic>`). In turn, the Category class derives from its second template argument — in this case `Classic`. `Classic` subsumes all expression operators and contains functions to test whether an expression resolves to a user implemented function. `Unary`'s operand defaults to `const Expr&`.

The class `Comment` inherits from the template class `Unary`, which in turn derives from the `Category` class for comments. The latter derives from IPR's root class `Node`. `Unary`'s, and subsequently `Comment`'s operand is a `const String&`.

Similar to `Unary`, IPR offers templates for binary, ternary, and quaternary classes. The template from which a specific interface class derives is a design choice that reflects the number of major subtrees. This means that cast operations, which take a `Type` and an `Expr`, inherit from `Binary` and not from `Unary` — from an evaluation point of view casts could be viewed as unary operators, because casts evaluate a single sub expression and convert the result to a specified type.

Unification: Representing large-scale industrial-size software projects in memory can be challenging. IPR addresses scalability by using a unified representation (i.e., the same object) for semantically equal data. This reduces the amount of memory needed. Experiments with GCC demonstrate that unification of type nodes save about 60% of memory usage for named types, 17% for pointers, and about 10% for cv-qualified types [64]. Which IPR nodes are unified is a design choice that can be easily altered. Currently, type and name classes use node unification.

Unification matters for the implementation of an efficient equality comparison. For unified nodes, pointer equality is sufficient, while non-unified nodes require recursive subtree comparison.

Querying the AST: The built-in querying mechanism for information retrieval from an AST relies on the visitor design pattern [79]. A detailed comparison of the visitor pattern to open-methods in the context of AST queries is provided in §V.B.2. Higher level techniques utilize the provided visitor as the base mechanism. We have experimented with a template library [51] that query implementations. The library instantiates visitors (using `is<...>`). AST objects can be queried by using functors whose name correspond to a class's member functions (e.g., `name`, `type`). For sequence traversal the library offers `has<>`. The following code snippet shows a simple example that queries whether a node represents a variable or field (data member) called "i".

```
is<ipr::Var, ipr::Field>
  ( name(is<ipr::Identifier>(stringobj == "i")), /* for Var */
    name(is<ipr::Identifier>(stringobj == "i")) /* for Field */
  )(node);
```

For each indirection, programmers specify the edge to the child node (i.e., `name`, `stringobj`) and which type to expect (i.e., `is<>` functions). The code snippet above queries, whether a node is a variable (`is<ipr::Var>`) whose name (`name`) points to an identifier (`is<ipr::Identifier>`) with the value "i" (`stringobj == "i"`).

Luke Wagner [192] builds an iterator based traversal and query framework on top of the visitor pattern. In addition, his framework uses a lower level, semantic-oriented representation that abstracts from the syntactic source code level of IPR.

III.A.2. eXternal Program Representation

XPR is a persistent and human readable (and also writable) representation. Since XPR preserves the information present in IPR, XPR can be considered as the external twin of IPR. XPR is suitable for storing temporary results between runs of different analyses.

The file size of an XPR file is in the same order of magnitude as the original source file.

The grammar of XPR is similar to C++. Some grammar rules are modified to simplify ambiguity resolution at parse time of code. Specifically, XPR uses a different syntax for declarations, types, and templates instantiations. The following snippet demonstrates the syntax modifications with a small example.

```
// in C++
template <class T>
class vector {
  T* p;
  const static int sz = 0;
};
// in XPR
vector : <T: class> class {
  p : *T;
  sz : const static int = 0;
}
```

The essential differences are that declarations start with a name (i.e., `vector`, `p`, `sz`) which is followed by a type (e.g., `<T: class> class`, `*T`) and optionally by an initializer (e.g., the body of the class, or the value `0`). Types are consistently spelled from left to right (e.g., `*T`).

XPR is based on an LALR(1) grammar, which accelerates parsing compared to parsing C++ source files. The Pivot offers Flex and Bison files, that generate an XPR lexer and parser. Users can utilize these tools for their own needs, for example to experiment with new language constructs. In this dissertation, we use an extension of the XPR as the input format for a pattern match generator (§III.C).

III.B. Support Libraries

This section presents libraries that factor common code, such as traversing of the AST, handling groups of nodes, recovering type information, and comparing nodes for equality.

To traverse the internal program representation, the Pivot provides the visitor pattern [79] as a standardized interface. In ISO C++, the essential value of the visitor pattern is the recovery of a node's dynamic type information without resorting to (cascaded) type tests that rely on runtime type information (RTTI).

In order to simplify writing traversal and query code for IPR, we factor common tasks into a reusable framework. One goal of this framework is to enable programmers to write generic AST queries. By generic, we mean that programmers can write template functions that handle groups of nodes. The presented framework offers functionality to select specific groups, such as all unary expressions, all cast expressions, etc. In contrast to object oriented polymorphism, template code preserves (recovered) type information.

Applications: We will utilize the presented techniques to develop a library of common AST traversal functions §III.B.3. Programmers use these libraries to traverse the AST, for example to embed the code that is produced by the pattern match generator (§III.C). In addition, the pattern matching code will rely on some libraries (i.e., equality generic visitor §III.B.2, comparison §III.B.4).

III.B.1. Compile-Time Meta Information

Querying nodes for compile time information is useful for programmers. It is even more useful in generic code, that has to account for type specific differences. Consider code that tests whether two nodes are equal:

```
// the equality test depends on properties of T
template <class IprNode>
bool equal(const IprNode& lhs, const IprNode& rhs);
```

The efficient implementation of `equal` depends on the specific type `IprNode`. If the type variable `IprNode` refers to a leaf class representing a built-in type no further information is needed — the two nodes are equal. Unified classes can be simply compared by testing whether the two objects (`lhs` and `rhs`) have the same address. Other nodes require subtree comparison.

The libraries presented in this section supply compile-time information of IPR types, such as meta data on the inheritance structure, concept membership, node comparison, and node arity. The `meta` library defines a compile time function (i.e., template class) `ct` that supports querying of this information. `ct` is defined over IPR's hierarchy.

```
// namespace ez::meta
template <class IprNode>
struct ct {
  ...
};
```

Some of this meta data is application specific. Other domains might require different design or implementation choices. The lack of generality is a reason for not embedding this information directly in the IPR classes. For example, are `cast` operations, as discussed in §III.A.1.a, unary or binary? While some applications consider `cast` operations as binary (they have an expression and a type), other applications that are interested in evaluating subexpressions consider `casts` as unary. Another question is, when are two `Id_expr` equal? `Id_expr` nodes wrap uses of named expressions (e.g., variables, parameters). Are two `Id_expr` nodes equal if they refer to the same declaration, or are they equal if the name of the declaration, as written in the expression, is equal?

Hierarchy: Scenarios exist, in which we would like to forward the handling of a particular node kind to a base case. `ct` supports hierarchical forwarding by providing the base type as typedef (i.e., `parent`).

```
  // member of ct<T>
  typedef /* ... */ parent;
```

The function `base_type` utilizes this compile time function to erase a single level of type information in the IPR hierarchy. For example, invoking `base_type` with a reference to a variable (i.e., an object of type `ipr:Var`) returns a reference to the same object but with one level of type information removed (i.e., a reference to `ipr::Decl`).

```
// namespace ez::meta
template<class IprNode>
const typename ct<IprNode>::parent&
base_type(const IprNode&);
```

Node forwarding is frequently applied in the code generated by the pattern matcher. For example, if a unary operation (e.g., `Unary_plus`) matches a pattern, but its operand does not, the node will be forwarded to code for the base class (e.g., `Expr`).

```
// n is a UnaryPlus
// match is a boolean indicating whether n's subexpression is a match
if (!match) this->handle(ez::meta::base_type(n));
```

Equality comparison: IPR addresses scalability issues by unifying some node categories. Equality comparisons benefit from node unification, because a simple comparison of two objects' addresses replaces subtree comparison. Moreover, the Pivot has separate IPR classes for C++ built-in types. The test whether a given node represents a built-in type can be implemented by a single type recovery (i.e., double dispatch to a visitor). Some declarations, such as functions, variables, user defined types, etc., can be (multiple times) forward declared. The question arises what an equality comparison of two different declarations of the same source code entity should return. The compile-time meta information library cannot presume to know the answer. The meta class `ct` provides a member `comparison_tag` that indicates how two nodes of this type can be compared efficiently.

```
  // member of ct<T>
  typedef /* ... */ comparison_tag;
```

Fig. 3 shows the comparison tag hierarchy. Arrows point from the derived to the base class.

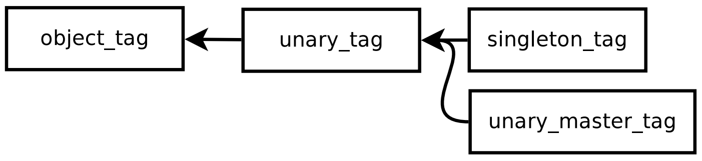

Fig. 3. Comparison tag hierarchy

In addition to the member in `ct`, the library provides a convenience function `comp_tag` that returns the same tag type.

```
// namespace ez::meta
template <class IprNode>
typename ct<IprNode>::comparison_tag
ez::meta::comp_tag (const IprNode &);
```

The following code snippet uses the comparison tag for the implementation of an equality comparison function. We sketch the implementation of equality comparison, which is further discussed in §III.B.4.

```
template <class IprNode>
bool _equals(const IprNode& lhs, const IprNode& rhs, singleton_tag)
{
  return true;
}
// internal functions that handle different object classes
template <class IprNode>
bool _equals(const IprNode& lhs, const IprNode& rhs, unified_tag)
{
  return &lhs == &rhs;
}
template <class IprNode>
bool _equals(const IprNode& lhs, const IprNode& rhs, object_tag);
template <class IprNode>
bool _equals(const IprNode& lhs, const IprNode& rhs, unified_master_tag)
{
  return &lhs.master() == &rhs.master();
}
// callable function, that invokes a suitable helper function
// _equals according to the comparison tag.
template <class IprNode>
bool equals(const IprNode& lhs, const IprNode& rhs)
{
  return _equals(lhs, rhs, ez::meta::comp_tag(lhs));
}
```

The main function `equals` uses the comparison tag to choose from four different implementations (overloaded function `_equal`). The first implements equality for singleton nodes, such as built in types and global scope. The second implementation is for unified nodes and requires pointer equality. The third implements equality by comparing subtrees. — the implementation is not shown. The fourth compares declarations that can have several source code entries (e.g., several function declarations for the same definition). The comparison checks whether the two declaration point to the same master declaration.

Arity tags: In IPR, the number of major subtrees a class depends on is visible in the derivation of the class hierarchy (e.g., unary expressions inherit from the Unary template). Programmers can write generic code against the base templates that IPR provides by overloading template functions. For example, to write code for all unary expressions, programmers could provide the following function template:

```
template <class Cat, class T>
void handle_unary(const Unary<Category<Cat, Classic>, T>&);
```

This style makes generic function declarations verbose. In addition, the type `Unary<...>` is a concrete intermediate type but not a fully derived IPR class. Consequently, any functions called from `handle_unary` cannot use fully derived parameter type names (e.g., `Address`, `Unary_minus`). To overcome these problems, the presented library offers arity tags that can be used as an additional parameter for tag dispatching [1]. Using arity tags, we can append the previous function declaration with a tag parameter. The `unary_tag` guards the function against being called with other nodes, such as binary expressions.

```
template <class IprNode>
void handle_unary(const IprNode&, ez::meta:unary_tag);
```

The arity tags defined in the library closely match IPR's category structure. Their inheritance graph is shown in Fig. 4.

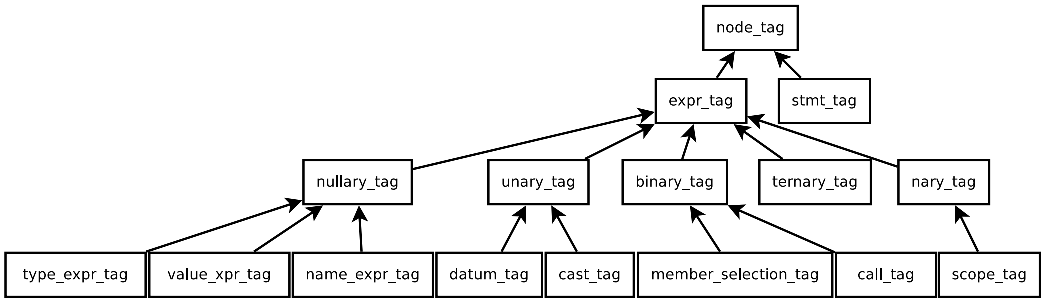

Fig. 4. Arity tag hierarchy

Programmers can generate the arity tags by using the function `ez::meta::tag`. The function is defined over all IPR classes. The following code snippet demonstrates the use of arity tags.

```
// empty function for non unary nodes
// complements the preceding declaration
void handle_unary(const ipr::Expr&, ez::meta::node_tag) {}
template <class IprExpr>
void handle_unary(const IprExpr& expr)
{
  // forward handling to specific implementations
  handle_unary(expr, ez::meta::tag(expr));
}
```

III.B.2. A Generic Visitor

This section describes a generic visitor. This class enables templated visit functions that operate on various node types, while retaining full type information; e.g., a single implementation for all unary expression nodes.

IPR offers the visitor pattern for traversing the AST node. The visitor pattern declares a virtual function `visit` for each interesting AST node type. Programmers can override these virtual functions with specific logic. IPR's class hierarchy allows the processing of groups of nodes, by forwarding a call to the base implementation of a specific node type (e.g., `Address` calls the visit for `Expr`). Forwarding erases concrete type information and is more general than it should be (e.g., the visitor class has no `visit` function for all unary nodes.)

This framework provides a template class (`GenericVisitor`) that remedies the loss of type information by recasting node visitation into the generic domain. User defined visitors derive from the generic visitor using the curiously recurring template pattern idiom [22]. The next code snippet demonstrates a user defined visitor pattern.

```
#include "ez/generic_visitor.hpp"
struct IsUnaryExprVisitor : GenericVisitor<IsUnaryExprVisitor>
{
  // fallback case, typically cannot be reached
  void gvisit(const ipr::Node& n) { assert(false); }
  // non unary expressions
  void gvisit(const ipr::Classic& n) {}
  // for all unary operations
  template <class Cat, class T>
  void gvisit(const Unary<Category<Cat, Classic>, T>&)
  {
    res = true;
  }
  IsUnaryExprVisitor() : res(false) {}
  bool res;
};
```

The user defined visitor contains a number of possibly templated `gvisit` member functions. The generic visitor chooses the best matching implementation. The visitor in the example contains a fallback implementation defined on `ipr::Node`. The member function to handle all unary operators uses the notation discussed in the previous section to discern between unary operations and other IPR nodes. Programmers use generic visitors like any regular visitor — the base class `GenericVistor` implements the `Visitor` interface.

```
bool is_unary_expression(const ipr::Classic& n)
{
  IsUnaryExprVisitor vis;
  n.accept(vis);
  return vis.res;
}
```

Applications: The pattern match generator (§III.C), the template analysis tool (Chapter VIII), and the traversal framework (§III.B.3) use the generic visitor to query IPR.

III.B.3. Traversal Library

The traversal library factors common traversal code into a reusable component. The library offers two template classes; one (`SourceSequencetraversal`) for traversals of statements and declarations, and another one (`ExprVisitor`) for left to right traversals of expression trees. The provision of a traversal framework, allows programmers focus on writing analysis logic. Both classes need to be instantiated with a user-defined client class.

Programmers write their analysis as a client class that gets invoked for every discovered node. We split the traversal into statement/declaration- and expression level code. This is useful (and common, such as in the EDG compiler [69]) because of different concerns (e.g., control flow, evaluation sequence points, ...).

Statement traversal: The template class `SourceSequencetraversal` implements an AST traversal in source sequence order. It is parametrized over a client class. The traversal mechanism invokes its client for each discovered node. The expected client interface consists of member functions called `pre` and `post`. Each `pre` and `post` function takes a single argument. Both functions are necessary for clients that require hierarchical analysis contexts (e.g., namespaces, classes, functions, blocks).

Similar to the generic visitor class (§III.B.2) there is no predetermined object-oriented interface to which a client has to adhere. Users can mix concrete and templated `pre` and `post` functions. For each discovered statement or declaration node (n), `SourceSequencetraversal` invokes a client's `pre` function. After traversing all of n's subtrees, a client's `post` function is called. The following code snippet shows an example client counting the number of declarations and statements in a translation unit.

```
struct DeclarationCounter {
  DeclarationCounter() : declctr(0), stmtctr(0) {}
  void pre(const ipr::Stmt& s) { ++stmtctr; }
  void pre(const ipr::Decl& s) { ++declctr; }
  void post(const ipr::Stmt& s) {} // handles Stmt and Decl
  size_t declctr;
  size_t stmtctr;
};
```

The following example demonstrates the integration with the traversal class. The `SourceSequencetraversal` is instantiated with its client (`DeclarationCounter`). An object of such type behaves like a visitor.

```
#include "ez/travez.hpp"
void analyse(const ipr::Unit& pivot)
{
  ez::SourceSequencetraversal<DeclarationCounter> ctr;
  pivot.accept(ctr);
  ...
}
```

Expression traversal: The template class `ExprVisitor` implements a expression tree traversal. Like `SourceSequencetraversal` the `ExprVisitor` is parametrized over the client. The expected client interface consists in a member functions called `pre` and `post`. Each `pre` and `post` function takes two arguments. The first argument is a reference to the visited IPR node, the second argument is an arity tag (§III.B.1). The following code snippet shows an example client, that counts the number of binary and call nodes in the AST.

```
struct ExprClient {
  ExprClient() : call_ctr(0), binaryexpr_ctr(0) {}
  void pre(const ipr::Node&, ez::meta::node_tag) {}
  // for all binary operations (except call nodes)
  void pre(const ipr::Node&, ez::meta::binary_tag) { ++binaryexpr_ctr; }
  // call nodes
  void pre(const ipr::Node&, ez::meta::call_tag) { ++call_ctr; }
  void post(const ipr::Node&, ez::meta::node_tag) {}
  // subtree traversal
  void operator()(const ipr::Expr& ex, ipr::Visitor& vis) { ex.accept(vis); }
  size_t call_ctr;
  size_t binaryexpr_ctr;
};
```

In addition to the `pre` and `post` member functions, the client is also expected to implement an `operator()`. The purpose of the `operator()` is to allow clients more control over descending the expression tree (i.e., skip over some sub expression trees).

The integration of statement and expression traversal is demonstrated by the following code example:

```
struct StmtTraversal {
  ExprTraversal et; // the traversal client for expr nodes
  ez::ExprVisitor<ExprClient> trav;

  StmtTraversal() : et(), trav(et) {}

  // defined over all ipr::Stmt classes
  template <class IprNode>
  void pre(const IprNode& node);

  void post(const ipr::Node&) {}
  ...
};
```

The traversal of all children that are expressions is handled by the function `traverse_expr` which takes an `ipr::Stmt` and an `ipr::Visitor` as argument. The implementation of `StmtTraversal`'s function `pre` demonstrates the use.

```
template <class IprNode>
void StmtTraversal::pre(const IprNode& node)
{
  // defined for all statements having expressions
  using ::ez::traverse_expr;

  // empty fallback implementation (for nodes w/o expressions)
  using ::ez::darkhole::traverse_expr;

  // traverse the sub expressions
  traverse_expr(node, trav);
}
```

Applications: The described traversal framework has been employed in the implementation of TACE (Chapter VIII), the pattern match generator §III.C, and the driver of the initializer list rejuvenation (Chapter VI).

III.B.4. Equality Comparison

Source code analysis often requires tests whether two nodes are equal. Whether a specific equality test implementation is suitable depends on the type of analysis (are two declarations equal if they refer to the same declaration or the same master declaration?) and on the source of nodes (do the two nodes belong to the same translation unit?).

In the described library, the equality tests checks two nodes that belong to the same translation unit. The equality test utilizes the comparison tags described in §III.B.1. The generated pattern matching code (§III.C) uses the equality tests to determine whether two nodes are equal. The following design decisions have been made accordingly:

- `Id_expr` are equal if they refer to the same master declaration
- `Names`, often appearing in template code, are equal if they are written in exactly the same way (including all scope references)

The next code snippet shows the interface.

```
template<class IprNode >
bool equal(const IprNode &lhs, const IprNode &rhs);
```

III.B.5. Implementation

Table II gives an overview of the described libraries and their implementation.

Table II. Support libraries overview

	Files
Compile-time information	ez/meta.hpp
Generic Visitor	ez/generic_visitor.hpp
Traversal Framework	ez/travez.hpp
Equality Comparison	ez/compez.hpp

III.C. A Lightweight Pattern Matching Tool

This section describes the use of a pattern match generation tool, its input language, and its output. Chapter VI utilizes the pattern match generator to recover initializer lists from C++ code.

The pattern matcher, as understood by this implementation, emphasizes statically checkable, and finite patterns. Consequently, the pattern match generator exhibits more similarities to parser generators than to general purpose programming languages. The pattern matcher reads in a pattern description file which is written in a language that extends XPR with syntax to specify patterns. A pattern description file consists of a sequence of patterns. Pattern definitions do not follow the typical XPR style of declarations, where a name is followed by a colon. Instead, the pattern name follows the prefix keyword `pattern`.

Pattern definitions are comprised of the following elements:

- interface and context specification §III.C.2.a: Names a pattern and specifies what kind of IPR node the pattern will match.
- C++ like pattern specification §III.C.2.a: the body of the pattern can be comprised of several alternatives. The alternatives are specified in XPR.
- action blocks §III.C.2.a: for each alternative, user can specify a code sequence that gets executed when the corresponding alternative matches.
- hole variables §III.C.2.b: placeholder objects that describe an AST node structurally.
- matching of sequence (e.g., expression lists) §III.C.2.c
- pattern composition §III.C.2.d

- pattern parameters §III.C.2.e

Ambiguity resolution of several alternatives is discussed in §III.C.2.g, the grammar of the input language is given by App. A. The grammar extends XPR with productions that allow the specification of the pattern interface an hole variables.

For each specified pattern, the generates creates a class (with the same name as the pattern) that implements the pattern matching in C++. An instance of such a class is a functor, that tests whether an IPR node and its subtrees matches the pattern description. This approach is lightweight because it is independent from any specific traversal mechanism. The output can be integrated in any kind of traversal and program query mechanism (e.g., top-down or bottom-up).

III.C.1. Motivation

Programmers think of programs in terms of higher level design and source code. Text based analysis and transformation systems (e.g., grep) do not provide enough semantic information for questions for more than trivial examples. Any analysis that requires expanded macros, type information (including `typedef`) of expressions, or structural and hierarchical data of user defined types relies on more sophisticated abstract program representations. Using an abstract representation, such as the Pivot's IPR, introduces the problem that any abstract representation of source code depends on idiosyncratic artifacts. We demonstrate with examples that writing code for matching a specific pattern is non trivial:

- IPR has three kinds of nodes. Independent (e.g, declarations), unified (e.g, literals), and singleton nodes (e.g., `int` and other built in types). The node kind determines how to efficiently compare nodes for equality.

- declarations in expression context are wrapped in a `Scope` node. Uses of declarations in expression context are wrapped by `Id_expr` nodes.

- a name can refer to either a declaration, in which case it is wrapped in an `Id_expr`, or it refers to an unbound name (e.g., a built-in function, a template argument dependent function), in which case the name is stored directly in the AST.
- the use of template arguments are represented by a de Brujin abstraction [141], called `Rname`. Depending whether the template argument is a type or not, `Rnames` need to be converted into a type node (and are wrapped by `As_type` nodes).

The pattern matcher operates on a meta level of IPR. It uses the information how a proper IPR is constructed to reduce a programmer's need to write specification code for common use cases. For example, expression nodes cannot point to declarations directly — any reference to a declaration is wrapped by an `Id_expr`. The pattern matching wraps declarations automatically. The built-in domain specific knowledge improves static checking. For example, it can report some ambiguities in the pattern.

III.C.2. Input Language

As input language, we use XPR augmented with a special notation to support the definition of patterns, pattern variables, user defined matching, and (eventually) transformation rules. The use of XPR allows pattern specification "by example" in a form that is similar to the source language C++. The fact that XPR is semantically and syntactically close to C++, empowers programmers to write patterns with only a short learning period. Subsequently, the similarities between the pattern specification and implementation language reduces the likelihood of errors.

III.C.2.a. Simple Patterns

A pattern declaration starts with the description of its interface. The keyword `pattern` is followed by the pattern name (e.g., `call_pushback`), the IPR base-type that the pattern expects (e.g., `Expr`), and a list of additional arguments (e.g., an empty parameter list `()`).

```
pattern call_pushback<Expr>()
```

The specification of the expected IPR base type is more than notational sugar. The base type determines the parse context (i.e., the active production rule) of the pattern body. For example, the specification of `Expr` lets the parser parse a comma (`,`) as Comma expression; if the IPR base type is specified as `Expr_list` a comma separates a list of expressions. Consequently, statement patterns are terminated with a semicolon, while expression patterns are not.

Patterns allow the specification of "pattern variables" (see §III.C.2.b for details). Essentially, a pattern variable is a placeholder for any IPR-node that fits its declaration. For example, we can declare a pattern variable (named `obj` and `val`) of type `Expr`.

```
  obj: Expr();
  val: Expr();
```

A pattern body is enclosed by { and } and contains syntactical definitions of alternatives. The following pattern contains a single alternative that matches the invocation of a function `push_back` that takes two arguments. Note, that the Pivot represents member and non-member calls uniformly. For member calls, the receiver object becomes an explicit function argument (i.e., `obj`). The left arrow (`=>`) separates the syntactic pattern from semantic actions (empty in this example).

```
{
  push_back(obj, val) => ;
}
```

Alternative patterns: The body of a pattern can contain several alternatives. The following pattern matches both invocations of `push_back` and `pop_back`.

```
{
  push_back(obj, val) => ;
  pop_back(obj) => ;
}
```

Action blocks: Each query can be followed by an optional semantic action. The action executes if the preceding pattern matches. Actions can encode additional matching criteria or user code (e.g., state transformations, tree rewrite operations). The action blocks use XPR as their input language and allow the use of the declared pattern variables. The action block in the following snippet tests whether the `obj`'s type is a pointer to an STL [15] container. Only if the action block is completely executed, the pattern matches successfully. The `return` statement (in the then-branch of the `if` statement) leaves the action block prematurely. Thereafter, the pattern matching tries a fallback alternative, if specified. Since this example specifies no alternative, the pattern does not match when `is_stl_container` returns false. An example for a fallback alternative would be an `Expr` pattern variable.

The content of action blocks is only syntactically checked and otherwise copied verbatim into the generated C++ output files. Action blocks do not contribute to ambiguity checking (§III.C.2.g). The use of action blocks is illustrated by the following complete pattern.

A complete example: The following code presents the complete examples discussed in this section. It recognizes member function calls to `push_back` and `pop_back` of STL container objects.

```
pattern call_pushback<Expr>()
  obj: Expr();
  val: Expr();
{
  push_back(obj, val)
    => { if (!is_stl_container(obj->type())) { return; }
```

```
          else { std::cout << "found push_back"; };
        };
  pop_back(obj)
    => { if (!is_stl_container(obj->type())) { return; }
          else { std::cout << "found pop_back"; };
        };
}
```

III.C.2.b. Pattern Variables

Pattern variables (or hole variables [45]) are placeholder objects that bind to IPR nodes, which match the declared type and structure of the variable. The example in §III.C.2.a demonstrates the use of pattern variables.

Purpose: The purpose of pattern variables is twofold. First, by binding to IPR nodes, variables make matching nodes available for queries, reports, and manipulations in action blocks. Second, the criteria specification in the body of the pattern is limited to syntax. To enable programmers to explicitly query for criteria that is not expressible in source form, but that is part of the internal representation of the Pivot, variable definitions can be augmented with structural information. For example, type information has no direct syntactic correspondence in source and XPR form. In the preceding example, queries for type information were expressed in the action block. Augmenting variable declarations with structural properties exposes these properties to the pattern matcher's semantic checking (and optimizations).

Simple Pattern: The following code snippet demonstrates the use of pattern variables. The example queries for additions, where both the left- and the right-hand side subexpression are literal constants. The notion of the *class literal* cannot be expressed in syntax form. Thus we introduce two variables, named `literal_x` and `literal_y`, that are defined to have type `Literal` (and match any IPR node of type `ipr::Literal`).

```
pattern literal_addition<Expr>()
  literal_x : Literal();
  literal_y : Literal();
{
  literal_x + literal_y => ;
}
```

Non-linear matching: Non-linear patterns are patterns where a variable occurs more than once [151]. The first and any following occurrence of this variable have to match, otherwise the AST tree does not match the pattern. The following code snippet matches any addition, where the left and right hand side of the operator are equal.

```
pattern double_expr<Expr>()
  e : Expr();
{
  e + e => { if (is_numeric(x−>type())) { /* replace x+x with 2*x */ } else {} };
}
```

Code examples that fulfill the specified criteria include: `x+x`, `foo(x) + foo(x)`, `a*b + a*b` (but not `a*b + b*a`), etc. The pattern matcher relieves programmers from knowing implementation details, such as equality comparison. It simplifies writing correct and efficient code.

Property definition: While variables allow to match entire classes of nodes, they are not expressive enough to query all information present in the AST. Consider the previous example, where the constraint on the type of the pattern variable (i.e., `e`'s type refers to an STL container) was only specified in the action block. To allow for more static checking, pattern variables can be augmented with additional structural information (e.g., type, name, etc.) The following pattern constrains the pattern variable to STL's container types.

```
// assuming pattern StlContainer<Type> () { ... }
x : Expr(StlContainer());
```

This syntax binds the property to a subtree of a node by position.

- `Expr(` *type* `)`
- for any unary expression: *UnaryType* `(` *type, operand* `)`
- ...
- for any declaration: *Decl* `(` *name, type, initializer* `)`

An example is the following pattern variable declaration, that matches any variable declaration of type `int` which is initialized to `0`. The name of the variable remains unspecified. The ellipsis is used as a "do not care" symbol.
v : **Var**(..., **Int**(), 0);

Future extensions: It might be desirable to extend the specification of queryable properties to include other data (e.g., qualifiers, master declarations — a Pivot term that refers to the first declaration that appears in an translation unit, etc.) present in the AST. In such a scenario, a fixed binding of a property position to an AST subtree might become cumbersome. An alternative syntax would make the name of the subtree / property explicit:
term : **Expr**(**type** = **Int**());

A complementary alternative syntax would allow to mix XPR style code with structural declarations:
term : **Expr**(∗∗**int**); // instead of Expr(Pointer(Pointer(Int())));

III.C.2.c. Matching Sequences

Sequences, such as argument lists of function calls, require a mechanism to query some properties of each element. For example, querying for function calls that pass at least one literal as argument (e.g., for constant propagation) requires the traversal of the calls' argument list. The mechanism of choice for the pattern matcher is tail recursion. The following code snippet defines two variables, `x` and `y`. If the first argument is a literal, the first syntactic pattern (*1) matches. Otherwise, the second branch (*2) accepts any expression node. The recursive invocation of `has_constant` continues to match the remaining arguments (if any).

```
pattern has_constant<ipr::Expr_list>()
  x : Literal();
  y : Expr();
{
  x ; // *1 a argument is a literal
  y, has_constant() ; // *2 not a literal
}
```

End of sequences The pattern matcher allows various terminal elements of sequences. For example, any pattern sequence containing three elements (like `a, b, c`) matches only AST sequences having exactly three elements. We call such a sequence closed.

```
// closed ended sequences
// x, y, z are expressions
x, y, z; // matches an argument list with exactly three arguments
push_back(obj, val); // matches any push_back with two elements
```

For the specification of sequences, where partial matching is sufficient (e.g., a single argument, or the first two arguments), we introduce a "do-not-care" symbol (i.e., ". . ."). Note, that the last element of the list and the do-not-care symbol are not separated by comma. A sequence terminating with the do-not-care symbol is open.

```
// open ended sequence
// x, y, obj are expressions
// n is an integer expression
x, y ... // matches an argument list with two or more arguments
resize(obj, n ...); // matches any resize with two or more arguments
```

Patterns that iterate through elements are terminated by a (recursive) call to a pattern that is defined over the same sequence type. Typically, this would be a tail recursion.

```
// tail call
// x is an expression
// has_constant is a pattern taking an Expr_list
x, has_constant() // tail recursion that traverses the argument list
```

Alternatives with differing termination have the following precedence rules:

- patterns that are not terminated by a call to sequence pattern or the "do not care symbol" have highest precedence.
- calls of sequence pattern
- sequence patterns terminated with the "do not care symbol" have lowest precedence.

III.C.2.d. Pattern Calls

To avoid code duplication and repetitive pattern specification, it is desirable to factor common queries into separate, reusable patterns. For example, we could combine several semantically equivalent but syntactically different ways of writing code into a pattern. Consider a case, that queries variable incrementation by one (e.g., to find for-each loops). We could specify the following pattern:

```
pattern increment_by_one<Expr>(x : Var())
{
  ++x => ;
  x++ => ;
  x += 1 => ;
  x = x + 1 => ;
  // ...
}
```

Context information and results can be passed as pattern arguments. For example, to record which variable is incremented by one, the pattern utilizes a variable (`x`). The following code demonstrates the invocation of `increment_by_one` with an example.

```
pattern loop_body<Expr>()
  v : Var();
{
  increment_by_one(v)
    => { increment_variable(v); }; // attempts to match increment_by_one
  // other interesting expressions in the loop body
};
```

Since the programmer uses the variable inside the action, the variable is defined in the calling pattern (`loop_body`) and passed as argument to the called pattern (`increment_by_one`). Details on pattern parameters are discussed in §III.C.2.e.

III.C.2.e. Pattern Arguments

The list of pattern parameters allows programmers to pass variables from the context to a pattern. Parameters can either be pattern variables (bound and unbound) or have a user defined type. The semantic difference between matching bound and unbound variable makes it necessary to syntactically differentiate the parameter type.

- *unbound parameters:* parameters that have not yet been matched by the calling pattern. Like local variables, unbound parameters can have supplemental requirements. The typing rules require the formal parameter (P) be a subtype of the actual argument (A). The typical subtyping rules apply:

  $$P(p_1, \ \ldots \ , p_n) \ <: \ A(a_1, \ \ldots \ , a_n), \ \textit{iff}\ P \ <: \ A \ \wedge \ \underset{i<n}{\forall} \ p_i \ <: \ a_i$$

  This typing rule requires that the unbound variables are less specific than the parameter specification. This guarantees that matching AST nodes can be assigned to pattern variables. The following code snippet demonstrates the typing rules with examples. The variable `y` is more specific than what `some_pattern`'s parameter requires. The body of `some_pattern` cannot guarantee, that a matching variable would have a name called "x". On the contrary, passing `z` is allowed, because any variable of type `int` is also an expression.

```
pattern some_pattern<Expr>(unbound_variable : Var(..., Int())) {}
pattern caller<Expr>()
  y : Var("x", int);
  z : Expr();
{
  some_pattern(y) => ; // error
  some_pattern(z) => ; // ok
}
```

- *bound parameters:* in non-linear patterns, bound parameters have been matched in the calling pattern, any subsequent match in the called pattern has to equal the first matching node. Since bound parameters are already defined, no supplemental properties can be specified. Bound parameters are prefixed with a an ampersand (`&`).

```
pattern some_pattern<Expr>(bound_variable : &Var) {}
```

- *undecided parameters:* in some scenarios (e.g., with a number of pattern calls) it seems desirable to have parameters where the status can remain undecided.

```
pattern some_pattern<Expr>(v : Var(), l : Literal())
{
  l => ; /* matches any literal */
  v => ; /* matches any variable */
}
pattern caller<Expr>()
  v : Var();
  l : Literal();
{
  // v,l are underspecified in the second call
  some_pattern(v,l) + some_pattern(v,l) =>; // error,
}
```

The pattern matcher does currently not support such variables.

Besides domain specific pattern variables, users can also pass parameters that have any regular C++ type (including user defined types). Such parameters can be passed on to other pattern calls, or be used inside actions blocks. For example:

```
pattern some_pattern<Expr>(x : &Var, y : &int)
{
  x => { ++y; };
}
```

Design Alternative: If we allow patterns passed as arguments, we could generalize and allow user defined code (e.g., written in C++) that meets certain interface requirements. Such a feature is currently not implemented, because the pattern matcher would not be able to include user defined code in its semantic checking. Some common scenarios can be modeled using the action block that contains code matching a node against a user specified pattern.

```
pattern some_pattern<Expr>(udp : &UserDefinedPredicate)
  x : Expr();
  l : Literal();
{
  x + l => { if (!udp(x)) { return; } else { /* OK */ } };
  //              ↑↑↑↑↑ user defined matching
}
```

III.C.2.f. Matching Order

The pattern matcher uses a predetermined order to match the subtrees of a given IPR node. For example, a binary expression first traverses the left subexpression, then the right subexpression, and then the type (if needed). The predetermined traversal order together with the syntactical different notation of bound and unbound parameters allows the pattern matcher to validate that the arguments of a pattern call are indeed bound or unbound, according to the specification in the pattern declaration.

III.C.2.g. Ambiguity Checking and Resolution

A pattern can contain several queries, the pattern matcher requires a policy to handle ambiguities. Consider the following pattern:

```
pattern zero_addition<Expr>()
```

```
  term : Expr();
{
        term + 0 => ; /* 1 */
        0 + term => ; /* 2 */
}
```

The expression `0 + 0` is a match for both alternatives and therefore ambiguous. The following policies can resolve ambiguities:

- *traversal order specificity:* This policy resolves ambiguities by preferring the query that is more specific earlier in the traversal order. For the given example, the generator matches the IPR nodes in the following order: The first match verifies that the current expression is an `Add` node. If this match succeeds, the matcher processes the node's left hand side expression (*lhs*). If *lhs* is a `Literal` with the value `0`, the second alternative (2) is chosen. Otherwise the matcher falls back to match the AST subtree tree against query (1).

- *lexicographical order:* This policy resolves ambiguities by preferring earlier alternatives in the pattern body. In this example, alternative (1) is preferred over (2).

- *full ambiguity checking:* This policy detects ambiguous alternatives and reports them. The pattern match generator only accepts patterns that are ambiguity free. The example would be rejected.

  A potential objection to this resolution policy is that for complex patterns users could be required to resolve a number of ambiguous cases. Ambiguity resolution would require near duplication of the preferred alternative and action block.

- *no ambiguity resolution:* An AST can have multiple matching alternatives. The action block of each matching alternative is executed in arbitrary order.

Refinements of these techniques have to define, how subsumption is handled (e.g., alternatives are specified for `Expr` and `Literal`.

The choice of ambiguity resolution policy does not affect the runtime of the pattern matching code.

The current implementation resolves ambiguities according to traversal order specificity. Cases where multiple alternatives exhibit the same matching sequence are rejected and reported to the programmer.

III.C.3. Output

The pattern match generator converts the high-level pattern declaration into a family of C++ classes. In essence, the family of classes implement a state machine that recognizes the specified patterns.

State transitions that depend on the dynamic type of ipr-Nodes are implemented with the visitor pattern. State transitions that depend on unified nodes, singleton nodes, or certain values of nodes (e.g., a literal has the value 0), are implemented with cascading branches. By using the visitor pattern to recover type information, the generated state machine can follow the most suitable type branch out of a number of alternatives. To benefit from this trait, the pattern match generator represents alternative patterns as a trie [75]. The following example matches the addition of two literals (1) or two expressions whose type is a fundamental type (2).

```
pattern foo<Expr>()
  lit1 : Literal();
  lit2 : Literal();
  e : Expr(FundamentalType());
{
  /* 1 */ lit1 + lit2  => /* action code omitted */;
  /* 2 */ exp + exp => /* action code omitted */;
}
```

Fig. 5 depicts the pattern' trie representation. A trie exploits the common prefix of alternatives. In this example, the two alternatives share the recovery code for the binary plus operation (p). If p's left hand side operand is a literal, the matching proceeds along the first (top) path. If the p's right hand side operand is not a literal, matching falls back to the second line. If p's left hand side is not a literal (it has to be an expression), the matching proceeds along the second path. In this case, both of p's operands have to be equal.

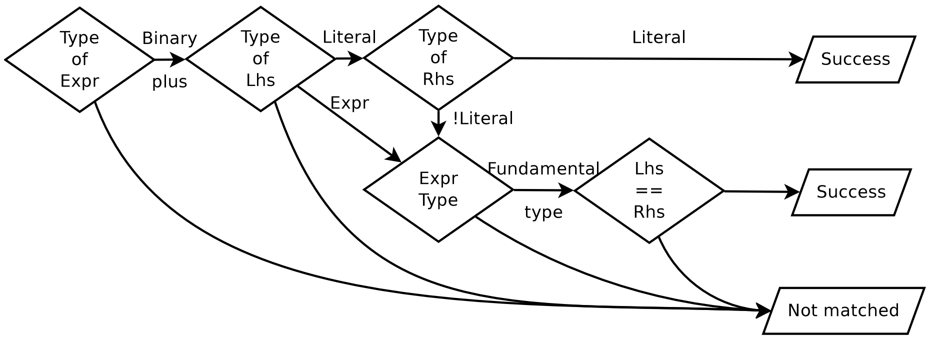

Fig. 5. Alternatives in a pattern trie

Matching patterns against an AST can be implemented in various ways. One approach is tree matching. It matches the specified pattern against each node in the AST. Another approach is based on string matching. It converts both AST and pattern to strings that can be compared using more efficient string matching implementations. We will discuss the implementation of both strategies. The current implementation is based on the simpler tree matching approach.

III.C.3.a. Tree Matching

Direct tree matching uses a rather naïve matching approach. From a given node, the pattern matcher traverses all interesting subtrees until a node does not match the pattern (failure), or no more nodes remain to match (success). No information is carried over from matching a node to matching all children of the node. This results in a worst case complexity of $O(p * c)$ (p = nodes(subtree), c = nodes(pattern)). Since the matching stops when a failure state is reached, the average case complexity is expected to be significantly better.

Tree matching implementations (can) have the traversal mechanism built in. An advantage of this approach is that the traversal of uninteresting subtrees can be skipped. For example, if the pattern does not query type information, no code for type traversal is generated. A disadvantage of building the traversal into the pattern is that whether to skip compiler generated nodes has to be known at pattern-generation time.

III.C.3.b. String Matching

Based on non-linear pattern matching as described by Ramesh and Ramakrishnan [151], the patterns are turned into strings in order to use the Aho-Corasick algorithm for string matching [8]. The basic idea is the following: every pattern is turned into a string that is terminated by a pattern variable (or pattern delimiter). Consider the pattern `term+0` as depicted in Fig. 6. Off-line preprocessing first linearizes the traversal (depicted in Fig. 7).

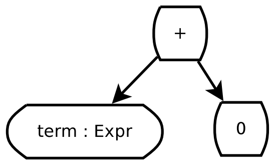

Fig. 6. Pattern tree

Then preprocessing splits the string at the variable (i.e., `term`) into two sub-strings. In Fig. 7, the splitting point is marked with a dotted gray arrow.

Based on this strings, an Aho-Corasick string recognition automaton is con-

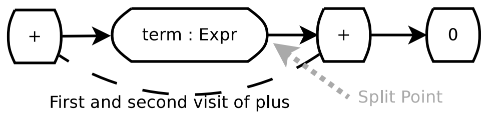

Fig. 7. Pattern tree — string traversal

structed. The traversal mechanism passes each node to the constructed automaton in a predetermined order. After the node and its subtrees have been completely traversed, the pattern matcher knows which strings match. The final step is to determine whether the substring matches are consistent with the pattern (and its non-linear pattern variables). This results in a worst case complexity of $O(p * k)$ (p = nodes in the subtree, k = number of variables (= number of substrings)).

III.C.3.c. Runtime Engine

Any of the two preceding approaches could also be implemented by a runtime engine that interprets a pattern definition.

III.C.3.d. Comparison

Table III summarizes trade-offs of implementation strategies: comparing direct tree matching with the string matching approach $k << c$ is expected. Thus, string matching should perform better than direct tree matching. Real results will vary depending on the average depth (c^*) of partial matches. We expect $avg(c^*) << c$ and consequently argue that string matching improves performance marginally in practice. Our conjecture awaits practical results.

Linearizing the tree into a string representation is an off-line process. An external traversal mechanism has to adhere to the same traversal order. On one side, an external traversal is more flexible as it can skip uninteresting intermediate nodes (e.g.,

Table III. Pattern matching strategies

	String Matching	**Tree Matching**
Worst case complexity	p * k	p * c
Avg case complexity		depends on matching depth
Binding to traversal	tight — sensitive to traversal order of children	relaxed — traversal can be built-in
Skip/Not-skip compiler generated nodes	depends on traversal	code must be generated correspondingly
p .. node count in the subtree k .. number of pattern variables c .. node count in the pattern		

compiler generated nodes, parenthesis, etc.) On the other side, a built-in traversal mechanism, as used by our implementation, has more information about the location of interesting subtrees. It can skip subtrees, such as type specification, if they are not part of the pattern specification. The disadvantage of a built-in mechanism is that it has to be known at pattern generation time which intermediary nodes can be skipped.

III.C.4. Future Work

The design choices made for the pattern match generator have been guided by its use for C++0x initializer lists recovery. The pattern match generator can be improved in a number of directions. In particular, we would like to mention three possible improvements.

Property specification: The fixed order of properties in “constructor calls” to hole variables can be replaced with named properties. This would make the declaration syntax and use of hole variables more flexible.

Pattern variables as arguments: The current implementation tracks whether a hole variable has already an assigned value. This is necessary for an efficient implementation of matching nodes. However, this scheme is too rigid for scenarios, where a pattern consists of consecutive pattern calls. The first pattern can have multiple unbound parameters. If the pattern variables are not used in all alternatives, we cannot determine whether the variables have been bound to concrete AST nodes. Calls to the second pattern cannot use the same pattern variables as argument, since we cannot statically determine their state. A possible solution would allow a third, unspecified kind of hole variable as argument.

Factor out traversal code from pattern matching code: The generated pattern match classes match an IPR node and its subtrees against the pattern specification. The traversal of subtrees is statically generated for each pattern match class. This makes it hard to write patterns that can handle propagation of constants or symbolic expressions. Separating the pattern matching code from the traversal would simplify the use of the pattern matcher in these respects.

CHAPTER IV

THE OPEN-METHODS LANGUAGE EXTENSIONS

Multiple dispatch, the selection of a function to be invoked based on the dynamic type of two or more arguments, is a solution to several classical problems in object-oriented programming. Open multi-methods generalize multiple dispatch towards open-class extensions [66], which improve separation of concerns and provisions for retroactive design. We present the rationale, design, implementation, performance, programming guidelines, and experiences of working with a language feature, called open multi-methods, for C++. Our open multi-methods support both repeated and virtual inheritance. Our call resolution rules generalize both virtual function dispatch and overload resolution semantics. After using all information from argument types, these rules can resolve further ambiguities by using covariant return types. Care was taken to integrate open multi-methods with existing C++ language features and rules. We describe a model implementation and compare its performance and space requirements to existing open multi-method extensions and workaround techniques for C++. Compared to these techniques, our approach is simpler to use, catches more user mistakes, and resolves more ambiguities through link-time analysis, is comparable in memory usage, and runs significantly faster. In particular, the runtime cost of calling an open multi-method is constant and less than the cost of a double dispatch (two virtual function calls). Finally, we provide a sketch of a design for open multi-methods in the presence of dynamic loading and linking of libraries.

IV.A. Introduction

Runtime polymorphism is a fundamental concept of object-oriented programming, typically achieved by late binding of method invocations. “Method” is a common term for a function chosen through runtime polymorphic dispatch. Most OOP languages (e.g., C++ [171], Eiffel [122], Java [14], Simula [31], and Smalltalk [82]) use only a single parameter at runtime to determine the method to be invoked ("single dispatch"). This is a well-known problem for operations where the choice of a method depends on the types of two or more arguments ("multiple dispatch"). For example, this problem has been studied in context of the binary method problem [38]. A separate problem is that dynamically dispatched functions have to be declared within class definitions. This is intrusive and often requires more foresight than class designers possess, complicating maintenance and limiting the extensibility of libraries. Open-methods provide an abstraction mechanism that solves these two problems by separating operations from classes and enabling the choice of dynamic vs. static dispatch on a per-parameter basis.

Workarounds for both of these problems exist for single-dispatch languages. In particular, the visitor pattern (double dispatch) [79] circumvents these problems without compromising type safety. Using the visitor pattern, the class designer provides an accept method in each class and defines the interface of the visitor. This interface definition, however, limits the ability to introduce new subclasses and hence curtails program extensibility [48]. Visser [190] presents a possible solution to the extensibility problem in the context of visitor combinators, which make use of runtime type information (RTTI).

Providing dynamic dispatch for multiple arguments avoids the restrictions of double dispatch. When declared within classes, such functions are often referred to as "*multi-methods*". When declared independently of the type on which they dispatch, such functions are often referred to as *open-class extensions*, *accessory functions* [195], *arbitrary multi-methods* [130], or "*open-methods*". Languages supporting multiple dispatch include CLOS [168], MultiJava [48, 129], Dylan [157], and Cecil [42]). We implemented and measured both multi-methods and open-methods. Since open-methods address a larger class of design problems than multi-methods and are not significantly more expensive in time or space, our discussion concentrates on open-methods.

Generalizing from single dispatch to open-methods raises the question how to resolve function invocations when no overrider provides an exact type match for the runtime-types of the arguments. Symmetric dispatch treats each argument alike but is subject to ambiguity conflicts. Asymmetric dispatch resolves conflicts by ordering the arguments based on some criteria (e.g., an argument list is considered left-to-right). Asymmetric dispatch semantics is simple and ambiguity free (if not necessarily unsurprising to the programmer), but it is not without criticism [41]. It differs radically from C++'s symmetric function overload resolution rules and does not catch ambiguities.

We derive our design goals for the open-method extension from the C++ design principles outlined in [170, §4]. For open-methods, this means the following: open-methods should address several specific problems, be more convenient to use than all workarounds (e.g., the visitor pattern), and outperform them in both time and space. They should neither prevent separate compilation of translation units nor increase the cost of ordinary virtual function calls. Open-methods should be orthogonal to exception handling in order to be considered suitable for hard real-time systems (e.g., [112]), and parallel to the virtual and overload resolution semantics.

IV.B. Application Domains

Whether open-methods address a sufficient range of problems to be a worthwhile language extension is a popular question. We think they do, but like all style questions it is not a question that can in general be settled without examples and data. This is why in the context of this dissertation we start with presenting examples that we consider characteristic for larger classes of problems and that would benefit significantly from language level support for multiple dispatch. We then explain fundamentals that drive the design, provide the details of our implementation and compare its performance to alternative solutions. In what follows, we mark examples with 1 when they primarily demonstrate multiple dispatch and with 2 when they demonstrate open-class extensions.

IV.B.1. Shape Intersection

An intersect operation is a classical example of multi-method usage [170, §13.8]. For a hierarchy of shapes, `intersect()` decides if two shapes intersect. Handling all different combinations of shapes (including those added later by library users) can be quite a challenge. Worse, a programmer needs specific knowledge of a pair of shapes to use the most specific and efficient algorithm.

Using the multi-method syntax from [170, §13.8], with `virtual` indicating runtime dispatch, we can write:

```
bool intersect (virtual const Shape&,     virtual const Shape& ); // open−method
bool intersect (virtual const Rectangle&, virtual const Circle &); // overrider
```

We note that for some shapes, such as rectangles and lines, the cost of double dispatch can exceed the cost of the intersect algorithm itself.

IV.B.2. Data Format Conversion

Consider an image format library, written for domains such as image processing or web browsing. Conversion between different representations is not among the core concerns of an image class and a designer of a format typically cannot know all

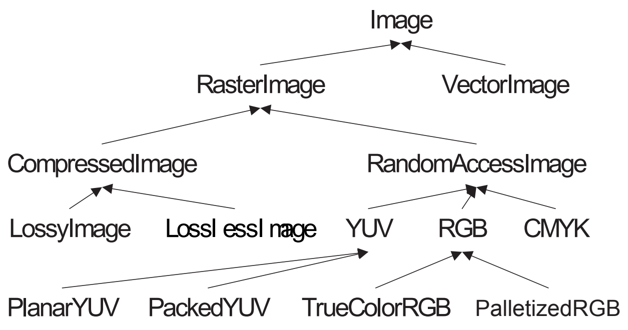

Fig. 8. Image format hierarchy

other formats. Designing a class hierarchy that takes aspects like this into account is hard, particularly when these aspects depend on polymorphic behavior. In this case, generic handling of formats by converting them to and from a common representation in general gives unacceptable performance, degradation in image quality, loss of information, etc. An optimal conversion between different formats requires knowledge of exact source and destination types, therefore it is desirable to have open-class extensions in the language, like open-methods. Fig. 8 shows some classes of a realistic image format hierarchy:

A host of concrete image formats such as RGB24, JPEG, and planar YUY2 will be represented by further derivations. The optimal conversion algorithm must be chosen based on a source-target pair of formats [96, 189]. In §V.B.1, we present an implementation of this example, here we simply demonstrate with a call:

```
bool convert(virtual const image& src, virtual image& dst);
RGB24 bmp("image.bmp");
JPEG jpeg;
convert(bmp, jpeg);
```

IV.B.3. Type Conversions in Scripting Languages

Similar to §IV.B.2, this example demonstrates the benefits of open-methods in the context of type conversions. Languages used for scripting are often dynamically typed and a value may often be converted to other types depending on use. For example, a variable x initialized as string can be used in contexts where integers or even dates are expected, while a variable y initialized as integer can be used in the concatenation operations of strings. In such cases, an interpreter will try to convert actual values to the type required in the context according to some conversion rules. A typical implementation of such conversion will use either nested switch statements or a table of pointers to appropriate conversion routines. None of these approaches is extensible or easy to maintain. However, multi-methods provide a natural mechanism for such implementations:

```
class ScriptableObject                 { ~virtual ScriptableObject (); };
class Number  : ScriptableObject  {};
class Integer : Number             {};
class String  : ScriptableObject  {};
void convert(virtual const ScriptableObject& src, virtual ScriptableObject & tgt);
void convert(virtual const Number&           src, virtual String&            tgt);
void convert(virtual const String&           src, virtual Number&            tgt);
// ... etc.
```

IV.B.4. Compiler Pass over an AST

High-level source-to-source transformation infrastructures [177, 155] typically use abstract syntax trees (AST) to represent programs. Using OOP, the commonalities of the AST classes can be factored in an OO-hierarchy. Then, programmers can write runtime polymorphic code for a family of classes by using pointers/references to a common base class.

```
struct Expr { virtual ~Expr(); };
struct UnaryExpr     : Expr { Expr& operand; };
struct NotExpr        : UnaryExpr {};
struct ComplementExpr : UnaryExpr {};
const Expr& propagate_constants(virtual const Expr& e);
```

For example, an expression class (i.e., `Expr`) would be the common base for unary (e.g., not or complement) and binary expressions (e.g., add or multiplication). Analysis or transformation passes that take the semantics of the expression into account (e.g., for propagating constants) need to uncover the underlying type. Typical implementations rely on the visitor pattern or type-tags to uncover this concrete type. Open-methods are a non-intrusive alternative for writing these compiler passes. In §V.B.2, we discuss our experience in implementing such a pass with open-methods and visitors.

IV.B.5. Binary Method Problem

Often times we have a two-argument method, whose meaning is trivial to define when both arguments are of the same type, but not so obvious in cases when arguments are of different, though related through inheritance, types. Such methods are well studied (e.g., Bruce et al [38], Boyland and Castagna [34], Langer and Kreft [109]) and characteristic to many logical and arithmetic operations. Bruce et al. call such a function a *binary method*. They define binary method of some object of type τ as a method that has an argument of the same type τ. Binary methods pose a typing problem and among different solutions to the problem, the authors propose to use multi-methods.

In the multi-methods setting, binary methods simply become multi-methods with two arguments of the same type. Consider for example an equality comparison of two objects:

```
class Point { double x, y; };
bool equal(const Point& a, const Point& b) {
  return a.x == b.x && a.y == b.y;
}
class ColorPoint : Point { Color c; };
```

When a class `ColorPoint` derives from `Point` and adds a color property, the question arises on how equal should be defined: should it just compare the coordinates or should it also compare the color properties of both arguments? The second option is only viable if both arguments are of type `ColorPoint`. If the argument types differ, we can choose either to return false or to compare the coordinates only. Depending on the problem domain both choices can be acceptable. Here we simply note that multi-methods are an ideal solution for the implementation of the latter policy where the comparison of `Point` with `ColorPoint` only compares coordinates:

```
bool equal(virtual const Point&      a, virtual const Point&      b);
bool equal(virtual const ColorPoint& a, virtual const ColorPoint& b);
```

IV.B.6. Algorithm Selection Using Dynamic Properties of Objects

Often, we can use dynamic types to choose a better algorithm for an operation than would be possible using only static information. Using open-methods we can use the dynamic type information to select more efficient algorithms at runtime without added complexity or particular foresight. Consider a matrix library providing algorithms optimized for matrices with specific dynamic properties. Storing these dynamic properties as object attributes is not easily extensible and is error prone in practice. Letting the compiler track them using open-methods for dispatch (run-time algorithm selection) is simpler. For instance, the result of $A * A^T$ is a symmetric matrix — if such a matrix appears somewhere in computations, we may consider a broader set of algorithms when the result is used in other computations.

```
class Matrix { virtual ~Matrix(); };
class SymMatrix  : Matrix     {}; // symmetric matrix
class DiagMatrix : SymMatrix  {}; // diagonal  matrix
```

```
Matrix&      operator+(virtual const Matrix&     a, virtual const Matrix&     b);
SymMatrix&   operator+(virtual const SymMatrix&  a, virtual const SymMatrix&  b);
DiagMatrix&  operator+(virtual const DiagMatrix& a, virtual const DiagMatrix& b);
```

Depending on the runtime type of the arguments, the most specific addition algorithm is selected at runtime and the most specific result type returned. The static result type would still be `Matrix&` when the static type of an argument is a `Matrix&` since we cannot draw a more precise conclusion about the dynamic type of the result (see §IV.C.4 and §IV.D.2 for details). However, since the operator is selected according to the dynamic type, the optimal algorithm will be used for the result when it is part of a larger expression.

Other interesting properties to exploit include whether the matrix is upper/lower triangular, diagonal, unitary, non-singular, or symmetric/Hermitian positive definite. Physical representations of those matrices may also take advantage of the knowledge about the structure of a particular matrix and use less space for storing the matrix.

The polymorphic nature of the multiple dispatch requires the result to be returned by either reference or pointer to avoid slicing. Since the reference must refer to a dynamically allocated object, this creates a lifetime problem for that object. Common approaches to such problems include relying on a garbage collector and using a proxy to manage lifetime. An efficient proxy is easy to write:

```
// A memory-managing proxy class (note lowercase name).
class matrix
{
    std::unique_ptr<Matrix> the_matrix; // pointer to the actual polymorphic Matrix
    matrix(Matrix& actual) : the_matrix(&actual) {}
};
matrix operator+(const matrix& a, const matrix& b)
{
    // Forward operator+ to the actual open-method and
    // attach the result to the proxy.
    return matrix(*a.the_matrix + *b.the_matrix);
}
```

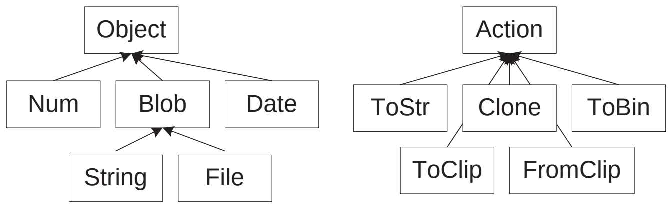

Fig. 9. Objects and actions

The `unique_ptr` is a simple and efficient (not-reference counting) "smart" pointer that is part of the C++0x standard [27] and has been widely available and used for years [25].

IV.B.7. Action Systems

Dynamically typed languages, such as Smalltalk [82], Ruby [183] or Python [186], can dispatch polymorphic calls on classes not bound by inheritance. As long as a method with a given name exists in the class, it will be called, otherwise an exceptional action is taken. Often, similar behavior is desirable in statically typed languages with possible restriction to objects derived from a certain base class. To achieve this, we may represent methods (here called actions) as objects and then apply a given action to a given set of parameters.

Fig. 9 shows a class hierarchy with objects and actions. Note that the same action applied to a different object may have a completely different meaning.

```
Object& execute(virtual const Action&     act, virtual Object& obj);
String & execute(virtual const ToString&   act, virtual Number& obj);
  File & execute(virtual const SaveToFile& act, virtual Blob&   obj);
// ... etc.
```

Action objects resemble function objects in C++ in a way. The main difference between actions and C++ function objects is that actions participate in call dispatching on equal bases with other arguments, while function objects invariably define the scope for call dispatching. Simply put, this means that in case of action objects, other arguments of a call can affect the choice of a call's target at runtime (symmetric behavior), while with function objects they cannot (asymmetric behavior).

IV.B.8. Extending Classes with Operations

Once defined, the object-oriented way to extend a class's functionality is to derive a new class and introduce the new behavior there. However, this technique only succeeds if the programmer has control over the source code that instantiates objects (for example, if the code has been designed to use a factory). Consider a system framework that responds to various events. The events may require logging in different logs and formats. While it is feasible to provide a common interface for different kinds of logs, it is rather difficult to foresee all possible formats in which logging can be done. Open-methods eliminate the need to modify class declarations directly, and improve the support for separation of concerns.

```
struct Log            {}; // Interface to different logs
struct FileLog   : Log {}; // Logs to File
struct EventLog  : Log {}; // Logs to OS event log
struct DebugLog  : Log {}; // Logs to debug output

struct Event { virtual ~Event(); }; // Interface for various types of events
struct Access          : Event  {};
struct FileAccess      : Access {};
struct DirectoryAccess : Access {};
struct DatabaseAccess  : Access {};

// Specializations of how to log various types of events in text format
void log_as_text   (virtual Access&          evt, Log& log, int priority );
void log_as_text   (virtual DirectoryAccess& evt, Log& log, int priority );

// Specializations of how to log various types of events in XML format
void log_as_xml    (virtual Access&          evt, Log& log, int priority );
void log_as_xml    (virtual DirectoryAccess& evt, Log& log, int priority );

// Specializations of how to log various types of events in binary format
void log_as_binary (virtual Access&          evt, Log& log, int priority );
void log_as_binary (virtual DirectoryAccess& evt, Log& log, int priority );

// etc. for any other formats that may be required in the future.
```

IV.B.9. Open-Methods Programming

The benefits noted in the examples stem from fundamental advantages offered by open-methods. They allow us to approximate widely accepted programming principles better than more conventional language constructs and allow us to see conventional solutions, such as visitors, as workaround techniques. For examples like the ones presented above, open-methods simply express the design more directly. From a programmer's point of view open-methods

- *are non-intrusive*: We can provide runtime dispatch (a virtual function) for objects of a class without modifying the definition of that class or its derived classes. Using open-methods implies less nonessential coupling than conventional alternatives.
- *provide order independence for arguments*: The rules for an argument are independent of whether it is the first, second, third, or whatever argument. The first argument of a member function (the this pointer) is special only in that it has notational support. The dynamic/static resolution choice has become independent of the choice of argument order.
- *improve ambiguity detection*: Ambiguous calls are detected in a way that cannot be done with only a per-argument check (as for conventional multiple dynamic dispatch) or only a per-translation-unit check (as for conventional static checking). Using open-methods there is simply more information for the compiler and linker to use.
- *provide multiple dynamic dispatch*: We can directly select a function based on multiple dynamic types; no workarounds are required.

- *improve performance*: faster than workarounds when more than one dynamic type is involved with no memory overhead compared to popular workaround techniques (see §V.A.7 and §V.C.2).

From a language design point of view, open-methods make the rules for overriding and overloading more orthogonal. This simplifies language learning, programming, reasoning about programs, and maintenance.

Potential objections to the use of open-methods include that:

- the set of operations on objects of a class are not defined within the class. However, that is true as soon as you allow any free-standing function, which is essential for conventional mathematical notation and programming styles based on that. Information hiding is not affected.

- the first argument is not fundamentally different so open-methods do not obey the "send a message to an object" ("object-oriented") model of programming. We consider that model unrealistically restrictive for many application domains, such as disciplines that apply classical math [170].

- the set of overriders for a virtual function is not found within a specific set of classes (the set of classes derived from the class that introduced the virtual function). On the other hand, we never have to define a new derived class just to be able to override.

- open-methods are open; that is, they do not provide a closed set of overloading candidates for a given function name. We consider this a good feature in that it allows for non-intrusive extension. For C++, the decision not to syntactically distinguish overriders or overloaded functions was taken in 1983 and cannot be changed now [170, §11.2.4].

Obviously, we consider open-methods a significant net gain compared to alternatives, but the final proof (as far as proofs are possible when it comes to the value of programming language features) will have to wait for the application of open-methods in several large real-world programs.

IV.C. Definition of Open Methods

Open-methods are dynamically dispatched functions, where the callee depends on the dynamic type of one or more arguments. ISO C++ supports compile-time (static) function overloading on an arbitrary number of arguments and runtime (dynamic) dispatch on a single argument. The two mechanisms are orthogonal and complementary. We define open-methods to generalize both, so our language extension must unify their semantics. Our dynamic call resolution mechanism is modeled after the overload resolution rules of C++. The ideal is to give the same result as static resolution would have given had we known all types at compile time. To achieve this, we treat the set of overriders as a viable set of functions and choose the single most specific method for the actual combination of types.

We derive our terminology from virtual functions: a function declared virtual in a base class (super class) can be overridden in a derived class (sub class):

Definition 1 *An* open-method *is a free-standing function with one or more parameters declared virtual.*

Definition 2 *An open-method f_2* overrides *an open-method f_1 if it has the same name, the same number of parameters, covariant virtual parameter types, invariant non-virtual parameter types, and a possibly covariant return type. In such case, we say that f_2 is an* overrider *of f_1.*

Definition 3 *An open-method that does not override another open-method is called a* base-method*.*

Definition 4 *A base-method together with all the open-methods that override it form an* open-method family*.*

While this is not strictly necessary, for practical reasons we require that a base-method should be declared before any of its overriders. This parallels other C++ rules and greatly simplifies compilation. This restriction does not prevent us from declaring different overriders in different translation units. For every overrider and base-method pair, the compiler checks if the exception specifications and covariant return type (if present) comply with the semantics defined for virtual functions.

Definition 5 *A* Dispatch table *(DT) maps the type-tuple of the base-method's virtual parameters to actual overriders that will be called for that type-tuple.*

The following example demonstrates a simple class hierarchy and an open-method defined on it:

```
struct A { virtual ~A(); } a;
struct B : A {} b;
void print(virtual A&, virtual A&); // (1) base−method
void print(virtual B&, virtual A&); // (2) overrider
void print(virtual B&, virtual B&); // (3) overrider
```

Here, both (2) and (3) are overriders of (1), allowing us to resolve calls involving every combination of A's and B's. For example, a call `print(a,b)` will involve a conversion of `b` to an `A&` and invoke (1). This is exactly what both static overload resolution and double dispatch would have done.

To introduce the role of multiple inheritance, we can add to that example:

```
struct X { virtual ~X(); };
struct Y : X, A {};
void print(virtual X&, virtual X&); // (4) base−method
void print(virtual Y&, virtual Y&); // (5) overrider
```

Here (4) defines a new open-method `print` on the class hierarchy rooted in X. Y inherits from both A and X, and according to our definition (5) overrides both (4) and (1).

We note that whether it would be better to require an overrider to be explicitly specified as such is an orthogonal decision beyond the scope of this dissertation. Here we simply follow the C++ tradition set up by virtual functions to do this implicitly.

IV.C.1. Type Checking and Call Resolution of Open-Methods

Type checking and resolving calls to open-methods involves three stages: compile time, link time, and runtime.

- *overload resolution at compile time*: the goal of overload resolution is to find a unique open-method in the overload set visible at the call site, through which the call can be (but not necessarily will be) dispatched. The open-method determines the necessary casts of the arguments, and the return type expected at the call site.

- *ambiguity resolution at link time*: the pre-linker aggregates all overriders of a given open-method family, checks them for return type consistency, performs ambiguity resolution, and builds the dispatch tables.

- *dynamic dispatch at runtime*: the dispatch mechanism looks up the entry in the dispatch table that contains the most specific overrider for the dynamic types of the arguments and invokes that overrider.

This three-stage approach parallels the resolution to the equivalent modular-checking problem for template calls using concepts in C++0x [83]. Further, the use of open-methods (as opposed to ordinary virtual functions and multi-methods) can be seen as adding a runtime dimension to generic programming [15].

IV.C.2. Overload Resolution

The purpose of overload resolution in the context of open multi-methods is to identify an open-method that the compiler will use for type checking and inferring the result type expected from the call. In general, the C++ overload resolution rules [97] remain unchanged: the viable set includes both open-methods and regular functions and the compiler treats them equally. Once a unique best match is found, the call can be type checked against it. For the dispatch, any of its base-methods can be chosen. Which one is selected is irrelevant as any further overrider would likewise override all base-methods.

Consider the following example:

```
struct X { virtual ~X(); };
struct Y { virtual ~Y(); };
struct Z { virtual ~Z(); };
void foo(virtual X&, virtual Y&); // (1) base−method
void foo(virtual Y&, virtual Y&); // (2) base−method
void foo(virtual Y&, virtual Z&); // (3) base−method
struct XY : X, Y {} xy;
struct YZ : Y, Z {} yz;
void foo(virtual XY&, virtual Y&); // (4) overrides 1 and 2
void foo(virtual Y&, virtual YZ&); // (5) overrides 2 and 3
```

A call `foo(xy,yz)` is ambiguous according to the standard overload resolution rules as overriders 4 and 5 are equally good matches. To resolve this ambiguity, a user may explicitly cast some or all of the arguments to make the call unambiguous accordingly to the overload resolution rules: e.g., calling `foo(xy,static_cast<Y&>(yz))` will uniquely select 4 as a base-method for the call. Alternatively, a user may introduce a new overrider `void foo(virtual XY&, virtual YZ&)`, which will become a unique best match for the call.

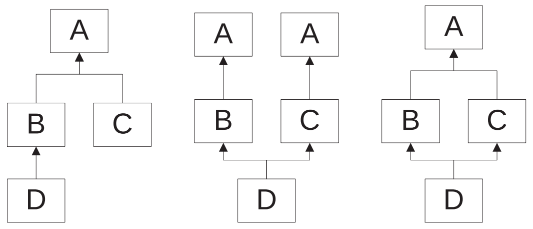

Fig. 10. C++ inheritance models

IV.C.3. Ambiguity Resolution

Once we are in the ambiguity resolution phase done by the prelinker, we assume that the overload resolution phase has selected a unique best match for type checking of each open-method call site (otherwise it would have reported a compile-time error). At this phase we have information about all available overriders of a particular open-method family, and we only report ambiguities that prevent us from building a complete dispatch table.

C++ supports single-, repeated-, and virtual inheritance:

Note that to distinguish repeated and virtual inheritance, Fig. 10 represents sub-object relationships, not just sub-class relationships. We must handle all ambiguities that can arise in all these cases. By “handle” we mean resolve or detect as errors.

Our ideal for resolving open-method calls combines the ideals for virtual functions and overloading:

- virtual functions: the same function is called regardless of the static types of the arguments at the call site.
- overloading: a call is considered unambiguous if (and only if) every parameter is at least as good a match for the actual argument as the equivalent parameter of every other candidate function and that it has at least one parameter that is

a better match than the equivalent parameter of every other candidate function.

This implies that a call of a single-argument open-method is resolved equivalently to a virtual function call. The rules described in this dissertation closely approximate this ideal. As mentioned, the static resolution is done exactly according to the usual C++ rules. The dynamic resolution is presented as the algorithm for generating dispatch tables in §IV.C.5. Before looking at that algorithm, we present some key motivating examples.

IV.C.3.a. Single Inheritance

In object models supporting single inheritance (§IV.C.3), ambiguities can only occur with open-methods taking at least two virtual parameters. Such ambiguities can only be introduced by new overriders, not by extending the class hierarchy. They can be resolved by introducing a new overrider. Open-methods with one dynamic argument are identical to virtual functions and are always ambiguity free. Thus, open-methods provide an unsurprising mechanism for expressing non-intrusive ("external") polymorphism. This eliminates the need to complicate a class hierarchy just to support the later addition of additional "methods" in the form of visitors.

IV.C.3.b. Repeated Inheritance

Consider the repeated inheritance case (§IV.C.3) together with this set of open-methods visible at a call site to `foo(d1,d2)`, where d1 and d2 are of type `D&`:

```
void foo(virtual A&, virtual A&);
void foo(virtual B&, virtual B&);
void foo(virtual B&, virtual C&);
void foo(virtual C&, virtual B&);
void foo(virtual C&, virtual C&);
```

Even though, overriders for all possible combinations of `B` and `C` (the base classes of `D`) are declared, the call with two arguments of type D gets rejected at compile time. The problem in this case, is that there are multiple sub-objects of type A inside D.

To resolve that conflict, a user can either add an overrider `foo(D&,D&)` visible at the call site or explicitly cast arguments to either the B or C sub-object. Making an overrider for `foo(D&,D&)` available at the call site eliminates the need to choose a sub-object. It would always be dispatched to the same overrider.

If the (B,C)-vs.-(C,B) conflict is resolved by casting, a question remains on how the linker should resolve a call with two arguments of type D? We know at runtime (by looking into the virtual function table's open-method table (see §IV.F)) which "branch" of a D object (either B or C) is on. Thus, we can fill our dispatch table appropriately; that is, for each combination of types, there is a unique "best match" according to the usual C++ rules:

Table IV. Dispatch table with repeated base classes

	A	**B**	**C**	**D/B**	**D/C**
A	AA	AA	AA	AA	AA
B	AA	BB	BC	BB	BC
C	AA	CB	CC	CB	CC
D/B	AA	BB	BC	BB	BC
D/C	AA	CB	CC	CB	CC

Table IV depicts the dispatch table for the repeated-inheritance hierarchy in §IV.C.3 and the set of overriders above. Since the base method is `foo(A&,A&)` and A occurs twice in D, each dimension has two entries for D: D/B meaning "D along the B branch". This resolution exactly matches our ideals.

Analog to single inheritance, extending a class hierarchy using repeated inheritance cannot introduce ambiguities. Ambiguous sub-objects are determined at compile time and reported as errors.

IV.C.3.c. Virtual Inheritance

Consider the virtual inheritance class hierarchy from §IV.C.3 together with the set of open-methods from §IV.C.3.b: In contrast to repeated inheritance, a D has only one A part, shared by B, C, and D. This causes a problem for calls requiring conversions, such as `foo(b,d)`; is that D to be considered a B or a C? There is not enough information to resolve such a call. Note that the problem can arise in such a way that we cannot catch it at compile time, because D's definition could be in a different translation unit:

```
C& rc = d;
foo(b,rc);
B& rb = d;
foo(b,rb);
```

Using static type information to resolve either call would violate the fundamental rule for virtual function calls: use runtime type information to ensure that the same overrider is called from every point of a class hierarchy. At runtime, the dispatch mechanism will (only) know that we are calling `foo` with a B and a D. It is not known whether (or when) to consider that D a B or a C. Based on this reasoning (embodied in the algorithm in §IV.C.5) we must generate the dispatch table in Table V:

We cannot detect the ambiguities marked with `??` at compile time, but we can catch them at link time when the entire set of classes and overriders is known.

Table V. Dispatch table with virtual base class

	A	**B**	**C**	**D/A**
A	AA	AA	AA	AA
B	AA	BB	BC	??
C	AA	CB	CC	??
D/A	AA	??	??	??

IV.C.4. Covariant Return Types

Covariant return types are a useful element of C++. If anything, they appear to be more useful for operations with multiple arguments than for single argument functions. Covariant return types complicate the use of workaround techniques (§V.B.2).

As an example for using covariant return type, consider a class `Symmetric` derived from `Matrix`:

```
Matrix& operator+(Matrix&, Matrix&);
Symmetric& operator+(Symmetric&, Symmetric&);
```

It follows that we must generalize the covariant return rules for open-methods. Doing so turns out to be useful because covariant return types help resolve ambiguities.

In single dispatch, covariance of a return type implies covariance of the receiver object. Consequently, covariance of return types for open-methods implies an overrider (*or*) - base-method (*bm*) relationship between two open-methods. Liskov's substitution principle [111] guarantees that any call type-checked based on *bm* can use *or*'s covariant result without compromising type safety.

This can be used to eliminate what would otherwise have been ambiguities. Consider the class hierarchies $A \leftarrow B \leftarrow C$ and $R1 \leftarrow R2 \leftarrow R3$ together with this set of open-methods:

R1∗ foo(**virtual** A&, **virtual** A&);
R2∗ foo(**virtual** A&, **virtual** B&);
R3∗ foo(**virtual** B&, **virtual** A&);

A call `foo(b,b)` appears to be ambiguous and the rules outlined so far would indeed make it an error. However, choosing `R2* foo(A&,B&)` would throw away information compared to using `R3* foo(B&,A&)`: an R3 can be used wherever an R2 can, but R2 cannot be used wherever an R3 can. Therefore, we prefer a function with a more derived return type and for this example get the result in Table VI:

Table VI. Resolution with covariance

	A	**B**	**C**
A	AA	AB	AB
B	BA	BA	BA
C	BA	BA	BA

At first glance, this may look useful, but ad hoc. However, a closer look reveals that one of the choices is simply not type safe: a call to `foo(b,b)`, type-checked against `R3* foo(B&,A&)` at compile time, would expect a pointer to an object of type R3 (or any of its sub-classes) returned, which R2 is not. This is why `R2* foo(A&,B&)` cannot be used for dispatching such a call. On the other hand, the same call type-checked against `R2* foo(A&,B&)` elsewhere is expecting a pointer to R2 (or any of its sub-classes) returned from the call, and hence would readily accept R3. This is why selecting `R3* foo(B&,A&)` is the only viable choice here, which consequently resolves the ambiguity.

From an implementational perspective, an open-method with a return type that differs from its base-method becomes a new base-method and requires its own dispatch table (or equivalent implementation technique). The fundamental reason is the need to adjust the return type in calls. Obviously, the resolutions for this new base-method must be consistent with the resolution for its base-method (or we violate the fundamental rule for virtual functions). However, since `R2* foo(A&,B&)` will not be part of `R3* foo(B&,A&)`'s dispatch table, the only consistent resolution is the one we chose.

If the return types of two overriders are siblings, then there is an ambiguity in the type-tuple that is a meet of the parameter-type tuples. Consider for example that $R3$ derives directly from $R1$ instead of $R2$, then none of the existing overriders can be used for $\langle B, B\rangle$ tuple as its return type on one hand has to be a subtype of $R2$ and on the other a subtype of $R3$. To resolve this ambiguity, the user will have to provide explicitly an overrider for $\langle B, B\rangle$, which must have the return type derived from both $R2$ and $R3$.

Using the covariant return type for ambiguity resolution also allows the programmer to specify preference of one overrider over another when asymmetric dispatch semantics is desired.

To conclude: covariant return types not only improve static type information, but also enhance our ambiguity resolution mechanism. We are unaware of any other multi-method proposal using a similar technique.

IV.C.5. Algorithm for Dispatch Table Generation

Let us assume we have a multi-method $r f(h_1, h_2, ..., h_k)$ with return type r and k virtual arguments. Class h_i is a base of the hierarchy of the i^{th} argument. $H_i = \{c : c <: h_i\}$ is a set of all classes from the hierarchy rooted at h_i. $X_f = H_1 \times H_2 \times \cdots \times H_k$ is the set of all possible argument type-tuples of f. Set $Y_f = \{\langle y_1, y_2, \cdots, y_k\rangle\} \subseteq X_f$ is the set of argument type-tuples, on which the user defined overriders f_j for f. The set $O_f = \{f_0, \cdots, f_{m-1}\}$ is the set of those overriders ($f_0 \equiv f$). $R =$

$\{r_i | r_i f_i(y_1, y_2, \cdots, y_k)\}$ is the set of return types of all the overriders. A mapping $F_f : Y_f \leftrightarrow O_f$ is a bijection between type-tuples on which overriders are defined and the overriders themselves. A function $R_f : Y_f \leftrightarrow R$ maps an argument tuple of an overrider to the return type of that overrider.

Because different derivation paths may get different entries in the dispatch table, we assume that x_i in the type-tuple $x = \langle x_1, \cdots, x_k \rangle$ identifies not only the concrete type, but also a particular derivation path for it (see [193] for formal definitions). Under this assumption, we define $\beta(x_i)$ to be a direct ancestor (base-class) of x_i in the derivation path represented by x_i. For example, for the repeated inheritance hierarchy from §IV.C.3, $\beta(D/B) = B, \beta(D/C) = C, \beta(C) = A$, while for the virtual inheritance hierarchy $\beta(D/A) = A, \beta(B) = A, \beta(C) = A$.

For the sake of convenience, we define:

$$\beta_i(x) \equiv \langle x_1, \cdots, \beta(x_i), \cdots, x_k \rangle, \text{ when } \beta(x_i) \text{ exists.}$$

With it, we extend the definition of β to type-tuples as follows:

$$\beta(x) \equiv \{\beta_i(x) \mid \beta_i(x) \text{ exists}, i = 1, k\}.$$

$P(X_f, <_P) : \langle x_1, ..., x_k \rangle <_P \langle y_1, ..., y_k \rangle \Leftrightarrow \forall i : x_i <: y_i \wedge \exists j : y_j \not<: x_j$ defines a partial ordering that models the ordering of viable functions for overload resolution as defined by the C++ ISO standard 98 [97]. $\mathit{most\_specific\_arg}(S) = \{s \in S \subseteq Y_f : \nexists t \in S : t <_P s\}$ is the set of the most specific (refined) argument tuples of S with respect to the partial ordering P. $\mathit{most\_specific\_res}(S) = \{s \in S \subseteq Y_f : \nexists t \in S : R_f(t) <: R_f(s)\}$ is the set of the most specific (refined) argument tuples of S with respect to subclassing relation <: on result types.

Dispatch table DT_f is a mapping $DT_f : X_f \rightarrow Y_f$ that maps all possible argument tuples to the argument tuples of overriders used for handling such a call.

For any combination of argument types $x \in X_f$, we recursively define entries of the dispatch table DT_f as following:

$$DT_f[x] = \begin{cases} x, x \in Y_f \\ DT_f[s], s \in S = \\ \qquad \mathsf{most\_specific\_res}(\mathsf{most\_specific\_arg}(\{\mathsf{DT_f(y)|y} \in \beta(\mathsf{x})\})) \wedge |\mathsf{S}| = 1 \\ \text{Ambiguity, otherwise} \end{cases}$$

The above recursion exhibits optimal substructure and has overlapping subproblems, which lets us use dynamic programming [54] to create an efficient algorithm for generation of the dispatch table, as shown in Algorithm 1 and Algorithm 2.

To demonstrate with an example, consider a simple class hierarchy with two classes `A` and `B`, where `B` derives from `A`, and two open-methods defined on the argument tuples $\langle A, A\rangle$ and $\langle A, B\rangle$. In this scenario $X_f = \{\langle A, A\rangle, \langle B, A\rangle, \langle A, B\rangle, \langle B, B\rangle\}$ with the following relations that hold on these argument tuples: $\langle A, B\rangle <_P \langle A, A\rangle$; $\langle B, A\rangle <_P \langle A, A\rangle$; $\langle B, B\rangle <_P \langle B, A\rangle$; $\langle B, B\rangle <_P \langle A, B\rangle$; $\langle B, B\rangle <_P \langle A, A\rangle$;

The reverse topological order of elements in X_f would thus match the order in which we listed tuples in X_f. Sets of immediate ancestors with respect to $<_P$ would be:

$$\beta(x) = \begin{cases} \emptyset, x = \langle A, A\rangle \\ \{\langle A, A\rangle\}, x = \langle A, B\rangle \\ \{\langle A, A\rangle\}, x = \langle B, A\rangle \\ \{\langle A, B\rangle, \langle B, A\rangle\}, x = \langle B, B\rangle \end{cases}$$

Note, that the empty set of immediate ancestors is only possible on the tuple that starts the open-method hierarchy, where we by definition would always have an overrider - the base-method. A set of argument tuples of overriders $Y_f = \{\langle A, A\rangle, \langle A, B\rangle\}$ and thus we can directly set $DT_f[\langle A, A\rangle] = \langle A, A\rangle$ and $DT_f[\langle A, B\rangle] = \langle A, B\rangle$

Now to fill in $DT_f[\langle B, A\rangle]$ where $\langle B, A\rangle$ is the first element in reverse topological order of X_f that is not in Y_f, we take the set of its immediate ancestors $\beta(\langle B, A\rangle) = \{\langle A, A\rangle\}$ and since there is only one, there cannot be a better match and thus $DT_f[\langle B, A\rangle] \leftarrow DT_f[\langle A, A\rangle] = \langle A, A\rangle$.

To fill in the remaining $DT_f[\langle B, B\rangle]$ that comes last in the reverse topological order, we look at its immediate ancestors $\beta(\langle B, B\rangle) = \{\langle A, B\rangle, \langle B, A\rangle\}$ and compare the overriders used for them: $DT_f[\langle A, B\rangle] = \langle A, B\rangle <_P \langle A, A\rangle = DT_f[\langle B, A\rangle]$. Thus we propagate the most specific overrider: $DT_f[\langle B, B\rangle] \leftarrow DT_f[\langle A, B\rangle] = \langle A, B\rangle$

To analyze its performance, we first note that comparison of two type-tuples from X_f can be done in time $O(k)$. If $n = max(|H_i|, i = 1, k)$ and $v = max(v_i, i = 1, k)$ (where v_i is a maximum number of times h_i is used as non-virtual base class in any class of hierarchy H_i) then $|X_f| <= (n * v)^k$ and the amount of edges for topological sort is less then $k * (n * v)^k$. Therefore the complexity of topologically sorting X_f is $O(k * n^k)$. The inner for-loop has complexity $O(k^2 * n^k)$ so the overall complexity is $O(n^k)$ since k is a constant defining the amount of virtual arguments. This means that the algorithm is linear in the size of the dispatch table.

IV.C.6. Alternative Dispatch Semantics

While our goal is to unify virtual function dispatch and overload resolution into an open-methods semantics, this is not always possible. Consider for example the repeated inheritance class hierarchy from §IV.C.3 with a virtual function added:

```
struct A       { virtual void foo(); }; // virtual function
struct B : A   {};
struct C : A   { virtual void foo(); }; // virtual function
struct D : B, C {};
```

```
void bar(A&);                 // overloaded function
void bar(C&);                 // overloaded function
void foobar(virtual A&);      // open−method
void foobar(virtual C&);      // open−method
D d;
B& db = d;                    // B part of D
C& dc = d;                    // C part of D
// (runtime) Virtual Member Function Semantics:
db.foo();                     // calls A::foo
dc.foo();                     // calls C::foo
d.foo();                      // error : ambiguous
// (compile time) Overload Resolution Semantics:
bar(db);                      // calls bar(A&)
bar(dc);                      // calls bar(C&)
bar(d);                       // calls bar(C&) (why not ambiguous?)
// (runtime) open−method Semantics:
foobar(db);                   // calls foobar(A&)
foobar(dc);                   // calls foobar(C&)
foobar(d);                    // error : ambiguous
```

Virtual dispatch semantics and overload resolution semantics go different ways in this case. Since the two language features are not entirely orthogonal, we had to decide which semantics to follow.

From a technical point of view, both semantics can be implemented for open multi-methods. The reason we decided not to model the semantics after overload resolution in this case is that the resulting cross-casting behavior could have been surprising to the user due to the implicit switching of different sub-objects. On the other hand, the difference between the ordinary virtual function (`foo`) call and the ordinary overloaded resolution for (`bar`) in this case is odd and depends on pretty obscure rules that may be more historical than fundamental. Calls to the open-method `foobar` follow the virtual function resolution. This is why our open-method semantics strictly corresponds to virtual member function semantics in ISO C++ but does not entirely reflect overload resolution semantics. The reason is that less information is available for compile time resolution than for link-time or runtime resolution. For example, the resolution of `static_cast` and `dynamic_cast` can differ even given identical arguments: `dynamic_cast` can use more information than `static_cast`.

Due to our decision to model the semantics after virtual dispatch, we require covariance of the return type on overriders, while had we modeled after overload resolution, we could have only required convertibility of return types.

IV.D. Discussion of Design Decisions

Type-safety and ambiguities have always been a major concern to systems with multiple open dispatch. One of the first widely known languages to support open-methods was CLOS [168]. CLOS linearizes the class hierarchy and uses asymmetric dispatch semantics to avoid ambiguity. Snyder [167] and Chambers [41, 42] observe that silent ambiguity resolution makes errors in programs hard to spot. Therefore, Cecil uses symmetric dispatch semantics and dispenses with object hierarchy linearization in order to expose these errors at compile time. Recent studies [11, 77, 128, 130] explore the trade-offs between multi-methods and modular type-checking in languages with neither a total order of classes nor asymmetric dispatch semantics. In particular, Millstein and Chambers discuss a number of models that embrace or restrict the expressive power of the language to different degrees. The described models range from globally type-checked programs to modularly type-checked units. We will briefly discuss our evaluation of these approaches in the context of C++ later in this section.

This work aims for maximal flexibility and relies on a global type-checking [7] approach for open-methods. We motivate this approach with the goal not only to support object-oriented programming but also to enhance the support for functional and generic programming styles in C++.

The cost of the global type-checking approach is that some ambiguities can be detected late — in particular at the load time of dynamically linked libraries (DLL). DLLs are almost universally used with C++, thus a design for open-methods that does not allow for DLLs is largely theoretical. We do not currently have an implementation supporting dynamic linking, but we outline a design addressing the major issues in such a scenario.

Our guiding principle is to support the use cases described in §IV.B with language features that are guaranteed to be type-safe in every scenario. The idea is to report errors as long as we can assume that ambiguities can be resolved by programmers. Only when it is too late for that, we have to use type-safe resolution mechanisms. This section discusses the design decisions we have made based on three language aspects: ambiguity resolution, covariant return types, and pure (abstract) open-methods.

IV.D.1. Late Ambiguities

Late ambiguities are ambiguities that are detected at a stage in the build process when programmer intervention is no longer feasible. They can occur, for example, when classes use virtual inheritance while some definitions necessary to declare a resolving overrider cannot be accessed. Consider the example given in §IV.C.3.c. Examples for late ambiguities include:

- the class `D` was defined as a local class, since the class name would be local to the function scope.
- the class `D` was defined in an implementation file of a library, but the class definition was not exported in a header file.
- a library defined classes `A`, `B`, and `C` as well as implemented, but did not export, an open-method `foo`. The definition of `D` results in a late ambiguity.

In all cases, a programmer could not declare a resolving overrider.

A second source of late ambiguities is when independently developed libraries define conflicting overriders, but the definition of one of the involved classes is not available. Consider, the single inheritance hierarchy of §IV.C.3 with an open-method `foo(A,A)`. A library defines, but does not export `B` and an overrider for `foo(B, A)`, while another library defines `C` and an overrider for `foo(A,C)`. A call `foo(b,c)` is ambiguous but cannot be resolved, because the definition of `B` is not available. Ambiguities that emerge from the use of dynamically linked libraries are always late.

Resolution mechanism for late ambiguities: If there is no unique best match for a possible type tuple, we choose an overrider from all best matches. Interestingly, any overrider will result in a correct program provided the rest of the program is correct and in principle we could even pick a random overrider from the set of best matches. Nevertheless, the choice is deterministic, but remains unspecified.

Not to specify which overrider we choose among type-safe candidates, keeps the resolution mechanism symmetric as no candidate is preferred. The use of a deterministic choice is not strictly necessary, but it allows for reproducibility - always the same method will be selected from a set of candidates.

Consider the following example of image format conversion. For a discussion of the problem and an implementation see §IV.B.2 and §V.B.1 respectively. The following code shows a common header file and two independently developed libraries that support additional image formats.

```
// Common header: ImageLibrary.h
struct Image {
  virtual ~Image();
  // ...
};
struct TiffImage : Image { /* ... */ };
void convert(virtual const Image&     from, virtual Image&    to)  { ... }
void convert(virtual const TiffImage& from, virtual Image&    to)  { ... }
void convert(virtual const Image&      from, virtual TiffImage& to) { ... }
// DLL-Jpeg supporting JPEG images
#include "ImageLibrary.h"
struct JpegImage : Image { /* ... */ };
void convert(virtual const Image&      from, virtual JpegImage& to) { ... }
void convert(virtual const JpegImage&  from, virtual Image&     to) { ... }
void convert(virtual const TiffImage&  from, virtual JpegImage& to) { ... }
void convert(virtual const JpegImage&  from, virtual TiffImage& to) { ... }
// DLL-Png supporting PNG images
#include "ImageLibrary.h"
struct PngImage  : Image { /* ... */ };
void convert(virtual const PngImage&   from, virtual Image&     to) { ... }
void convert(virtual const Image&      from, virtual PngImage&  to) { ... }
void convert(virtual const TiffImage&  from, virtual PngImage&  to) { ... }
void convert(virtual const PngImage&   from, virtual TiffImage& to) { ... }
```

The header file of an image library framework defines two classes (i.e., `Image`, `TiffImage`), and a base-method `convert` together with two overriders that implement conversions from `TiffImage` to a general `Image` and vice versa. A library (DLL-Jpeg) derives a new type `JpegImage` from `Image` and introduces new overriders for `convert` that handle all possible combinations of known image formats. Likewise, another library (DLL-Png) derives a new class `PngImage` from `Image` and introduces similar overriders. Now a call to `convert` a `JpegImage` into a `PngImage` is ambiguous. Libraries DLL-Jpeg and DLL-Png could stem from different vendors that do not know about each other. In systems that use dynamically linked libraries, such problems are hard to predict and design for.

Note that, since the class definitions of the respective other library were not available when DLL-Png and DLL-Jpeg were implemented, neither developer could possibly provide resolving overriders. The question thus arises to which `convert` should a call `convert(JpegImage, PngImage)` resolve?

Any overrider (including base-method) has to assume that a dynamic type resolving to `Image` is an unknown derived type. Consequently, each `convert` must be written so that it manipulates its arguments of types `Image` polymorphically (for example, by using virtual functions). This implies that as long as `convert`'s code does not make more assumptions about its arguments than the interface defined in the base-class guarantees, any overrider can be chosen.

Alternative techniques to handle or prevent (late) ambiguities include asymmetric choice, preventive elimination of overriders that could be prone to symmetry, or exceptions that signal an error:

- *system specified choice:* Other systems with open-methods use a specified policy to resolve ambiguities. These involves preferred treatment of overriders that are more specialized on a specified argument (e.g., CLOS [168]) and class hierarchy linearization (CLOS, Dylan [157]). Making the resolution explicit, breaks symmetric dispatch, as programmers can write code that exploits the specification.

- *limit extensibility:* Millstein and Chambers [128, 130] discuss limitations to the type systems that prevent late ambiguities. Their system M disallows virtual inheritance across modules. Moreover, open-methods have a specified argument position. Adding overriders across module boundaries is permitted only when the type in that argument is covariant and the type is defined in the same module. MultiJava [48] is based on system M. In practice, these limitations have been found to be overly restrictive (Relaxed MultiJava [129, 49] and C++ concepts [100]). In addition, requiring C++ code to comply with the provided inheritance restrictions is not an option.

  Allen et al. [11] develop a different set of restrictions for modular type checking of multiple dispatch for Fortress [10]. Instead of restricting multiple inheritance across modules, the notion of a meet function resolves ambiguities that originate from virtual inheritance. Moreover, their set of restrictions is sensitive to whether a function is a multi-method (defined in class) or an open-method (freestanding function). Overriding open-methods across module boundaries is not possible. Like in System M, overriding multi-methods is tied to a single distinguished argument position (the self argument) and the module of the type definition.

Systems that require overriders defined in another module to override a specific argument position with a covariant type defined in that module are unable to handle bidirectional image conversion well. Assuming that the first argument is special, DLL-Jpeg could not provide overriders for conversions to `JpegImage`.

- *user specified choice:* Parasitic methods as implemented in Java [35] (an implementation for Smalltalk also exists [72]) add an object-oriented flavor to multi-methods and make them an integral part of classes. Multi-methods can be inherited from a base class and overridden (or shadowed) in the derived class. Parasitic methods give the receiver precedence over other arguments. The encapsulation guarantees that a compiler can check for multiple argument ambiguities. Virtual inheritance ambiguities are implicitly resolved by users, as the resolution is sensitive to the order of multi-method declarations within the class definition.

  Frost and Millstein [77] unify encapsulated multiple dispatch with predicate dispatch. They replace the dependence on textual order with first match semantics, where later predicates implicitly exclude earlier predicates.

  The global checking presented in this dissertation resolves less virtual inheritance ambiguities silently than an encapsulated approach would. Moreover, the use of encapsulation requires control over the construction of the receiver object. Even if that can be handled by using a factory approach, this would be unable to solve and only recast the ambiguity illustrated by the conversion example: Which converter class takes precedence, the one defined by DLL-Jpeg or the one defined by DLL-Png?

- *glue-methods:* Relaxed Multi-Java [129] resolves ambiguity conflicts by introducing glue methods (to glue DLL-Jpeg and DLL-Png) that the system-integrator provides. This is a viable solution for software developers integrating several libraries, but it is not a feasible scenario for end-user applications, as dynamically linked modules can be loaded into the process without the direct request of a developer. This is the case for various component object models where applications may request an object by name from the system. The operating system will locate and load the module in which the object resides.

- *throw an exception:* Some implementations (e.g., Cmm [166]) throw an exception at dispatch time when an ambiguity is encountered. We disagree with this approach because each candidate alone is a type-safe choice and should be able to handle the requested operation. Moreover, this approach forces programmers to consider open-method calls as a potential source for exceptions, while their choice of how to handle this exception is limited and likely will result in program termination.

- *program termination:* Instead of waiting until runtime, the application can terminate (or fail to link) when ambiguous overriders are detected. We argue analogously to the exception case that termination is an inadequate response for a choice among type-safe operations.

IV.D.2. Consistency of Covariant Return Types

Before we go into a detailed discussion, we would like to point out that the main focus of this section is on the consistency of covariant return types among overriders available at runtime. The use of covariant return type for ambiguity resolution is orthogonal to the problems discussed here and is discussed in detail in §IV.C.4.

Different DLLs can specify conflicting covariant return types. Consider a two-class hierarchy $A \leftarrow B$ and another two-class hierarchy $R1 \leftarrow R2$. The base-method `R1 foo(virtual A&, virtual A&)` is defined in a header visible by two dynamically linked modules D_1 and D_2 that do not know anything about each other. Module D_1 introduces overrider `R2 foo(A&, B&)` and module D_2 introduces overrider `R1 foo(B&, B&)`. Each of the dynamically linked modules perfectly type-checks and links with `foo()` resolved through the dispatch table in Table VII (a superscript in a cell denotes the type that is returned by an overrider e.g., AB^2 denotes `R2 foo(A&, B&)`):

Table VII. Call resolution in D_1 and D_2

$AA^1 in D_1$	**A**	**B**	$AA^1 in D_2$	**A**	**B**
A	AA^1	AB^2	**A**	AA^1	AA^1
B	AA^1	AB^2	**B**	AA^1	BB^1

When both libraries are linked together, we get the dilemma of how to resolve a call with both arguments of type B. On one side `foo(B&,B&)` from D_2 is more specialized, but on the other side `foo(A&,B&)` from D_1 imposes the additional requirement that the return type of whatever is called for $\langle B, B\rangle$ should be a subtype of R2, which R1 is not. Such scenario would be rejected at compile/link time, however at load time we do not have this option anymore.

Keeping all dispatch tables of a particular open-method consistent on the overrider that will be called for a particular combination of types will force us to choose between suboptimal and type unsafe alternatives. What is worse is that there may not be a unique type-safe alternative.

Imagine for example that a module D_3 introduces an overrider `R3 foo(B&, A&)` where $R1 \leftarrow R3$, so $R2$ and $R3$ are siblings. When D_1 and D_3 are loaded together, neither `R2 foo(A&, B&)` nor `R3 foo(B&, A&)` can be used to resolve a call with both arguments of type B - both alternatives are type unsafe for the other overrider.

To deal with this subtlety, we propose for the DLL case to weaken the requirement that the same overrider should be called for the same tuple of dynamic types regardless of the static types used at the call site. We require that the same overrider be used only if it is type-safe for the caller. Strictly speaking `R1 foo(B&,B&)` is not an overrider of `R2 foo(A&, B&)` as defined in §IV.C, because its return type is not changing covariantly in respect to the types of arguments. Therefore, it cannot be considered for the dynamic resolution of calls made statically through the base-method `R2 foo(A&, B&)`.

Taking the above into account, we propose that the dynamic linker fills in the dispatch table of every base-method independently. This results in the following Table VIII.

Table VIII. D_1 and D_2 loaded together

AA^1	**A**	**B**	AB^2	**B**	BA^3	**A**	**B**
A	AA^1	AB^2	**A**	AB^2			
B	BA^3	BB^1	**B**	AB^2	**B**	BA^3	BA^3

It looks as if the dispatch table for the base-method `R1 foo(A&,A&)` now violates covariant consistency, but in reality it does not because all the return types in it are cast back through thunks to R1, which is the type statically expected at the call site.

As can be seen, this logic may result in different functions being called for the same type tuple depending on the base-methods seen at the call site. We note, however, that *the call is always made to the most specialized overrider that is type-safe for the caller.*

IV.D.3. Pure Open-Methods

There are no abstract (pure virtual) open-methods; that is, every open-method must be defined. Consider a (dynamic) library D1 that introduces a new class and a second (dynamic) library D2 that defines a new abstract open-method. When both libraries are (dynamically) linked together the presence of an overrider for the class in D1 can not be guaranteed. The alternative would be runtime "method not defined" errors (reported as exceptions), but that solution would be inconsistent with the rest of C++ and would limit the use of open-methods in embedded systems.

IV.E. Relation to Orthogonal Features

In this section, we discuss the relationship of open-methods to other language features.

IV.E.1. Namespace

Virtual functions have a class scope and can only be overridden in the derived classes. Open-methods do not have such a scope by default, so the question arises when should an open-method be considered an overrider and when just a different open-method? Let's look at the following example:

```
namespace X
```

```
{
  class A {};
  void bar(virtual A&); // base method
  class B : A {};
  void bar(virtual B&); // (1)
}
namespace Z
{
  void bar(virtual B&); // (2)
}
namespace Y
{
  class D : X::A {};
  void bar(virtual D&); // (3)
}
class C : X::A {};
void bar(virtual C&); // (4)
```

In the presented implementation, an overrider has to be declared in the same namespace as its base-method (1). Open-methods with the same name and compatible parameter types, defined in different namespaces would not be considered overriders. The major benefit of this approach is that it is easy to understand and implement. Unfortunately such semantics are not unifiable with overrider declarations of virtual function calls, where derived classes can be declared in a different namespace. `using` declarations present a potential work around to these limitations.

An alternative would be to let overriders be declared in any namespace (1,2,3,4). It is easy to understand, but defeats the purpose of namespaces that were introduced to better structure the code and avoid name-clashes among independently developed modules.

Another alternative may consider an open-method to be an overrider, if its base-method is defined in the same scope or in the scope of their argument types and their base classes. In this scenario (1, 3, 4) would override; (2) would not. Among its advantages is that it closely resembles argument dependent lookup. It would also work for virtual functions. Its downside, however, is that it is harder to comprehend.

IV.E.2. Access Privileges

Open-methods are generic freestanding functions, which do not have the access privileges of member functions. If an open-method needs access to non-public members of a class, that class must declare it a particular open-method as a friend.

IV.E.3. Smart Pointers

In C++, programmers use smart pointers, such as `auto_ptr` (in current C++) as well as `shared_ptr` and `weak_ptr` (in Boost [26] and C++0x [25]) for resource management. The use of smart pointers together with open-methods is no different from their use with (virtual) member functions. For example:

```
struct A { virtual ~A(); };
struct B : A {};
void foo(virtual A&);
void bar(shared_ptr<A> ptr)
{
  foo(*ptr);
}
```

Defining open-methods directly on smart pointers is not possible. In the following example, (1) yields an error, as `ptr1` is neither a reference nor a pointer type. The declaration of (2) is an error, because `shared_ptr` is not a polymorphic object (it does not define any virtual function). Even when `shared_ptr` were polymorphic, the open-method declaration would be meaningless. A `shared_ptr<B>` would not be in an inheritance relationship to `shared_ptr<A>`, thus the compiler would not recognize `foo(virtual shared_ptr<B>&)` as an overrider.

```
void foo(virtual shared_ptr<A> ptr1); // (1) error
void foo(virtual shared_ptr<A>& ptr2); // (2) error
```

IV.F. Implementation

We have implemented open-methods as described in §IV.C by modifying the EDG compiler front-end [69]. This includes dispatch table generation and thunk generation for multiple inheritance and covariant return. To reduce the dispatch table size, we have also implemented the dispatch table compression techniques presented in [12]. Our current implementation does not support dynamically linked libraries and detection of late ambiguities.

IV.F.1. Changes to Compiler and Linker

Our mechanism extends ideas presented in [71, 195] as to the compiler and linker model. We adopted the multi-method syntax proposed in [170], which in turn was inspired by an earlier idea by Doug Lea (see [170, §13.8]). One or more parameters of a non-static freestanding function can be specified to be `virtual`. Overloading functions based only on the virtual specifier is not allowed.

A virtual argument must be a reference or pointer to a polymorphic class (that is, a class containing at least one virtual function). For example:

```
struct A { virtual ~A(); };
void print (virtual A&);              // ok
void print (int, virtual A*);         // ok
void print (int, virtual const A&); // ok
void dump(virtual A);                 // compiler error
void dump(virtual int);               // compiler error
```

For each translation unit, the EDG compiler lowers the high level abstractions in C++ to equivalent code in C. We added an implementation that lowers open-method calls to C according to the object-model presented in §IV.F.2. In addition, the compiler puts out an *open-method description* (OMD) file that stores the data needed to generate the runtime data structure discussed in §IV.F.2. This includes the names of all classes, their inheritance relationships, and the kind of inheritance. Open-methods are represented by name, return type, and their parameter list. Finally, the OMD-file also contains definitions of all user-defined types that appear in signatures of open-methods (both as virtual and regular parameters). These definitions are necessary to generate class definitions for arguments to open-methods that are passed by value.

The pre-linker uses Coco/R [196] to parse the OMD-files. Then, the pre-linker synthesizes the OMD-data, associates all overriders with their base-methods, generates dispatch tables, issues link-errors for ambiguities, determines the indices necessary to access the open-method, and initializes the data structures described in §IV.F.2.

When the call of an overrider requires adjustments of the this-pointers (as is sometimes needed in multiple inheritance hierarchies), the pre-linker creates thunks and makes the dispatch table entries refer to them instead. During dispatch table synthesis, the linker will report errors for all argument combinations that do not have a unique best overrider. The output of the pre-linking stage is a C-source file containing the missing definitions. If the linker generates a library, the pre-linker also puts out a merged OMD-file.

IV.F.2. Changes to the Object Model

We augment the IA-64 C++ object model [98] by four elements to support constant time dispatching of open-methods. First, for each base-method there will be a dispatch table containing the function addresses. Second, the v-table of each sub-object contains an additional pointer to the *open-method table (om-table).* Finally, the indices used for the open-method-table offsets are stored as global variables.

The Figures 11 and 12 show the layout of objects, v-tables, om-tables and dispatch-tables for repeated and virtual inheritance. Our extensions to the object-

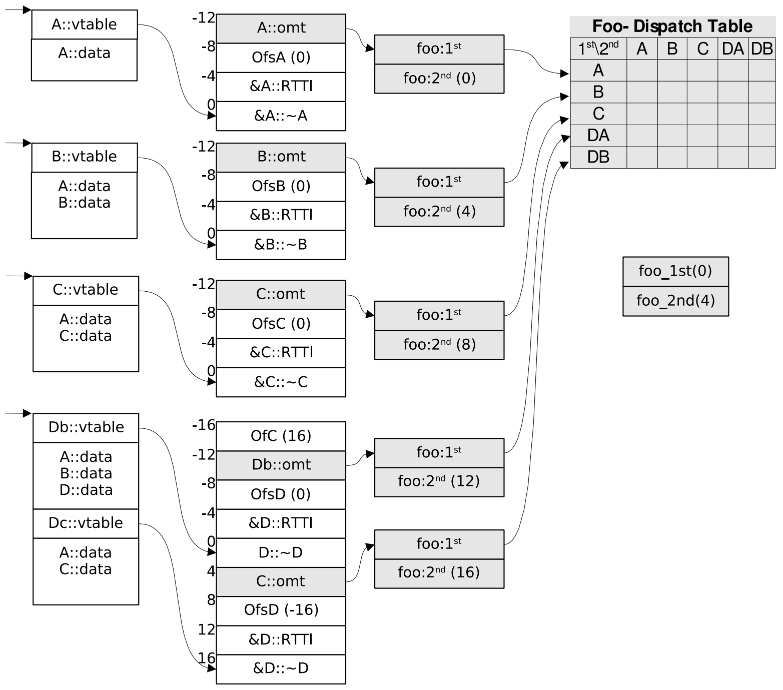

Fig. 11. Object model for repeated inheritance

model are shown with gray background. From left to right the elements in each diagram represent the object, v-table, om-table, and dispatch table(s) for the class hierarchy in §IV.C.3. From top to the bottom, the objects are of type A, B, C, and D respectively.

An open-method can be declared after the declarations of the classes used in its virtual parameters. Therefore, the compiler cannot reserve v-table entries to store the data related to open-method dispatch immediately in a class's virtual function table. Hence, we always extend every v-table by one pointer referencing the om-table, which can be laid down later by the pre-linker.

The om-table reserves one position for each virtual parameter of each base-method, where objects of this type can be passed as arguments. This position stores an index into the corresponding dimension of the dispatch table. Since the size of

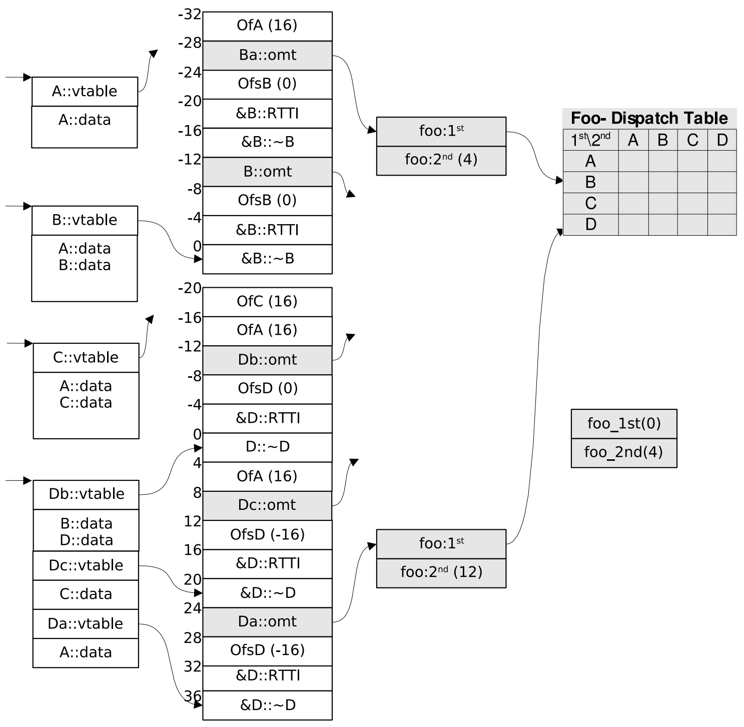

Fig. 12. Object model for virtual inheritance

the om-tables is not known at compile time, our technique relies on a literal for each open-method and virtual parameter position (called foo_1st, foo_2nd in Figures 11 and 12 that determines the offset within the om-tables.

Note that these figures depict our actual implementation, where entries for first argument positions already resolve one dimension of the table lookup. Entries for all other argument positions store the byte offset within the table.

In the presence of multiple-inheritance, a this-pointer shift might be required to pass the object correctly. In this case, we replace the address of the overrider by an address of a thunk that takes care of correctly adjusting the this-pointer. As described in §IV.C.3.b in case of repeated inheritance, different bases can show different dispatch behavior depending on the sub-object to which the this-pointer refers. As a result, different bases may point to different om-tables. In case of virtual inheritance, the open-method dispatch entries are only stored through the types mentioned in the base-method. Hence, in the virtual inheritance case, all open-method calls are dispatched through the virtual base type.

IV.F.3. Alternative Approaches

We considered a few other design alternatives and explored their trade-offs in extensibility and performance.

IV.F.3.a. Multi-Methods

Unlike open-methods, multi-methods require the base-method to be declared in the class definition of its virtual parameters. This allows the offset within the v-table be known at compile time, which saves two indirections per argument of a function call (one for the om-table, and one to read the index within the om-table). For a call with k virtual arguments, open-methods need $4k + 1$, while multi-methods need only $2k + 1$ memory references to dispatch a call. The downside of multi-methods is that existing classes cannot easily be extended with dynamically dispatched functions.

With the restriction of in-class declarations imposed by multi-methods it seems logical to declare a multi-method either as a member function or as a friend non-member function. Consider:

```
class Matrix
{
 // multi-method declaration as a non-member function
 friend Matrix& operator+(virtual const Matrix& lhs, virtual const Matrix& rhs);

 // equivalent declaration as a member function
 virtual Matrix& operator*(virtual const Matrix&);
};
```

We implemented only the non-member version of multi-methods. The member version can be implemented with exactly the same techniques. However, in many cases it is harder to write code that uses the member version because an overrider must be a member of (only) one class – and the main rationale for multi-methods is to elegantly deal with combinations of classes. Even the non-member (friend) version is hard to use.

By requiring a declaration to be present in a class, we limit the polymorphic operations to those that the class designer thought of. That requires too much foresight of the class designer or leads to unstable classes (classes that keep having multi-methods added). Such problems are well-known in languages relying on member functions. Open-methods provide an abstraction mechanism that solves such problems by separating operations from classes.

IV.F.3.b. Chinese Remainders

As we saw in §IV.F.2, support of open-methods required an extra indirection via open-method table to get index of the class in appropriate argument position. This extra indirection was needed because open-methods are not bound to the class and as a result, we do not know how many of them a class may have, therefore we cannot reserve entries in the v-table for them. In this section, we present an "ideal" scheme for implementing open-methods, inspired by ideas presented in [80]. The proposed scheme circumvents the necessity for open-method tables by moving all the necessary information from the class to the dispatch table.

Suppose that for every multi-method f there is a function $I_f : T \times N \rightarrow N$ such that for any type $t \in T$ (where T is a domain of all types) and argument position $n \in N$ it returns index of type t in the n^{th} dimension of the f's dispatch table. If such function is reasonably fast (preferably constant time) and its range is small (preferably from zero to the maximum number of types that can be used in any argument position) then we can efficiently implement multiple dispatch by properly arranging rows and columns according to the indices returned by I_f. As in [80], we use the Chinese Remainder theorem [54] to generate function I_f.

Chinese Remainder Theorem

Let $m_1, \cdots, m_k$ be integers with $gcd(m_i, m_j) = 1$ whenever $i \neq j$. Let m be the product $m = m_1 m_2 \cdots m_k$. Let $a_1, \cdots, a_k$ be integers. Consider the system of congruences:

$$\begin{cases} x \equiv a_1 (\text{mod } m_1) \\ x \equiv a_2 (\text{mod } m_2) \\ \cdots, \\ x \equiv a_k (\text{mod } m_k) \end{cases}$$

Then there exists exactly one $x \in Z_m$ satisfying this system.

Since we may have different class hierarchies in different argument positions, we have to consider each argument position separately. Assuming that there can be q different types $t_{i1}, t_{i2}, \cdots, t_{iq}$ in an argument position i, we may assign a different prime number $m_{ij} : j = 1, q$ to each of them and then according to the Chinese Remainder Theorem find a number x_i that satisfies the above equation. Storing x_i for each dimension (argument position) of dispatch table, we will come to the dispatching algorithm shown in Algorithm 3. Since k is known at compile time, no actual iteration is required and the algorithm takes constant time.

In this scenario, every class (or more specifically every argument position i where this class may appear as virtual argument) will have a prime number m_i assigned to it, while the dispatch table will have a number x_i computed through Chinese Remainders, associated with each of its dimensions. The result of $x_i \mod m_i$ gives us the column within the appropriate dimension of dispatch table.

This dispatching technique has the nice property that it does not need any modifications of the v-table in order to introduce a new open-method on the class. Once allocated, prime numbers can be reused for any number of open-methods defined on the class regardless of the argument position in which a type is used. After dispatch table allocation, we simply have to compute the number x_i for each of the argument positions. Extending such a table, that may be required after introduction of a new class in the hierarchy, is also simple: allocate new rows and columns and recompute x_i taking prime numbers of newly added classes into account.

We demonstrate the approach with an example. Consider the following class hierarchy and an open-method `foo` defined on it:

```
class A {}; // Assigned prime 2
class B : public A {}; // Assigned prime 5
class C : public A {}; // Assigned prime 3
class D : public B, public C {}; // Assigned prime 13 for D/B and 7 for D/C
class E : public D {}; // Assigned prime 17 for E/B and 19 for E/C sub-object

void foo(virtual A&, virtual A&);
void foo(virtual B&, virtual B&);
void foo(virtual B&, virtual C&);
void foo(virtual C&, virtual B&);
void foo(virtual C&, virtual C&);
void foo(virtual E&, virtual E&);
```

Table IX depicts the resulting dispatch table. Dispatching a call will then look like $DT_{foo}[X_{DT_{foo}} \bmod P(a_1), X_{DT_{foo}} \bmod P(a_2)](a_1, a_2)$. Having pointers to the actual arguments of a call, we can look up the v-tables of those arguments and the prime numbers associated with their types. Suppose that the prime number associated with the first argument is 11, while the prime number associated with the second argument is 3. To get the row number inside the dispatch table associated with the first argument, we compute the remainder of dividing 1062506 by 11, which is 5. Row number 5 corresponds to the B sub-object of an object with dynamic type E. Similarly we get the column associated with the type of the second argument through finding

Table IX. Dispatch table with repeated base classes

	$\mathbf{A}^2$	$\mathbf{B}^5$	$\mathbf{C}^3$	$\mathbf{D/B}^{13}$	$\mathbf{D/C}^7$	$\mathbf{E/B}^{11}$	$\mathbf{E/C}^{17}$	
$\mathbf{A}^2$	AA	AA	AA	AA	AA	AA	AA	$x_{\equiv 0(\bmod 2)}$
$\mathbf{B}^5$	AA	BB	BC	BB	BC	BB	BC	$x_{\equiv 1(\bmod 5)}$
$\mathbf{C}^3$	AA	CB	CC	CB	CC	CB	CC	$x_{\equiv 2(\bmod 3)}$
$\mathbf{D/B}^{13}$	AA	BB	BC	BB	BC	BB	BC	$x_{\equiv 3(\bmod 13)}$
$\mathbf{D/C}^7$	AA	CB	CC	CB	CC	CB	CC	$x_{\equiv 4(\bmod 7)}$
$\mathbf{E/B}^{11}$	AA	BB	BC	BB	BC	EE	EE	$x_{\equiv 5(\bmod 11)}$
$\mathbf{E/C}^{17}$	AA	CB	CC	CB	CC	EE	EE	$x_{\equiv 6(\bmod 17)}$
	$x_{\equiv 0(\bmod 2)}$	$x_{\equiv 1(\bmod 5)}$	$x_{\equiv 2(\bmod 3)}$	$x_{\equiv 3(\bmod 13)}$	$x_{\equiv 4(\bmod 7)}$	$x_{\equiv 5(\bmod 11)}$	$x_{\equiv 6(\bmod 17)}$	$x = 1062506$

the remainder of dividing 1062506 by 3, which is 2. Column number 2 corresponds to type C, which means that the dynamic type of the second argument is C. The number 1062506 is associated with the dispatch table, through which the call is being dispatched. To find the overrider that will be handling the call, we simply look up an address of the function that is stored at the intersection of the fifth row and the second column, which is `foo(B&,C&)`.

Despite its elegance, this approach is rather theoretical because it is hard to use for large class hierarchies. The reason is that we need to assign different prime numbers to each class and perform computations on numbers that are bound by the product of these primes. The product of only the first nine primes fits into a 32-bit integer and the first 15 primes into a 64-bit integer. Table compression techniques [12] or the use of minimal perfect hash functions [54] instead, can help overcome the problem.

In response to our paper [145], Gabor Greif sent us his unpublished notes on a similar use of Chinese Remainders for implementing multiple dispatch [85] in Dylan.

Algorithm 1 Dispatch Table Generation

```
$T \leftarrow$ topological_sort($X_f$) // Topologically sort according to $<_P$
$S \leftarrow$ reverse($T$) // Reverse the order to have the least specific first
**for all** $x \in S$ **do**
  **if** $x \in Y_f$ **then**
    $DT_f[x] \leftarrow x$ // Overriders themselves are the best matches for their arguments
  **else**
    $ancestors = \beta(x)$ // Get type-tuples of immediate ancestors
    $most\_specific = \{DT_f[\text{extract\_any}(ancestors)]\}$
    **while** $\neg$ empty($ancestors$) **do**
      $a \leftarrow$ extract_any($ancestors$)
      $dominated \leftarrow$ find_dominated($a, most\_specific$)
      **if** $\neg$ $dominated$ **then**
        // When none of the overriders was more specific
        $most\_specific \leftarrow most\_specific \cup \{DT_f[a]\}$
      **end if**
    **end while**
    **if** $|most\_specific| = 1$ **then**
      // There was a unique most specific overrider, use it
      $DT_f[x] \leftarrow y,$ where $most\_specific = \{y\}$
    **else**
      Error: Unable to find unique best overrider among $most\_specific$
    **end if**
  **end if**
**end for**
```

Algorithm 2 find_dominated(a, most_specific)

```
for $e \in most\_specific$ do
    if $DT_f[a] <_P DT_f[e]$ then
        // This ancestor's overrider is more specific
        $most\_specific \leftarrow most\_specific - \{e\}$
    else if $DT_f[e] <_P DT_f[a]$ then
        // Overrider in the most_specific set is more specific
        $dominated \leftarrow true$
        return $false$
    else if $R_f(DT_f[a]) <: R_f(DT_f[e])$ then
        // Incomparable by arguments, but more specific return type
        $most\_specific \leftarrow most\_specific - \{e\}$
    else if $R_f(DT_f[e]) <: R_f(DT_f[a])$ then
        // Incomparable by arguments, but ancestor's return type is less specific
        return $false$
    end if
end for
return $true$
```

Algorithm 3 Dispatching with Chinese Remainders

```
for all argument positions $i$ of a multi-method $f$ do
    $n_i = x_i \mod m_i$
end for
call $D[n_1, \cdots, n_k]$ with arguments provided
```

CHAPTER V

COMPARISON WITH WORKAROUND TECHNIQUES

The previous chapter introduced an open-method language extension. In this chapter, compare source code for C++ with open-methods and source code for C++ that makes use of alternative implementation techniques.

We study the use of open-methods together with the runtime concept idiom. Open multi-methods are a language feature at the intersection of object-oriented programming and generic programming §IV.C.1. Runtime concepts, developed by Sean Parent and Mat Marcus at Adobe Systems, augment generic programming with a programming model that supports dynamic polymorphism. We utilize the open-methods to provide runtime dispatch to algorithm implementations for runtime concepts. We compare the open-method implementation with an implementation in ISO C++.

Then we utilize open-methods for the implementation of image format conversion and the traversal of a compiler AST. We compare the implementations with a visitor pattern based counterpart. We conclude this chapter by comparing run time and space implications of different implementation methods.

V.A. Background on Runtime Concepts

A key benefit of generic programming is its support for producing modules with clean separation. In particular, generic algorithms are written to work with a wide variety of types without requiring modifications to them. The *Runtime concept* idiom extends this support by allowing unmodified concrete types to behave in a runtime polymorphic manner. In this dissertation, we describe one implementation of the runtime concept idiom, in the domain of the C++ standard template library (STL). We complement the runtime concept idiom with an algorithm library that considers both type and concept information to maximize performance when selecting algorithm implementations. We present two implementations, one in ISO C++ and one using an experimental language extension. We use our implementations to describe and measure the performance of runtime-polymorphic analogs of several STL algorithms. The tests demonstrate the effects of different compile-time vs. run-time algorithm selection choices.

V.A.1. Introduction

ISO C++ [97] supports multiple programming paradigms [174], notably object-oriented programming and generic programming. Object-oriented techniques are used when runtime polymorphic behavior is desired. When runtime polymorphism is not required, generic programming is used, as it offers non-intrusive, high performance compile-time polymorphism; examples include the C++ Standard Template Library (STL) [15], the Boost Libraries [26], Blitz++ [187], and STAPL [13].

Recent research has explored the possibility of a programming model that retains the advantages of generic programming, while borrowing elements from object-oriented programming, in order to support types to be used in a runtime-polymorphic manner. Parent [136] introduces the notion of non-intrusive value-based runtime-polymorphism, which we will refer to as the *runtime concept* idiom. Marcus et al. [114, 4], and Parent [137] extend this idea, presenting a library that encapsulates common tasks involved in the creation of efficient runtime concepts. Järvi et al. [101] discuss generic polymorphism in the context of library adaptation.

A key idea in generic programming is the notion of a *concept*. A concept [83] is a set of syntactic and semantic requirements on types. Syntactic requirements stipulate the presence of operations and associated types. In the runtime concept idiom, a class C is used to model these syntactic requirements as operations. The binding from C to a particular concrete type T is delayed until runtime. Any type T that syntactically satisfies a concept's requirements can be used with code that is written in terms of the runtime concept.

In this section, we apply these principles to develop a runtime-polymorphic version of STL sequence containers and their associated iterators. Runtime concepts allow the definition of functions that operate on a variety of container types.

Consider a traditional generic function expressed using C++ templates:

```
// conventional template code
template <class Iterator>
Iterator
random_elem(Iterator begin, Iterator end) {
  typename Iterator::difference_type dist = distance(begin, end);
  return advance(begin, rand() % dist);
}
// ...
int elem = *random_elem(v.begin(), v.end()); // v is a vector of int
```

Objects of any type that meet the iterator requirement can be used as arguments to `random_elem`. However, those requirements cannot be naturally expressed in C++03, (though they can in C++ with concepts [27]), and the complete function definition is needed for type checking and code generation. The resulting code is very efficient, but this style of generic programming does not lend itself to certain styles of software development (e.g., those relying on dynamic libraries).

We can write essentially the same code using the runtime concept idiom (the classes of the runtime concept idiom and their implementation are discussed in §V.A.3):

```
// with runtime concept idiom
wrapper_forward<int>
random_elem(wrapper_forward<int> begin, wrapper_forward<int> end) {
  wrapper_forward<int>::difference_type dist = distance(begin, end);
  return advance(begin, rand() % dist);
}
// ...
int elem = *random_elem(v.begin(), v.end()); // v is a vector of int
```

Here the binding between the iterator type and the function is handled at runtime and we can compile a use of `random_elem` with only the declarations of `random_elem` available:

```
// with runtime concept idiom:
wrapper_forward<int>
random_elem(wrapper_forward<int> begin, wrapper_forward<int> end);
// ...
int elem = *random_elem(v.begin(), v.end()); // v is a vector of int
```

By using runtime concepts, function implementations (e.g., `random_elem`) are isolated from client code. The parameter type `wrapper_forward` subsumes all types that model the concept forward-iterator. The implementation can be explicitly instantiated elsewhere for known element types, and need not be available to callers.

This reduced code exposure in header files makes runtime concepts suitable for (dynamically linked) libraries and when source code cannot be shared. However, the use of runtime concepts comes at a cost. The function `random_elem` is written in terms of the concept forward-iterator. The runtime complexity of `distance` and `advance` is $O(n)$ for forward-iterators, while it is constant time for random access iterators. Passing iterators of `vector<int>` as arguments would incur unnecessary runtime overhead.

When using STL iterators directly, the compiler checks that two iterators have the same type (though it does not validate that two iterators belong to the same container instance). As a consequence of subsuming iterators from various data structures under a single runtime concept, type checking that two iterators have the same concrete type is postponed until run time.

V.A.2. Generic Programming

The ideal for generic programming is to represent code at the highest level of abstraction without loss of efficiency in both actual execution speed and resource usage compared to the best code written through any other means. The general process to achieve this is known as *lifting*, a process of abstraction where the types used within a concrete algorithm are replaced by the semantic requirements of those types necessary for the algorithm to perform.

Semantic Requirement: Types must satisfy these requirements in order to work properly with a generic algorithm. Semantic requirements are stated in tables, in documentation, and may at times be asserted within the code. Checking types against arbitrary semantic requirements is in general undecidable. Instead, compilers for current C++ check for the presence of syntactic constructs, which are assumed to meet the semantic requirements. For example, an implementation of `operator=` is expected make a copy.

Concept: Dealing with individual semantic requirements would be unmanageable for real code. However, sets of requirements can often be clustered into natural groups, known as *concepts*. Although any collection of requirements may define a concept, only concepts which enable new classes of algorithms are interesting.

Model: Any type that satisfies all specified requirements of a concept is said to be a model of that concept.

Generic Algorithm: A generic algorithm is a derivative of a family of efficient algorithms, whose implementation is independent from concrete underlying data structures. The requirements that an algorithm imposes on a data structure can be grouped into concepts. An example that is part of the STL is `find` and the requirement on the template argument is to model a forward-iterator.

Concept Refinement: A concept C_r that adds requirements to another concept C_0 is a concept refinement. The number of types that satisfy C_r is less or equal to the number of types that satisfy C_0. The number of algorithms that can be directly expressed with C_r is greater or equal than the number of algorithms expressed with C_0. For example, constant time random access, a requirement added by the concept randomaccess-iterator, enables the algorithm `sort`.

Algorithm Refinement: Parallel to concept refinements, an algorithm can be refined to exploit the stronger concept requirements and achieve better space- and/or runtime-efficiency. For example, the complexity of `reverse` for bidirectional-iterator is $O(n)$, while it is $O(n\ lg\ n)$ for forward-iterator (assuming less than $O(n)$ memory usage).

Regularity: Dehnert and Stepanov [59] define regularity based on the semantics of built-in types, their operators, the complexity requirements on the operators, and consistency conditions that a sequence of operations has to meet. Regularity is based on value-semantics and requires operations to construct, destruct, assign, swap, and equality-compare two instances of the same type. This is sufficient for a number of STL data structures and algorithms, including `vector, queue, reverse, find`. A stronger definition adds operations to determine a total order, which enables the use of STL's `map, set, sort`. Programmers' likely familiarity with built-in types makes the notion of regularity important, because code written with built-in types in mind (e.g., STL data structures) will work equally well for regular user defined types. Consequently, conformance with regular semantics reduces the need for specialized and customized implementations.

V.A.2.a. Standard Template Library

The C++ STL provides implementations of efficient algorithms and data structures. The links between them are called iterators, which resemble pointers to elements plus operations that move this pointer to other elements in the container. Each data structure provides its own iterator implementation. The iterators can be grouped according to the access capabilities of their respective data structure. STL defines five iterator concepts: input iterator (for data sources), output iterator (for data sinks), forward iterator (for sequential and repeated access), bidirectional iterator, and random access iterator.

Algorithms are defined in terms of iterator concepts. At compile time the compiler statically selects the most suitable algorithm for a concrete iterator type. Data structures and algorithms are coupled through iterators. Instead of having data structures specific iterators, STL classifies them according to data access capabilities into only five categories. This abstraction simplifies programming (programmers deal with fewer categories), reduces the coupling between components (algorithms and data structures are independent), and improves code maintainability. STL consists of about 60 algorithms and 12 data structures [174]. The non-generic equivalent would be 720 specific algorithms. Most algorithms, such as `merge`, operate on ranges. A range consists of a pair of iterators that delimit a sequence of elements. The first iterator points to the first element in the sequence, the second to the position one element past the end of the sequence. A common access pattern is to iterate through the range until the first and second iterator compare equal. Thus, 720 can be considered a conservative estimate because it ignores algorithms that operate on multiple ranges.

V.A.3. Runtime Concepts for STL Iterators

This section presents our implementation of the runtime concept idiom for iterators in C++03. For a general treatment of the runtime concept idiom, along with its library support and optimizations we refer the reader to Marcus et al. [114, 4]. We extend the external polymorphism pattern [47] with a regular wrapper. The result is a three layered architecture: the concept layer, the model layer, and the wrapper layer. We illustrate the implementation of these classes and their interaction based on the concept forward iterator and its refinement bidirectional iterator. Each of the layers is parameterizable with a value-type and a reference-type. We omit the latter in our discussion.

V.A.3.a. Concept Layer

The concept layer defines the runtime concept interfaces - abstract base classes without any data. Each of these interfaces defines a set of function signatures that reflect the concept requirements. The following code snippet shows parts of the forward-iterator class.

```
template <class ValueType>
struct concept_forward {
  virtual void next() = 0; // for operator++
  virtual bool equals(const concept_forward& rhs) const = 0; // for operator==
  virtual concept_forward& clone() const = 0; // for copy ctor, operator=
  virtual ValueType& deref() const = 0; // deref operator*
  // ...
};
```

For operators that have a prefix and a postfix version (e.g., `operator++`), the concept layer contains only one member function declaration (e.g., `next`), which returns `void`. The wrapper layer (§V.A.3.c) implements the prefix and postfix operator in terms of this function and returns the appropriate result.

Refinements of a runtime concept inherit from another runtime concept interface and add new operations or refine inherited signatures with covariant return types.

```
template <class ValueType>
struct concept_bidirectional : concept_forward<ValueType> {
  virtual void prev() = 0; // for operator--
  virtual concept_bidirectional& clone() const = 0; // overrider with covariant result
  // ...
};
```

V.A.3.b. Model Layer

The model layer defines model classes that inherit from the runtime concept interfaces. Models are parametrized classes, where the first template argument determines the concrete iterator type and the second template argument corresponds to the concept interface that this model will implement. For an iterator of `list<int>`, these would be `list<int>::iterator` and `concept_bidirectional<int>` respectively. At the root of the model-hierarchy is the class `model_base` that holds a copy of the concrete iterator (member variable `it`).

```
template <class Iterator, class IterConcept>
struct model_base : IterConcept { // connect implementation type to interface type
  Iterator it;
};
```

The class `model_base` together with its two template arguments `Iterator` and `IterConcept` already determine the model refinement.

The implementation of the pure virtual functions defined in the concept interfaces is accomplished by mixin techniques [165] and class hierarchy linearization [177]. For each runtime concept interface there exists a model class that implements the corresponding pure virtual functions. For example, `model_bidirectional` implements the pure virtual functions defined in `concept_bidirectional`. The model classes (directly or indirectly) derive from `model_base` and linearly mixin their function implementations, which forward the calls to the concrete iterator `it`.

```
template <class Iterator, class IterConcept>
struct model_forward : model_base<Iterator,IterConcept> {
  IterConcept& clone();

  bool equals(const concept_forward& rhs) const {
    assert(typeid(*this) == typeid(rhs));
    return this->it == static_cast<const model_forward&>(rhs).it;
  }
  /* ... */
};
template <class Iterator, class IterConcept>
```

```
struct model_bidirectional : model_forward<Iterator,IterConcept> {
  void prev();
  /* ... */
};
```

The model classes require the same template arguments as `model_base`. This is necessary because, for example, `model_forward` could be mixed in with a `model_bidirectional` to implement the functions defined by `concept_forward`. In this case, `model_forward` would indirectly derive from `concept_bidirectional`.

Operations with two polymorphic arguments (e.g., `equals`) require both arguments to have the same dynamic type. Since C++ does not allow covariant arguments, this restrictions cannot be modeled statically (i.e., `equals` takes a `concept_forward`), but the same type requirement allows the argument be cast to the dynamic type of the receiver object. This limits the use of the equality operator to comparisons with the same underlying iterator type; comparisons of an iterator with its const-counterpart are currently not supported, but could be encoded with true multi parametric functions (see §V.A.6, double dispatch [30], or multi-method libraries [9, 159]).

The factory-function `generate_model` selects a model refinement based on the concept of a concrete iterator. Its use guarantees the existence of exactly one runtime model type for each concrete iterator type. This allows operations with two runtime-concepts as arguments to rely on the same type requirement.

```
template<class Iterator>
typename map_iteratortag_to_model<Iterator>::type*
generate_model(const Iterator& it) {
  typedef typename map_iteratortag_to_model<Iterator>::type iterator_wrapper_type;
  return new iterator_wrapper_type(it);
}
```

The template meta function `map_iteratortag_to_model` uses the iterator category to generate the proper model type. For example, for a `list<int>::iterator`, the generated class would be `model_bidirectional<list<int>::iterator, concept_bidirectional<int> >`. Then `generate_model` constructs an object of this type on the heap, and returns its address.

V.A.3.c. Wrapper Layer

The wrapper layer defines regular classes that wrap a concept layer object and manage its lifetime.

```
template <class ValueType>
struct wrapper_forward {
  concept_forward<ValueType>* iterator_m;
  // Constructor that takes any iterator
  template <class ConcreteIterator>
  wrapper_forward<ValueType>(const ConcreteIterator& iter)
      : iterator_m(generate_model(iter)) {}
  wrapper_forward<ValueType>(const wrapper_forward<ValueType>& iter)
      : iterator_m(iter.iterator_m−>clone()) {}
  // Postfix operator implemented in terms of concept_forward::next
  wrapper_forward<ValueType> operator++(int) {
    wrapper_forward<ValueType> tmp(*this);
    this−>iterator_m−>next();
    return tmp;
  }
  /* ... */
};
```

It implements operations that guarantee regular semantics (constructor, destructor, etc.) and other operations required by the STL concept. Here, the implementations of the copy constructor and the assignment operator makes use of the function `clone`.

Fig. 13 summarizes the interaction of the different layers of the runtime concept idiom:

The wrapper class is the placeholder for a concrete iterator type and holds a pointer to an interface. The model base inherits from this interface and holds a

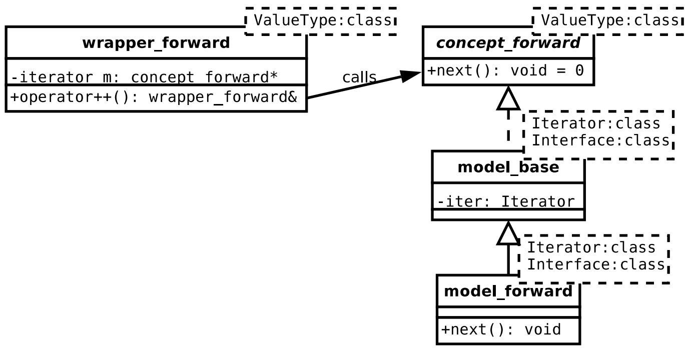

Fig. 13. Layers of the runtime concept idiom

copy of a concrete iterator (e.g., `std::vector<int>::iterator`), while the model classes implement the pure virtual functions. A member function call, for example `operator++`, invokes a virtual function of the concept interface (`next`), implemented by `model_forward`, which forwards the call to the `operator++` of the concrete iterator. The functions in the wrapper class, and the call to the concrete iterator can be inlined. Each invocation of a wrapper function incurs the overhead of at most one virtual function call. `Swap` and `move` can be implemented by manipulating the concept interface pointer of the wrapper class [114].

V.A.4. The Algorithms Library

In this section, we address performance problems that arise in the presence of type erasure [1]. First, virtual functions incur runtime overhead and impede inlining. Second, the functions `distance` and `advance` in the body of `random_elem` are selected at compile time according to the static type of the wrapper class which can lead to less than optimal runtime complexity. A dispatch solution (algorithm library) that postpones the function selection until runtime, when dynamic information about the wrapped iterator type and its concept is available, can do better (as the C++98

STL does through static resolution). We outline the design ideals and discuss two implementations: one in ISO C++ and one using an experimental language feature, open multi-methods [145].

V.A.4.a. Design Goals

Practical solutions to the dispatch problem should address the following issues:

Extensibility: the runtime concept idiom allows for separate compilation of library code and user code. Therefore, the library implementation cannot make any assumption about specific container and iterator types in the user code. The algorithm library enables user code to optimize the runtime of the library functions for the data structures it uses.

Applicable to algorithms operating on multiple ranges: for STL algorithms that operate on a single range (e.g., `find`, `reverse`, `sort`, etc.) the best function can be selected according to the dynamic type of a single argument (the begin of a range); we can assume that the end of the range has the same dynamic type. Some algorithms operate on two ranges (e.g., `merge`, `set_union`, `transform`, etc.) and require the dispatch mechanism to be sensitive in respect to more than one argument type to achieve optimal performance. Any implementation that supports dispatch according to more than one dynamic argument, requires some policy to handle ambiguities.

Conservative in executable size: the executable should not contain code for algorithms that are not needed. This means that our example library for `random_elem` contains algorithm instances only for `advance` and `distance` but not for any other STL algorithm.

Independence from runtime concept classes: providing new algorithms and better algorithm instances has to be achievable without repeated modification of the runtime concept classes.

Usability: From the user perspective, programming with the algorithm library should be similar to programming with the STL and providing algorithms and algorithm instances should be straightforward.

Implementable in ISO C++: a practical solution should be implementable in current C++ and not require any language or compiler specific extension. From a research perspective, however, we are interested in finding generally usable abstraction mechanisms that improve the design and implementation.

Performance: to be usable independently from the size of the data set, the runtime overhead for making a dispatch decision should be as small as possible.

Particularly, the support for multiple arguments and extensibility is so important for generality and performance that the designers of compile time concepts for C++0x decided to postpone checking of specialized algorithms for some potential ambiguities to instantiation time [100].

V.A.5. Implementation in ISO C++

Our implementation prototypes an algorithms library for several STL algorithms. Each function in the library originates from an algorithm instantiation with one of the iterator wrappers or a concrete iterator. By default, the library contains an instance for the weakest concept an algorithm supports. For example, the default entry for `lower_bound` would be instantiated with `wrapper_forward`. These default instances are meant to serve as fallback-implementations. To improve performance, the system integrator or even a (dynamically loaded) library can add more specialized algorithm instances. This is essential to preserve the algorithmic efficiency of key STL algorithms. The default implementation and the added instances form an algorithm family. Each algorithm family is defined in terms of an existing STL algorithm and the iterator-`value_type` (e.g., `algolib::advance<int>`).

To provide for modular extensibility, we decouple the algorithm library from static dispatch structures (i.e., enumerations, switch statements, visitor classes, typelists, variant types). This is what visitor based implementations and many C++ multimethod libraries depend on (e.g., Loki library [9], Shopyrin's deferred recursive mechanism [159], doublecpp [30], GIL [33]). Other open method implementations depend on language extensions and require preprocessor/compiler support (e.g., Cmm [166], Omm [145]) to extract the information that is necessary for an efficient implementation. Instead, our implementation relies on runtime type information (RTTI) and data embedded in the concept and model classes.

The mechanics of our implementation rest on three core components:

- a *class hierarchy traversal* that returns the typename of the base class for a given typename (or class).
- an *associative data structure* that stores functions and the RTTI information of their runtime polymorphic argument.
- a *lookup mechanism* that uses the class hierarchy traversal to find the best matching algorithm instance from the associative data structure.

To remain independent from specific compilers and their application binary interface (ABI), we encode the refinement (inheritance) relationship by mapping a typeid of a class to the typeid of its base class. In the context of the modeled iterator hierarchy, the use of single inheritance suffices.

```
struct inheritance {
  static std::map<const std::type_info*, const std::type_info*> bases;
  inheritance(const std::type_info& base, const std::type_info& derived) {
    bases[&derived] = &base;
  }
};
```

To automatically keep track of the refinement relationships of all instantiated runtime concept classes, we augment the class `model_base` and all classes of the concept layer (e.g., `concept_bidirectional`), except the base (`concept_forward`), with a static data member `inh`. By making the constructor refer to `inh` we force `inh`'s instantiation without incurring any runtime overhead because referencing `inh` has no direct effect and can be optimized away.

```
template <class ValueType>
struct concept_bidirectional : concept_forward<ValueType> {
  static inheritance inh;
  concept_bidirectional() { inh; } // triggers instantiation of inh
  // ...
};
```

`inh`'s constructor records the refinement relationship in the global mapping `inheritance::bases` that we later use to query a class for its base.

```
// static member initialization
// inh's constructor records the refinement relationship
template <class ValueType>
inheritance
concept_bidirectional<ValueType>::inh(typeid(concept_forward<ValueType>),
                                      typeid(concept_bidirectional<ValueType>));
```

Since a library developer is unaware of the specific data structures in user code, it is at the discretion of the user to enhance the runtime by providing better algorithm instances. This is supported by two functions that take the algorithm family and an iterator type as argument. The first (`add_generic`) instantiates a function that rewraps the model into a more powerful wrapper class specified by the iterator type and invokes the STL algorithm with it. The second, `add_specific` generates a function that peels off all runtime concept layers and invokes the STL algorithm with the concrete iterator. The following sample code shows how to register a generic implementation for `wrapper_randomaccess` and a specific for `list<int>::iterator`. Note that, the distinction by name of `add_generic` and `add_specific` is not necessary, but choosing to do so makes the use of virtual functions inside of an algorithm instance explicit.

```
// add generic implementation suitable for all random access iterators.
algolib::add_generic<
    algolib::advance<int>, // library name
    wrapper_randomaccess<int> // iterator−type
  >();
// add specific implementation for std::list<int>.
algolib::add_specific<
    algolib::advance<int>, // library name
    std::list<int>::iterator // iterator−type
  >();
```

In addition, we provide library functions with names that match their STL counterparts (e.g., `distance`). They look up and forward the call to the best matching algorithm instance. These functions are defined in the same namespace as the wrapper classes. Together with argument dependent lookup (ADL) in C++, this allows source code resemble code written with STL iterators. The following code snippet shows a function that takes two iterator wrappers as arguments and calls the library functions (i.e., `distance`, `advance`).

```
wrapper_forward<int>
random_elem(wrapper_forward<int> begin, wrapper_forward<int> end) {
  wrapper_forward<int>::difference_type dist = distance(begin, end);
  return advance(begin, rand() % dist);
}
```

At runtime, a library call selects the best applicable function present based on the dynamic type of the model. Starting with the `typeid` of the actual iterator model, it walks the `typeid`s of the inheritance chain until an algorithm or the fallback implementation is found. The following code snippet shows the lookup mechanism:

```
template <class A>
typename A::instance_map::data_type
lookup(typename A::dyn_iterator_type_pack pack) {
  typename A::instance_map::iterator end = A::instances.end();
  do {
    // lookup algorithm instance for type tuple described by pack
    typename A::instance_map::iterator pos = A::instances.find(pack);
    if (pos != end) return pos->second;

    // try with next type tuple in the poset
    pack = next(pack);
  } while (pack != A::fallback_iterator_pack);
  return A::fallback_instance; // return fallback, if no better instance is present
}
```

The template argument `A` is a data structure that describes the algorithm and has the following dependent types defined:

- `dyn_iterator_type_pack` is a type tuple that initially contains `type_id`s of the iterator models that are involved in the dynamic dispatch.
- `instance_map` is the type of the associative data structure, which maps a `dyn_iterator_type_pack`s to an algorithm instance.

In addition, `A` defines the following static data members:

- `fallback_iterator_pack` is a type tuple (of type `dyn_iterator_type_pack`) that identifies the concept classes of the fallback implementation.
- `fallback_instance` is a pointer to the fallback instance.

The function `next` uses the stored inheritance relationships to generate a poset of type tuples with the following ordering relation: $a \succeq b$ if for each component i of the tuples the subclassing relationship $a_i <: b_i,\ or\ a_i = b_i$ holds.

This ordering relation guarantees the lookup to find a most specific algorithm instance, if one exists. Should multiple equally specific instances exist, one of them is chosen depending on the implementation of `next`. When, like in the presented algorithms, the selection depends on the dynamic type of a single parameter, `next`'s order becomes total.

If `random_elem`'s `begin` and `end` wrap the concrete type `std::list<int>::iterator`, the lookup mechanism will find an algorithm instance that peels off all runtime concept layers and calls `std::advance` with a `std::list<int>` iterator. In case `begin` and `end` belong to a `std::vector`, the runtime model is re-wrapped by a `wrapper_randomaccess` iterator and `std::advance<wrapper_randomaccess>` is invoked.

Virtual function based design: Although the dispatch mechanism is semantically equal to virtual function calls, we rejected alternative library designs that would model algorithms as pure virtual functions declared in concept interfaces. This would break the separation between concept requirements and algorithms. Providing a new algorithm would require adding a new function signature to the concept interface, thereby breaking binary compatibility with existing applications. In addition, such a design would create a number of unused instantiated functions. For example, the class `concept_forward` would need virtual function declarations for all STL algorithms that are defined for forward iterators (e.g., `adjacent_find`, `fill`, `equal_range`, etc.). Consequently, the model classes would need to implement those functions regardless whether a specific program uses them or not. Finally, extending a virtual function based design towards multiple virtual arguments is complex when not all classes are known beforehand.

Evaluation: The described library eliminates virtual function calls inside an algorithm, when a matching algorithm instance is present. Its dispatch mechanism meets the outlined design criteria except for performance and the capability to consider more than one dynamic iterator type for dispatch decisions. The performance tests presented in §V.A.7.b (i.e., for `lower_bound`) indicate that the dispatch cost can be significant. In particular, for small data sets the overhead of algorithm lookup can outweigh the more efficient execution of the STL algorithm. Extending the dispatch towards multiple arguments aggravates the situation. Even for optimized implementations, that flatten inheritance data and record uniquely best matching functions for a given type tuple (as implemented in Smith's Cmm [166]), Pirkelbauer et al. [145, 147] report that a significant overhead remains.

Incorporating parts of the implementation outlined in [145, 147] into a ISO C++ based design of the algorithm library with the goal to improve performance is possible, but raises other problems. For example, dispatch tables, which map a combination of types to a function, need to be computed beforehand. Moreover, a library implementation cannot modify the object-model, thereby requiring an additional indirection to map RTTI to indexes into the dispatch table.

Consequently, we argue that a practical solution to the multi-parametric dynamic dispatch and performance problems would benefit from proper language support. A design principle of C++ is to aim for language features that are general and address a number of specific concerns in several domains [170]. Following these guidelines we postulate a design based on an experimental language extension — open multi-methods. Open multi-methods, allow for programming styles that mix elements from object-oriented and functional programming [157] and object-oriented and generic programming [147].

Open multi-methods address two fundamental problems in object-oriented software design. First, they overcome dispatch problems where the choice of method depends on the dynamic type of more than one argument. This is paralleled by generic programming, where all arguments have equal weight during overload resolution. Second, extending classes, possibly defined in third party libraries, with dynamically dispatched functions (virtual functions) is intrusive or requires provision for the visitor pattern [79]. This often requires more foresight than class designers are given, complicating maintenance and limiting the extensibility of libraries. Open multi-methods separate the definition of dynamically dispatched functions from class definitions. Knowing that classes can be efficiently extended later allows class designers focus on the core functionality and eliminate unnecessary code. This is another parallel with generic programming, where algorithms build on core requirements that concepts define.

V.A.6. Implementation in C++ with Open Multi-Methods

In this section, we describe our algorithm library implementation based on open multi-methods. To begin, we show an open multi-method declaration `advance` for forward iterators with a more specific implementation for random access iterators. In open multi-method terminology, the most general declaration is called a *base-method*, and a more specific declaration is called an *overrider*. A base-method declaration has to precede an overrider declaration.

```
void advance(virtual concept_forward<int>& it, int dist); // base−method
void advance(virtual concept_randomaccess<int>& it, int dist); // overrider
```

With these open-method declarations, we can provide an implementation for an iterator wrapper:

```
void advance(wrapper_forward<int>& it, int dist) {
  advance(∗it.iterator_m, dist); // invokes open−method advance
}
```

Depending on the dynamic type of `iterator_m` the call to advance either invokes the base-method (for forward and bidirectional iterators) or the overrider (for random access iterators).

The internal mechanism of open multi-methods renders the core components of the previous implementation (§V.A.5) obsolete. Our experimental compiler extracts the signature of each open multi-method and the class hierarchy from each translation unit. (This eliminates the need for recording the inheritance relationship of the concept- and model-classes.) The information is passed to a pre-linker, which synthesizes this data across translation units, checks for ambiguities, and generates a dispatch table containing entries for each possible type combination. (This eliminates the need to maintain the data structure, that associates algorithm instances with RTTI of the dedicated argument.) The runtime lookup to find the best matching function occurs in constant time for a given number of virtual parameters. (This eliminates the hierarchy traversal during algorithm instance lookup.)

For the implementation of our algorithm library, we reject the simple design outlined in the previous code snippet. This would require library users to specify function bodies for overriders (e.g., for `advance` with `concept_randomaccess<int>`). Moreover, runtime concept based libraries could only contain declarations of the open multi-methods. They could not contain definitions for open multi-methods because other libraries that use the runtime concept idiom could define them as well, leading to violations of the one definition rule (ODR). Solutions to this problem would either require users to write more open multi-method bodies, or make the pre-linker runtime concept aware and have it generate the missing definitions. Neither of these approaches seems practical at this stage.

Instead, we base our implementation on a combination of open-methods and the template mechanism. Currently, open multi-methods cannot be defined as template, and for this work we refrain from extending our experimental compiler in that direction. C++ does not support templated virtual functions - as this feature would require the linker to support creating the v-tables [170]. Templated open multi-methods would take this idea one step further by allowing templated virtual parameters:

```
template <class T>
void advance(virtual T& it, int dist);
// or
```

```
template <class T>
void advance(virtual concept_forward<T>& it, int dist);
```

Such a language feature is unprecedented and the definition of its semantics is a topic for further research. However, the instantiation of classes and their functions is well defined. For our current implementation, we harness the friend mechanism and the standard template class instantiation to generate algorithm instances. Friends allow us to comply with the requirement that base-methods and overriders have to be free standing functions that are defined in the same scope. Class template instantiation allows the user to generate open multi-methods without the need to write function bodies. In addition, the C++ standard has relaxed ODR rules for functions that are generated from templates. The problem of linker generated v-tables is overcome by having a pre-linker create open-method tables (the open multi-method analogue to v-tables).

We show a somewhat simplified implementation of `reverse` to illustrate the idea. The actual code is a bit more involved and requires, for example, a superfluous extra parameter to enable overload resolution to include friend functions that are only declared inside a class into the candidate set. The omitted details are transparent to the library user and therefore do not impact the usability.

The base-method `reverse` is declared as friend inside of a template class and thus is not a class member per se. The template class `rtc_reverse` allows us to indirectly define `reverse` based on a template argument.

```
// defined in namespace algolib
template <class ValueType>
struct rtc_reverse {
  typedef concept_bidirectional<ValueType> base_concept;
  friend void reverse(virtual base_concept& begin, base_concept& end) {
    /* wrap begin,end into wrapper_bidirectional and call std::reverse */
  }
};
```

Similarly, we define `rtc_reverse_generic` and `rtc_reverse_specific` to generate instances that use wrapper classes or concrete iterators respectively. Deriving these classes from `rtc_reverse` guarantees that the compiler will find a base-method for the overrider.

```
// defined in namespace algolib
template <class Wrapper>
struct rtc_reverse_generic : rtc_reverse<typename Wrapper::value_type> {
  using typename rtc_reverse<typename Iterator::value_type>::base_concept;
  typedef typename Wrapper::concept_type iter_concept;

  friend void reverse(virtual iter_concept& begin, base_concept& end) {
    /* wrap begin, end into the matching wrapper class and call std::reverse */
  }
};
template <class Iterator>
struct rtc_reverse_specific : rtc_reverse<typename Iterator::value_type> {
  using typename rtc_reverse<typename Iterator::value_type>::base_concept;
  typedef typename map_iteratortag_to_model<Iterator>::type iter_model;

  friend void reverse(virtual iter_model& begin, base_concept& end) {
    /* unwrap begin, end and call std::reverse with the concrete iterator */
  }
};
```

The analogues to the functions `add_generic` and `add_specific` in §V.A.5 are explicit template instantiation directives.

```
// add generic implementation suitable for all random access iterators.
template class rtc_reverse_generic<wrapper_randomaccess<int> >;

// add specific implementation for std::list<int>.
template class rtc_reverse_specific<std::list<int>::iterator>;
```

The described library functions (§V.A.5) can simply invoke the open multi-method:

```
// defined in the same namespace as the wrapper classes
template <class WrapperType>
void reverse(const WrapperType& begin, const WrapperType& end) {
  algolib::reverse(*begin->iterator_m, *end->iterator_m);
}
```

This call is resolved through the open-method table of the first dynamic argument and invokes the best matching algorithm instance. In contrast to §V.A.5, this involves a simple lookup, similar to a virtual function call, because the open-method tables have been precomputed by the pre-linker.

Ambiguity resolution policy: With the modeled concept hierarchy and the dynamic dispatch restricted to only a single parameter, the ISO `C++` implementation has no ambiguities. An extension towards considering a second argument requires some ambiguity resolution policy. The options for a library implementation are to either signal the ambiguity at runtime or choose somehow a preferred candidate (e.g., higher preference of earlier parameters, arbitrary choice, or user defined). A compiler based implementation has one more possibility: it can flag ambiguities before runtime and require the user to provide a resolving overrider. This is what the open multi-method based implementation does. Only when it is too late for programmer intervention (e.g., in cases that involve dynamic linking), the runtime would make an unspecified non-random choice. This is sufficient to guarantee the correctness of STL's algorithms. An optimal choice between ambiguous overriders would minimize the number of virtual function calls. For an algorithm such as `merge` this would require runtime information of the length of its ranges before a call can be dispatched.

V.A.7. Tests

To assess the performance cost of runtime concepts we tested the approaches described in sections §V.A.3 and §V.A.4. The numbers presented in this section were obtained on an Intel Pentium-D (2.8GHz clock speed; 512MB of main memory at 533 MHz) running CentOS Linux 2.6.9-42. We compiled with gcc 4.1.2 using `-O3` and `-march=prescott`.

V.A.7.a. Algorithm Performance

Initially, the vector contained 8 million numbers in ascending order starting from zero. Then we invoke four algorithms: `reverse`, `find` of zero, `sort`, and `lower_bound` of zero.

`vector<T>`: As a reference point for our performance tests we use `vector` instantiated with a concrete type. Table X shows the number of cycles each operation needs to complete for a container of `int` and `double` respectively. The column to the right of the number of cycles shows the slowdown factor compared to `vector<int>`. The algorithms with $O(n)$ runtime complexity (i.e., `reverse` and `find`) run approximately twice as long when used with type `double`. This discrepancy can be explained by the size of the stored data; `double` is twice as big as `int`.

Table X. Reference test (`vector<int>`)

	`int`	x `vector<int>`	`double`	x `vector<int>`
reverse	50230726	1	101301256	2.0
find	24801042	1	54668768	2.2
sort	554503572	1	1157274566	2.1
lower_bound	5838	1	13544	2.3

Operations on a `Sequence<T>`: Table XI shows the results, when the algorithm library contains instantiations for concrete iterators. The time needed to select the best match is the only overhead that occurs. Note, that compared to [143] we have enhanced the dispatch (by avoiding 3-4 heap allocations), which improved the runtime of `lower_bound` noticeably.

Only when instances for the concrete iterators are missing, does our system

Table XI. Test concrete iterators (`sequence<T>`)

	`int`	x `vector<int>`	`double`	x `vector<int>`
reverse	50635816	1.0	101897418	2.0
find	25428424	1.0	53027609	2.1
sort	551084688	1.0	1173371150	2.1
lower_bound	6244	1.1	14616	2.5

resort to fallback implementations. Table XII reports the runtime of these operations. Note, that the runtime of `lower_bound` is based on an implementation for `wrapper_randomaccess` iterators (cmp. the call to `add_generic` in §V.A.5).

Table XII. Test fallback functions (`sequence<T>`)

	`int`	x `vector<int>`	`double`	x `vector<int>`
reverse	4724088964	94.0	4770263204	95.0
find	474239598	19.1	529099774	21.1
sort	16956331602	30.6	17419730692	31.4
lower_bound	17066	2.9	18522	3.1

The 94x slower performance for `reverse` is unacceptable, even for a fallback implementation. The analysis of these tests reveals three responsible factors: fallback algorithms can be significantly slower (though the slowdown is not always driven by a worse runtime complexity), virtual iterator functions, and model allocation on the heap.

To quantify the contribution of each of these factors we performed additional experiments: Adding a `reverse` instance for `wrapper_randomaccess` improved performance marginally, 92x for int and 94x for doubles. Each iteration of `reverse` calls `iter_swap` once. Gcc's implementation of `iter_swap` calls another function that swaps the two elements to which the iterators point. Each function invocation creates copies of the iterators, which results in 16 million *unnecessary* heap allocations (and deallocations). By providing our own `reverse` implementation, we eliminated those copies. Then, `reverse` is only 7.4x slower for `int` (7.6x for `double`). However, passing arguments by value, which is the source of the heap allocations, is common in STL. For example, the fallback implementation of `sort` involves more than 36 million heap allocations. Instead of rewriting the STL algorithms, we could adopt Adobe's small object optimization [114] where the wrapper classes reserve a buffer to embed small objects (Adobe's open source library [4]).

The analysis of `find` indicates the following causes degrade performance. The fallback implementation is based on the forward iterator concept, while the optimal implementation takes advantage of random access iterators by calculating the trip-count beforehand and manually unrolling the main loop four times. Adding a generic instance for `wrapper_randomaccess` eliminates $\approx \frac{1}{3}$ of the virtual functions calls and improves performance to a factor of 11.7x for `ints` (12.4 for `double`).

V.A.7.b. Library Function Selection

With open multi-methods: Since our experimental compiler extends the EDG frontend [69] and generates C files as output, we modified the test setup. Instead of directly compiling the C++ source, we translate all test cases first to C and compile those with gcc 4.1.2 and the same optimization flags (`-O3` and `-march=prescott`). We measure the dispatch overhead by comparing the performance of static calls to STL functions, with the performance when specific algorithm instances have been

Table XIII. Dispatch performance - cold cache

cold-cache	static STL call	open multi-methods	ISO C++
reverse	585	1731	3786
find	927	3065	3905
sort	7776	7912	8634
lower_bound	377	1994	2512

added to the library. To underscore the time spent for dispatching, we reduced the data size to 8 elements of type `int`. For such a small data set the status of processor cache and predictors make a difference. Therefore we split the tests into two, where the first times an initial algorithm invocation - the dispatch structures are not yet in cache, and the second measures a subsequent invocation of the same four algorithms.

Table XIII shows the number of cycles each call takes. The first call to an algorithm from the runtime library (`reverse`) incurs higher cost as it requires loads of the open method tables/indexes in the case of open multi-methods and RTTI in the case of the ISO C++ implementation. The calls to algorithms that return an iterator (`find` and `lower_bound`) incur additional overhead for allocating the result of the operation (an iterator model) on the heap. The open multi-method based implementation is significantly slower than a static call but outperforms the ISO C++ implementation by 500 to 2000 cycles.

Table XIV shows the results for an an ideal scenario, where all the data (i.e., open method tables/indexes, RTTI, code) is in cache and the processor's predictors can draw from statistical data of previous runs. The overhead of the open multi-method invocation shrinks to less than 100 cycles when compared to a static call (i.e., `reverse` and `sort`. `find` and `lower_bound` require a heap allocation). The ISO

Table XIV. Dispatch performance - hot cache

hot-cache	static STL call	open multi-methods	ISO C++
reverse	157	220	445
find	158	1008	1181
sort	5631	5725	5934
lower_bound	224	1159	1209

C++ implementation is about 200 cycles slower.

V.A.8. Related Work

The ASL [4] introduced the runtime concept idiom, employing type erasure [1] to provide the *any regular* library (similar to the boost any library), and its generalization, the *poly* library. The *poly* library generalizes the idiom to support refinement and polymorphic down-casting, encapsulates the common tasks required to create non-intrusive runtime-polymorphic value-based wrappers. The *poly* library design goals and implementation are elaborated in [114].

ASL also provides the *any iterator* library offering runtime-polymorphic iterators for specific types as a proof of concept. Becker [28] presents a similar library. Bourdev and Järvi [33] discuss a mechanism for a closed set of types that falls back to static-dispatch when type erasure is present.

Our work extends the previous results to an open library of algorithms operating on runtime-polymorphic containers, achieving realistic performance levels by using static dispatch where possible.

V.B. Comparison of Programming Styles

In order to compare open-methods with double dispatch and the visitor pattern, we have implemented some of the examples discussed in §IV.B.

V.B.1. Image Format Conversion

The first example is image conversion. To meet the performance requirements typical for image processing applications, information about the exact source and destination formats is indispensable for an efficient conversion. With this information, we can call a routine geared for that specific pair of formats. Any attempt to work through a common base interface will significantly hinder performance, and should be avoided. This is why we use a fairly shallow class hierarchy to represent different image formats. Another interesting aspect of this example is that when the pair of formats is known statically, it is feasible to write a generic conversion algorithm that relies on some format traits. This is an approach taken by Adobe's GIL library [33]. Therefore the main goal in this example is to uncover the dynamic types of both arguments and pass on these uncovered arguments together with their static types to a set of overloaded template functions.

```
template <class SrcImage, class DstImage>
bool generic_convert (const SrcImage& src, DstImage& dst);
typedef unsigned char color_component;
struct image
{
  // member−functions to access row buffer, width, height etc.
};
struct RGB : image // abstract base of all RGB images
{
  struct color { color_component R, G, B, A; };
  virtual color get_color (int i, int j) const          = 0;
  virtual void  set_color (int i, int j, const color& c) = 0;
};
```

```
struct RGB32 : RGB { /*implements get_color, set_color */ };
// ... Similar definitions for RGB24, RGB16, RGB15, RGB08
struct YUV : image // abstract base of all YUV images
{
  struct color { color_component Y, U, V, A; };
  virtual color get_color (int i, int j) const           = 0;
  virtual void set_color (int i, int j, const color& c) = 0;
};
struct UYVY : YUV { /*implements get_color, set_color*/ };
// ... Similar definitions for YUY2,Y41P,CLJR,YVU9,YV12,IYUV,I420,Y800 etc.
struct CMYK : image
{
  struct color { color_component C, M, Y, K; };
  virtual color get_color (int i, int j) const           = 0;
  virtual void set_color (int i, int j, const color& c) = 0;
};
```

Open multi-methods to handle the cases can be listed separately from class definitions:

```
// Base open−method. Fails as we do not know anything about the formats
bool convert(virtual const image& src, virtual image& dst) { return false; }
// Slow polymorphic conversions
bool convert(virtual const RGB& src, virtual RGB& dst);
bool convert(virtual const RGB& src, virtual YUV& dst);
bool convert(virtual const YUV& src, virtual RGB& dst);
bool convert(virtual const YUV& src, virtual YUV& dst);
// Fast generic conversions, generated for each combination of types
bool convert(virtual const RGB32& src, virtual RGB32& dst)
{
  return generic_convert(src, dst);
}
bool convert(virtual const RGB32& src, virtual RGB24& dst)
{
  return generic_convert(src, dst);
}
bool convert(virtual const RGB32& src, virtual YUY2& dst) { /* ... */}
bool convert(virtual const RGB32& src, virtual YVU9& dst) { /* ... */}
bool convert(virtual const RGB32& src, virtual I420& dst) { /* ... */}
```

In case of double dispatch, the code becomes cluttered with definitions to support the mechanism:

```
// Forward declare all classes that would participate in double dispatch
struct RGB32;
struct RGB24;
// ... others
struct image
{
  // member−functions to access row buffer, width, height etc.
  // Double dispatch support code
  virtual bool convert_to(image& dst) const = 0;
  virtual bool convert_from(const RGB32 & src) { return false; }
  virtual bool convert_from(const RGB24 & src) { return false; }
  virtual bool convert_from(const RGB16 & src) { return false; }
  // ... etc. for all other leaf image classes
};
struct RGB32 : RGB
{
  virtual bool convert_to(image& dst) const    { return dst.convert_from(∗this); }
  virtual bool convert_from(const RGB32 & src)
  {
    return generic_convert (src, ∗this);
  }
  virtual bool convert_from(const RGB24 & src)
  {
    return generic_convert (src, ∗this);
  }
  // ... etc. for all other leaf image classes
};
```

The major disadvantage of the double dispatch approach is that we have to foresee the whole hierarchy at the moment we are defining its root. This is necessary for declaring the interface for uncovering types. Once it is defined, we cannot extend it for newly created classes - they will all be treated as their closest ancestor in the hierarchy. Another problem with double dispatch is that its supportive structures clutter the code. This may be acceptable when double dispatch is needed for only one algorithm, but when several algorithms require it (e.g., we would also like to have a polymorphic `bool compare(virtual const image& a, virtual const image& b)`) then the code may quickly get out of hands. While this aspect of the double dispatch can be solved with the visitor pattern at the cost of two extra virtual calls, the open-method solution will remain cleaner as open-methods do not even need to be defined together with the class. We discuss the visitor pattern in greater detail in our second example.

The number of lines in the implementation with open methods was smaller, but all in all, the number of lines in both implementations is growing as square of the number of classes in the hierarchy. We note that for open multi-method implementations this is a rather exceptional case, because the class hierarchy was shallow, while we were interested in uncovering all possible type combinations. For the double dispatch, this is rather typical case because the supportive definitions will have to be there anyway.

In the image conversion example the main purpose of the open-methods is to discover the dynamic types of both arguments and then forward the call to an overloaded function. Templated open-methods, as described in §V.A.6, further simplify the design:

```
// Base open-method. Fails as we do not know anything about the formats
bool convert(virtual const image& src, virtual image& dst) { return false; }
// Slow polymorphic conversions
bool convert(virtual const RGB& src, virtual RGB& dst);
bool convert(virtual const RGB& src, virtual YUV& dst);
bool convert(virtual const YUV& src, virtual RGB& dst);
bool convert(virtual const YUV& src, virtual YUV& dst);
// Fast generic conversions, generated for each combination of types
template <class Source, class Destination>
bool convert(virtual const Source& src,virtual Destination& dst)
{
  return generic_convert(src, dst);
}
```

V.B.2. AST Traversal

The second example discusses the use of open-methods to traverse ASTs. The key focus thereby is on extending classes with open dispatch rather than multiple dispatch. Open-methods essentially become virtual functions that can be added to a class after it has been defined. The examples in this section reflect our experience of writing an analysis pass for the Pivot source-to-source transformation infrastructure §III.A. The Pivot uses the visitor pattern to type safely uncover the dynamic type of AST nodes. The Pivot consists of approximately 150 classes, but in the ensuing discussion, we limit the AST hierarchy to only two of them, where one `Expr` is a base class for all kinds of expressions, and the other `Unary` is an implementation of unary expressions.

```
struct Expr
{
  ...
  virtual accept(Visitor& v) const { v.visit(*this); }
};
struct Unary : Expr
{
  ...
  accept(Visitor& v) const { v.visit(*this); }
};
struct Visitor
{
  void visit(const Expr&) = 0;
  void visit(const Unary&) = 0;
};
```

Forwarding calls to base implementations: Currently, the Pivot has about 160 node types. The Pivot provides a number of intermediate abstract base classes that factor commonalities (e.g., `Expr`, `Type`, `Declaration`, etc.) of the 160 node types. If the logic of the visitor can be implemented in terms of a single base class, the bodies of the more specific types will need to explicitly invoke the base implementation (compare to the implementation for `Unary`). Open-methods have this forwarding behavior by default.

```
struct SimpleVisitor : Visitor
{
  virtual void visit(const Expr& e) { /* ... */ };
  virtual void visit(const Unary& u)
  { visit(static_cast<Expr&>(u)); // calls visit(Expr&)
  }
};
// All objects derived from Expr will be handled by simpleOpenMethod
void simpleOpenMethod(virtual const Expr&) {/* ... */}
void foo(Expr& e, Unary& u)
{
  SimpleVisitor vis;
  e.accept(vis); // invokes SimpleVisitor::visit(const Expr&)
  u.accept(vis); // invokes SimpleVisitor::visit(const Unary&) first
  simpleOpenMethod(e); // invokes simpleOpenMethod(const Expr&)
  simpleOpenMethod(u); // invokes simpleOpenMethod(const Expr&)
}
```

Passing of arguments and results: The signatures of the visit and accept functions are determined when the visitor and the AST node classes are defined. Passing additional arguments to (or returning a value from) the visit functions requires intermediate storage as part of the visitor class. In the following example, `inh` and `syn` correspond to the input and result values of a function.

```
struct AnalysisPassVisitor : Visitor
{
  const InheritedAttr & inh; // data member for input parameter
  SynthesizedAttr *     syn; // data member for return value
  Visitor (const InheritedAttr & inherited ) : inh(inherited ), syn() {}
  ...
};
```

To avoid code duplication, it is useful to factor constructing the visitor and reading out the result into separate functions.

```
SynthesizedAttr * visit_foo (const Expr& e, const InheritedAttr & inh)
{
  AnalysisVisitor  v(inh); // construct the visitor  and pass the context
  e.accept(v);
  return v.syn;            // read and return the result
}
```

With open-methods, additional arguments can easily be specified as part of their signatures.

```
SynthesizedAttr* analysisPass(virtual const Expr& e, const InheritedAttr& inh);
```

Covariant return type: Since the result requires intermediate storage, covariant return types cannot easily be implemented with the visitor pattern. Consider the following implementation of a visitor that creates and returns a copy of an AST node.

```
struct CloneExpr : Visitor
{
  Expr* result ; // data member for return value
  // make a copy of an Expr object
  virtual void visit (const Expr& e) { result = new Expr(e); }
  // make a copy of an Expr object
  virtual void visit (const Unary& u) { result = new Unary(u); }
};
Expr* clone(const Expr& e) // analog of a base−method
{
  CloneExpr v;
  e.accept(v);
  return v.result ;
}
```

Cloning a unary expression loses some type information, because the cloned objects would get returned as `Expr`. An implementation that is able to return covariant types requires a different visitor implementation (or instantiation) with similar boilerplate code for each covariant return type. These repetitive definitions can be eliminated by using templates. The following example shows a cloning visitor that returns a `Unary` object.

```
struct CloneUnary : Visitor
{
  Unary* result ; // data member for covariant return value
  virtual void visit (const Expr& e) { assert (false); } // Can never be called !
  virtual void visit (const Unary& u) { result = new Unary(u); }
};
Unary* clone(const Unary& u) // analog of an overrider  with covariant  return  type
{
  CloneUnary v;
  u.accept(v);
  return v.result ;
}
```

The definition of open-methods with covariant return types is straight forward:

```
Expr& clone(virtual const Expr& a); // base−method
Unary& clone(virtual const Unary& u); // overrider with covariant return type
```

Similar to open-methods, a hand crafted technique for modeling covariant return type with visitors also creates additional "dispatch tables" for each overrider with covariant return type. Those are created in a form of v-tables for additional visitors. Interestingly enough, the overall size of such dispatch tables would be larger than those generated for open-methods, because visitors require v-table entries for visit methods that can never occur at runtime (see the assert in `CloneUnary::visit(Expr&)` above). This is not the case with open-methods as overriders with covariant return type will operate on smaller class hierarchies for their arguments.

Passing open-methods as callbacks: Open-methods nevertheless have disadvantages sometimes in comparison with the visitor pattern. Consider a traversal mechanism that traverses an AST in certain order. The mechanism can accept either a visitor or an open-method for node visitation:

```
void evaluation_order_traversal   (const Expr& e, Visitor & v);
void evaluation_order_traversal   (const Expr& e, void(*fn)(const Expr&));
```

In case of a visitor, we can pass or accumulate some data during visitation. However in case of open-method, we would need to accumulate data elsewhere.

```
struct CodeGenerationVisitor : Visitor
{
  std::vector<Instruction> instructions; // instruction stream inside visitor
  void visit(const Expr& e) { /* generate code for e*/ }
  // ...
};
CodeGenerationVisitor v;
evaluation_order_traversal(root_node(),v);
```

Although our current implementation does not support taking the address of an open-method, we can simulate that behavior by wrapping the open-method call inside a thunk.

```
std::vector<Instruction> instructions; // global instruction stream
void generate_code(virtual const Expr&  e) { instructions.push_back(...); }
void generate_code(virtual const Unary& e) { instructions.push_back(...); }
//...
void thunk_generate_code(const Expr& e) { generate_code(expr); }
evaluation_order_traversal(root(), &thunk_generate_code);
```

V.C. Comparison with Other Implementation Techniques

In order to discuss time and space performance, we compare code generated by our C++ Open Method Compiler, described in §**??**, to a number of prototype implementations, the visitor pattern, Cmm, DoubleCpp, and the Loki library. The prototypes of our design alternatives were implemented in C to approximate the lowering of C++ code to C. They were initially developed to assess performance trade-offs of different approaches and workaround techniques and include open-methods (can be declared freely), multi-methods (have to be declared in class, thus the om-tables can be embedded into the v-table, saving two indirections per argument §IV.F.3.a.), and a Chinese Remainder (§IV.F.3) based implementation. These implementations layout the dispatch table for the concrete example described below as C data structures, and then dispatch calls through it.

We wrote 20 classes (representing shapes, etc.) that can intersect each other. Overall, this results in 400 combinations for binary dispatch functions. We implemented 40 specific intersect functions to which all of the 400 combinations are dispatched. In order to get a reliable timing of the function invocation, these 40 intersect functions only increment a counter. Since not all techniques we use support multiple inheritance, these 20 classes only use single inheritance. The actual test consists of a loop that randomly chooses 2 out of 32 objects and invokes the intersect method. We implemented a table-based random number generator that is simple and does not contain any floating-point calculations or integer-divisions. We ran the loop twice with the same random numbers: The first run allows implementations that build the dispatch data structure on the fly to warm up and load data/code into the cache. The second loop was timed. The clock-cycle based timer takes the time before and after the loop and we calculate the average number of clock-cycles per loop to compare the results.

V.C.1. Implementations

We tested the approaches on a Pentium D, 2.8 GHz running CentOS Linux and a Core2Duo running Mac OSX. The code for the performance tests was compiled with g++ 4.1 (Linux) and gcc 4.0.1 (OSX) with optimization level set to -O3. The C++ Open Method Compiler generates source code lowered to C, which was compiled with the corresponding gcc versions and linked to the pre-linker generated dispatch tables.

Using the Chinese Remainder approach, the number associated with the dispatch table grows exponentially with the number of types. Therefore the test is limited to eight types instead of 20 and the size of the executable is omitted.

For Loki, we only tested the static dispatcher because the others require manual handling of all possible cases. Using other dispatchers would have been closer to a scenario of a manually allocated array of functions through which calls are made. However, as we indicated before, the dual nature of multi-methods require them to provide both dynamic dispatch and automatic resolution mechanism.

Table XV. Experimental results

Approach	Size (bytes) Linux	Cycles (per loop) Pentium-D	Cycles (per loop) Core2Duo
Virtual function	n/a	75	55
Multi-methods prototype	42 972	78	60
Open-methods prototype	40 636	82	63
C++ Open-method Compiler	42 504	82	64
Double Cpp	34 812	120	82
C++ Visitor	38 236	132	82
Chinese Remainders prototype	n/a	175	103
Cmm (constant time)	155 344	415	239
Cmm	155 056	1 320	772
Loki Library	75 520	3 670	2 238

V.C.2. Results and Interpretation

Table XV shows a summary of our experimental results for execution time and program size.

Executable size: To obtain a comparable size of the executable, we used the regular EDG frontend to generate C code for the alternative approaches. Then we compiled all intermediate C files with gcc, where optimizations were set to minimize code size. Moreover, we stripped off the symbols from the executables. The size of dispatch tables is mentioned as one of the major drawbacks of providing multi-methods as programming language feature [195]. However, our results reveal that the best achievable code size is roughly the same for visitors, prototyped multi-/open-method, and C++ Open-method Compiler implementations. With the visitor, each shape class has intersect methods for all 20 shapes of the hierarchy. A somewhat smarter approach would be to remove redundant intersect overriders. However, removing specific overriders is tedious and difficult to maintain, since the dispatch would be based on the static type information of the base class. Even an optimized approach would require as many v-table entries as there are in a dispatch table, simply because each type contains 20 intersect entries in the v-table. Multiplying this with the number of shapes, 20, results in 400, exactly the number of entries found in the dispatch table. We do not discuss the program size of the two Cmms and Loki, since they use additional header files, such as `<typeinfo>` and `<stdexcept>` that distort a direct comparison.

Execution time: The results for prototyped multi-methods, prototyped open-methods, and C++ Open-method Compiler are (as expected) roughly comparable to a single virtual function dispatch, which needs 75 (55 on the Core2Duo) cycles per loop. Hence, the better performance compared to the visitors is not surprising. However, the fact that multi-methods reduce the runtime to 62% (73%) of the reference implementation using the visitor is noteworthy. We conjecture this is an effect of the size of the class hierarchy and that the time to double dispatch depends on the number of overriders. On the Pentium D, two observations support our conjecture: firstly, the DoubleCpp-based visitor has no redundant overriders and runs slightly faster. Secondly, we simulated an analysis pass dispatching over AST-objects of 20 different types and counting the category to which they belong (type, declaration, expression, statement, other). In this case, the double dispatch has only 20 leaf-functions instead of 400 and our dispatch test runs 78 cycles instead of 132. The open-method approach requiring only five overriders, is still faster and needs 68 cycles.

The difference between the prototyped multi-methods and open-methods (the comparison with the C++ Open-method Compiler is stated in parenthesis) is within the expected range. Four more indirections require 4 (4) more clock cycles on the Pentium and 3 (4) more on the Core2Duo. Although significantly slower, Cmm (constant time) performs better than expected, since its author estimates the dispatch cost as 10 times a regular virtual function call. As expected the two non-constant time approaches perform worst.

Significance of performance: The performance numbers come from experiments designed to highlight the cost of multiple dispatch: the functions invoked hardly do anything. Depending on the application the improved performance may or may not be significant. For the image conversion example, gains in execution speed are negligible compared to time spent in the actual conversion algorithm. In other cases, such as the evaluation of expressions using user-defined arithmetic types, traversal of abstract syntax trees, and some of the most frequent shape intersect examples, the speed differences among the double dispatch approaches appear to be notable.

Contrary to much "popular wisdom", our experiments revealed that for many applications the use of dispatch tables for open-methods and multi-methods actually reduce the program size compared to brute-force and work-around techniques. Under the assumption that the use of open-methods in C++ would be similar to Muschevici et al.'s results (see §II.B), we conclude that the size of the dispatch table will remain small for most practical cases.

CHAPTER VI

RECOVERY OF HIGHER LEVEL LANGUAGE ABSTRACTIONS

This section discusses the interaction of the tools described in (Chapter III) to find code that is expressible by a higher level language feature or library function. We start by analyzing a higher level abstraction: initializer lists. We describe the semantics of the language feature and how it can be approximated by workaround codes. We formulate the recovery techniques by using initializer list constructor used by C++0x containers as the canonical example.

VI.A. Semantics and Workaround Definition

Initializer list constructors (§I.B.1, [3], [120]) allow the specification of initial values that will be stored in the data structure. For example:

```
// initializes a vector with three ints
std::vector<int> v1 = { 1, 2, 3 };
// constructs a vector with three elements of type SomeType
// SomeType has to be constructable from an int
std::vector<SomeType> v2 = { 1, 2, 3 };
```

The first case (`v1`) creates a `vector` with three `int` values. The second case (`v2`) creates a `vector` with three `SomeType` values, where each element is created by a constructor that takes an argument of type `int`.

After the construction, both containers hold three elements. In current C++, there is a single technique allowing to construct a container with a number of different elements. This technique relies on an auxiliary container (e.g., C style array) to pass a range of elements (e.g., the position of the first element and the position one past the last element) to a constructor.

```
// C++03 uses range construction
static const int arr[] = { 1, 2, 3 };
```

```
std::vector<SomeType> v2 (arr, arr+sizeof(arr)/sizeof(arr[0]));
```

Numerous other workaround techniques exist. Besides a sequence of `push_back` operations mentioned in the introduction §I.B.1, programmers can resort to various combinations. For example, code can use STL's algorithms (e.g., `for_each` in combination with a `back_inserter`) or a loop to copy elements from a static array or a combination of both. Other variations include member variables that have an STL container type (they are declared at class scope, initialized as part of the constructor member initialization list, and potentially filled with values in the constructor body), global containers, or classes that (privately) inherit from a STL container.

VI.A.1. Semantic Equality

For any kind of rejuvenation analysis, the question arises whether the replacement of the workaround code produces semantically equivalent code. The argument to establish the validity of a rejuvenation can rest on several models:

Strict state equivalence: One strategy to establish that the source codes before and after the rejuvenation are semantically equal is to compare the program state after the execution of a rejuvenated code region. In addition to directly involved objects, the program state also depends on operations with side effects on memory management and IO system. As a consequence, strict state equivalence allows rejuvenations that have no observable impact. To establish strict state equivalence a priori, a system requires that semantic guarantees can be derived from the ISO language standard.

Consider the rejuvenation that replaces C++03 style range construction with a C++0x initializer list. (Only) if the implementation of the range and initializer list constructors use the same underlying implementation, the result is state equivalent.

Relaxed state equivalence: Often strict equivalence curtails the applicability of rejuvenations. Relaxing requirements on some observable side effects allows *behavior improving* transformations. Consider code that constructs a vector and pushes back a number of elements. If the capacity of the vector is not properly adjusted, the sequence of `push_back` might trigger a resize. Resize entails dynamic memory allocation and the relocation of elements. Rejuvenating this code towards initializer lists eliminates these intermediate steps. The result is a changed memory allocation pattern (less allocations and lower likelihood of memory fragmentation). The less frequent need to copy elements is observable by user defined copy constructors (e.g., to count the number of constructed objects). Rejuvenation might change the final state of an object. For example, the use of an initializer list provides the constructor better size information, which a data structure could use to optimize memory management.

We argue that these optimizations are valid in the context of migrating source code to a newer edition of a programming language:

- code where the correctness depends on the memory management strategy of a standard library implementation is inherently fragile. Changing the library implementation, which is nothing extraordinary during programming language evolution, would invalidate such code.

- C++ allows code optimizations if they do not modify a program's behavior [27]. In order to avoid unnecessary overhead from copying objects, compilers can elide calling copy constructors in several contexts. Moreover, the C++0x will introduce rvalue constructors (to enable move semantics) [27]. Consequently, implementations of standard containers will rely less on copy constructors.

Unknown state equivalence: It is also desirable to detect source code that looks as if it could benefit from rejuvenation, but where the semantics' preservation cannot be guaranteed by the program. These cases need to be judged by a human software maintainer. An example for such a rejuvenation would be code transformations towards C++0x's `for` range loop:

```
void forall_books(std::vector<Book>& vec)
{
  for (Book& book : vec)
    /* ... */;
}
```

For this code, the C++0x standard describes its semantics with the following loop.

```
void forall_books(std::vector<Book>& vec)
{
  for ( std::vector<Book>::iterator first = vec.begin(), last = vec.end();
        first != last; ++first)
    /* ... */;
}
```

The loop is turned into a regular `for` loop, where the iterator `first` is incremented with the prefix `++` operator. Likewise, source code that use a different syntax (e.g., `first++`, `first+=1`) to increment the iterator might also be a rejuvenation candidate. For such cases, static analysis cannot always guarantee semantic equivalence (e.g., when virtual functions are involved §V.A.1).

VI.B. Workaround Recognition

For the implementation of the initializer list rejuvenation, we utilize the tools described in (Chapter III). In particular, we use the pattern matcher to specify "micro-patterns" that describe portions of a workaround idiom. We use a state machine to determine whether the recognition sequence of these micro patterns constitute a proper workaround pattern.

VI.B.1. Patterns of Initializer List Construction

This section describes the micro-patterns that we utilize to recover initializer lists.

Types: The rejuvenation's goal is the recovery of STL container constructions that can benefit from initializer lists. The following pattern defines the STL container types that we analyze.

```
pattern sequence_container<Type>(ilist : &InitializerList)
  alias : Alias();
{
  /*1*/ vector<|...|> => ;
  /*2*/ list<|...|> => ;
  /*3*/ e :: sequence_container(ilist)
     => /* skips any possible namespace prefix */;
  /*4*/ alias
     => { if (!is_sequence_container(*alias->aliasee)) { NOMATCH; }; else {}; };
}
```

The pattern describes the type as it is written in the source code. The first two lines describing an instantiation in XPR, where the template arguments are specified within the `<|` and `|>` tokens. The `...` indicate that any sequence of arguments matches the pattern. Potential name namespace prefix are skipped by line 3. Container instantiations that are obscured with `typedefs` can be skipped as indicated by (4). Type qualifiers can be treated likewise.

Object Constructors: The first code snippet defines code that recognizes container construction. The pattern uses a user-defined object `ilist`. The `InitializerList` implements a simple state machine to determine whether a given sequence makes a valid initializer list constructor. Note that with the pattern match generator it is currently not possible to specify the absence of an initializer. Thus, the distinction between default and other constructors is encoded inside the action block (i.e., call to `uses_default_ctor`).

```
pattern construction<Stmt>(ilist : &InitializerList)
  default_ctor : Var(..., sequence_container(ilist));
  range_ctor : Var(..., sequence_container(ilist), range_initializer(ilist));
{
  default_ctor;
     => { if (uses_default_ctor(v)) { ilist.ctor(v); } else { NOMATCH; } };
  range_ctor;
    => { ilist.range_ctor(range_ctor); }
}
```

The `range_initializer` describes the initializer consisting of two pointers to an array.

```
pattern range_initializer<Expr_list>(ilist : &InitializerList)
  arr1 : Var();
  arr2 : Var();
  t : Type();
{
  array_base(arr1), arr2 + array_ofs(arr2, t, ilist)
    => { ilist.range(arr1, arr2, t); };
  array_base(arr1), &arr2[array_ofs(arr2, t, ilist)]
    => { ilist.range(arr1, arr2, t); };
}
```

The following code gives the definition of `array_base` and `array_ofs`. The latter recognizes two variants (1,2) of specifying the end of an array. It could be extended to include any arbitrary expression (or constant value).

```
pattern array_ofs<Expr>(arr : Var, t : Type, ilist : &InitializerList)
  e : Expr();
{
  /*1*/ sizeof(arr) / sizeof<|t|>() => { ilist.array_end(arr, t); };
  /*2*/ sizeof(arr) / sizeof(arr[e]) => { ilist.array_end(arr, e); };
  /*3*/ ( array_ofs(arr, t, ilist) ) => ; /* skips parenthesis */
}
pattern array_base<Expr>(arr : Var)
  a : Var(..., Array());
{
  a => { arr.set(a); };
  ( array_base(arr) ) => ; /* skips parenthesis */
}
```

Coding patterns: This following code snippet defines the expression patterns we use for describing a workaround technique (sequence of `push_back` operations). The first two alternatives recognize calls to `reserve` and `push_back` respectively and forward the data to the `ilist` state machine. Line (3) defines cases when the container is found in any other context; the `ilist` state machine interprets such a case as the end of the initialization sequence. Line 4 recognizes any other expression. The `ilist`'s function `expr` traverses `e`'s sub expression and matches them against the expression patterns defined by `pat_ilist`.

```
pattern pat_ilist<Expr>(ilist : &InitializerList)
  reserve : Fundecl("reserve");
  push_back : Fundecl("push_back");
  cont : Var();
  e : Expr();
{
  /* 1 */ reserve(&cont, e) => { ilist.reserve(reserve, cont, e); };
  /* 2 */ push_back(&cont, e) => { ilist.push_back(push_back, cont, e); };
  /* 3 */ cont => { ilist.var_use(cont); };
  /* 4 */ e => { ilist.expr(e); };
}
```

VI.B.2. The State Machine

To validate that the combination of recognized micro patterns form a workaround pattern, we use a state machine. Each recognized workaround pattern leads to a state transition. Fig. 14 shows the state machine that recognizes the sequences of `push_back` statements.

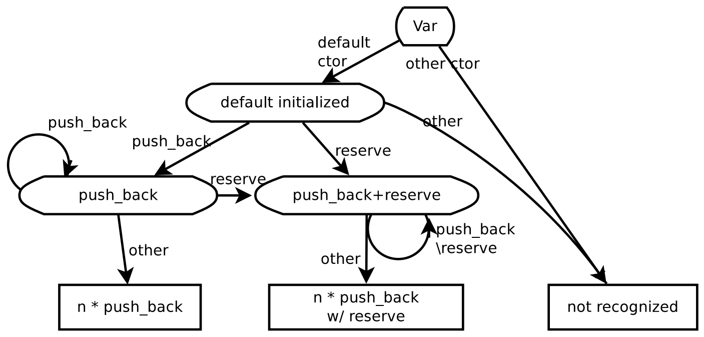

Fig. 14. Statemachine recognizes initializer list workaround

VI.C. Results

Our actual implementation extends the presented version and recognizes more variations of initializer list workarounds, such as calls to `push_back` in simple `for` loops, and the use of STL containers as data members. In addition, our implementation reports code where two pointers and a newly constructed container are directly or indirectly (e.g., as member of `back_inserter`) passed to function calls.

The pattern definition file is 69 lines of code (loc). The generated pattern matching code is about 700 loc. The remaining project (i.e., state machine, binding of the traversal framework to the pattern matching code, and analysis of function call's arguments) consists of 700 hand written lines of code.

VI.C.1. Tests

Besides synthetic tests, we ran our rejuvenation tool on four arbitrarily chosen open source projects that use STL containers: Eboard (v1.1.1, an internet chess client based on GTK, available from http://www.bergo.eng.br/eboard/), WxCam (v1.0.4, an application that records video streams based on the WxWidget library, http://wxcam.sf.net/), Fluxbox (v1.0rc3, a window manager, http://www.fluxbox.org/), and Referencer (v1.1.6, a bibliography management system based on GNOME, http://icculus.org/referencer/). Our application found code in Fluxbox and Referencer that can benefit from the C++ initializer list extension. Fluxbox consists of about 38,200 LOC (not counting standard and third-party library header files). It contains three instances where a STL container is initialized with user-interface elements by using between 8 and 17 consecutive `push_back` operations. Referencer's source code has about 16,000 LOC and our implementation found a map being initialized with 22 predefined key/value pairs. For Eboard (33,600 LOC) and WxCam (10,400 LOC) our tool did not find any code that can be modeled with initializer lists.

VI.C.2. Discussion

Using a pattern matcher simplifies the implementation. By automatically writing boilerplate code, the generator uses well studied techniques for constructing a pattern recognition engine. This eliminates time and error prone user code development.

The use of a high level source code representation poses a number of challenges for analysis programs. Workaround idioms can be written in a wide variety of forms. With the pattern matcher we can address these problems to some degree. Extra case handling can deal with most common variations, such as parenthesized expressions, numerous forms of getting the beginning address of an array, or type aliases. Preparing for all *unusual* styles and variations becomes cumbersome. Consider the many variations programmers can express loops (e.g., several syntactic constructs multiplied by several variations). The pattern definition of such diverse implementation technique would benefit from a lowering pass that generates canonical tree representations. Alternatively, the pattern matcher could be extended to generate equivalent patterns from a base definition.

Moreover, pattern matching on a source level tree representation does not lean itself well towards data flow sensitive analysis. Bringing the representation into a canonical form, for example through symbolic expression propagation, can address some of these problems. Integration with dataflow frameworks is another way to extend the current implementation.

CHAPTER VII

REJUVENATION OF LIBRARY USE

Multi-core architectures are state-of-the-art on new personal computers (and some embedded systems) with the number of available cores per system bound to increase rapidly in the foreseeable future. Programming parallel software is non-trivial because concurrency entails many hazards including race conditions, deadlocks, livelocks, order violations, software lockout, and atomicity violations. Providing safe and efficient concurrent synchronization is important for the design of real-time systems. Lock-free programming techniques [93] have been demonstrated to be effective in delivering performance gains and preventing some hazards, typically associated with the application of mutual exclusion. In this section, we develop a non-blocking, growable array that can replace STL vectors that are shared by multiple threads in concurrent software and whose access is protected by fine grained mutual exclusion locks.

VII.A. Introduction

Developers writing parallel and multi-threaded code face challenges not known in sequential programming: notably to correctly manipulate data where multiple threads access it. Currently, the most common synchronization technique to guard critical sections is mutual exclusion locks. A mutual exclusion lock guarantees thread-safety of a concurrent object by blocking all contending threads except the one holding the lock. This can seriously affect the performance of the system by diminishing its parallelism. The behavior of mutual exclusion locks can sometimes be optimized by using fine-grained locks [95] or context-switching.

The use of non-blocking (lock-free) techniques has been suggested to prevent the interdependence of the concurrent processes introduced by the application of locks [90].

VII.A.1. Non-Blocking Synchronization

As defined by Herlihy [90, 91], a concurrent object is *non-blocking* if it guarantees that *some* process in the system will make progress in a *finite* number of steps. An object that guarantees that *each* process will make progress in a *finite* number of steps is defined as *wait-free*. Non-blocking (lock-free) and wait-free algorithms do not apply mutual exclusion locks. Instead, they rely on a set of atomic primitives such as the word-size CAS instruction. Common CAS implementations require three arguments: a memory location, Mem, an old value, V_{old}, and a new value,V_{new}. The instruction atomically exchanges the value stored in Mem with V_{new}, provided that its current value equals V_{old}. The architecture ensures the atomicity of the operation by applying a fine-grained hardware lock such as a cache or a bus lock (e.g.: IA-32 [94]). The use of a hardware lock does not violate the non-blocking property as defined by Herlihy. Common locking synchronization methods such as semaphores, mutexes, monitors, and critical sections utilize the same atomic primitives to manipulate a control token. Such application of the atomic instructions introduces interdependencies of the contending processes. In the most common scenario, lock-free systems utilize CAS in order to implement a speculative manipulation of a shared object. Each contending process speculates by applying a set of writes on a local copy of the shared data and attempts to CAS the shared object with the updated copy. Such an approach guarantees that from within a set of contending processes, there is at least one that succeeds within a finite number of steps.

VII.A.2. Non-Blocking Data Structures

Recent research into the design of lock-free data structures includes linked-lists [87, 123] double-ended queues [124, 178], stacks [89], hash tables [123, 156] and binary search trees [74]. The problems encountered include excessive copying, low parallelism, inefficiency and high overhead. Despite the widespread use of the STL vector in real-world applications, the problem of the design and implementation of a lock-free dynamic array has not yet been discussed. The vector's random access, data locality, and dynamic memory management poses serious challenges for its non-blocking implementation. Our goal is to provide an efficient and practical lock-free STL-style vector.

VII.A.3. A Non-Blocking Vector

The vector is the most versatile and ubiquitous data structure in the C++ STL [171]. It is a dynamically resizable array that provides automatic memory management, random access, and tail element insertion and deletion with an amortized cost of O(1).

Our non-blocking resizable array implementations [57] are based on a single 32-bit word atomic compare-and-swap (CAS) instruction. It provides a linearizable and highly parallelizable STL-like interface, lock-free memory allocation and management, and fast execution. Our current implementation is designed to be most efficient on multi-core architectures. The presented design implements the most common STL vector's interfaces, namely random access read and write, tail insertion and deletion, pre-allocation of memory, and query of the container's size. The main target of our design is to deliver good performance for such systems (§VII.C).

VII.B. Implementation

The major challenges of providing a lock-free vector implementation stem from the fact that key operations need to atomically modify two or more non-colocated words. For example, the critical vector operation `push_back` increases the size of the vector and stores the new element. Moreover, capacity-modifying operations such as `reserve` and `push_back` potentially allocate new storage and relocate all elements in case of a dynamic table [54] implementation. Element relocation must not block concurrent operations (such as `write` and `push_back`) and must guarantee that interfering updates will not compromise data consistency. Therefore, an update operation needs to modify up to four vector values: size, capacity, storage, and a vector's element.

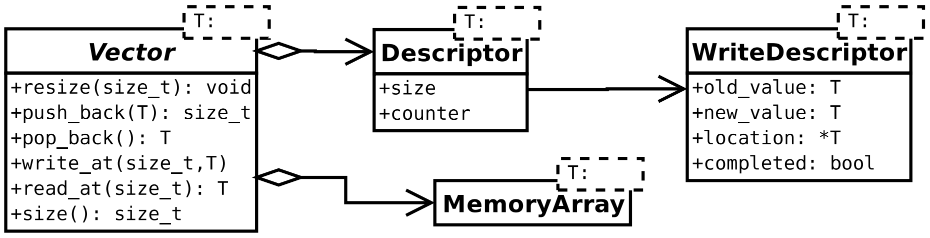

Fig. 15. Lock-free vector, T denotes a data structure parameterized on T.

The UML diagram in Fig. 15 presents the collaborating classes, their programming interfaces and data members. Each vector object contains the memory locations of the data storage of its elements as well as an object named "Descriptor" that encapsulates the container's size, a reference counter required by non-blocking reference scheme ([126]) and an optional reference to a "Write Descriptor". Our approach requires that data types bigger than word size are indirectly stored through pointers. Like Intel's concurrent vector [153], our implementation avoids storage relocation and its synchronization hazards by utilizing a two-level array. Whenever `push_back` exceeds the current capacity, a new memory block twice the size of the previous one is added.

The semantics of the `pop_back` and `push_back` operations are guaranteed by the "Descriptor" object. The use of a "Descriptor" allows a thread-safe update of two memory locations (Barnes-style announcement [21]) thus eliminating the need for a DCAS instruction. An interrupting thread intending to change the descriptor will need to complete any pending operation. Not counting memory management overhead, `push_back` executes *two successful CAS* instructions to update *two memory locations.*

VII.B.1. Implementation

Table XVI illustrates the implemented operations as well as their signatures, descriptor modifications, and runtime guarantees.

Table XVI. Vector - operations

	Operations	Descriptor (Desc)	Complexity
push_back	$Vector \times Elem \rightarrow void$	$Desc_t \rightarrow Desc_{t+1}$	$O(1) \times cong.$
pop_back	$Vector \rightarrow Elem$	$Desc_t \rightarrow Desc_{t+1}$	$O(1) \times cong.$
reserve	$Vector \times size\_t \rightarrow Vector$	$Desc_t \rightarrow Desc_t$	$O(1)$
read	$Vector \times size\_t \rightarrow Elem$	$Desc_t \rightarrow Desc_t$	$O(1)$
write	$Vector \times size\_t \times Elem \rightarrow Vector$	$Desc_t \rightarrow Desc_t$	$O(1)$
size	$Vector \rightarrow size\_t$	$Desc_t \rightarrow Desc_t$	$O(1)$

The remaining part of this section presents the generalized pseudo-code of the implementation and omits code necessary for a particular memory management scheme. We use the symbols `^`, `&`, and `.` to indicate pointer dereferencing, obtaining an object's address, and integrated pointer dereferencing and field access respectively. The function `HighestBit` returns the bit-number of the highest bit that is set in an integer value. On modern x86 architectures `HighestBit` corresponds to the `BSR` assembly instruction. `FBS` is a constant representing the size of the first bucket and equals eight in our implementation.

add one element to end: The first step is to complete a pending operation that the current descriptor might hold. In case that the storage capacity has reached its limit, new memory is allocated for the next memory bucket. Then, `push_back` defines a new "Descriptor" object and announces the current write operation. Finally, `push_back` uses CAS to swap the previous "Descriptor" object with the new one. Should CAS fail, the routine is re-executed. After succeeding, `push_back` finishes by writing the element. Algorithm 4 shows `push_back`'s pseudocode.

remove one element from end `pop_back` does not utilize a "Write Descriptor". It completes any pending operation of the current descriptor, reads the last element, defines a new descriptor, and attempts a CAS on the descriptor object. Pseudocode for `pop_back` is shown in Algorithm 9.

Non-bound checking Read and Write at position i: The random access `read` (Algorithm 7) and `write` (Algorithm 8) do not utilize the descriptor and their success is independent of the descriptor's value.

`reserve` (increase allocated space): In the case of concurrently executing `reserve` operations (Algorithm 10), only one succeeds per bucket, while the others deallocate the acquired memory.

`size` (read number of elements): The `size` operation (Algorithm 6) returns the size stored in the "Descriptor" minus a potential pending write operation at the end

of the vector.

Algorithm 4 lf::vector — PushBack $vector, elem$

```
1: repeat
2:     $desc_{current} \leftarrow vector.desc$
3:     $CompleteWrite(vector, desc_{current}.writeop)$
4:     $bucket \leftarrow HighestBit(desc_{current}.size + \texttt{FBS}) - HighestBit(\texttt{FBS})$
5:     if vector.memory[bucket] = NULL then
6:         $AllocBucket(vector, bucket)$
7:     end if
8:     $writeop \leftarrow WriteDesc(At(desc_{current}.size)\hat{}, elem, desc_{current}.size)$
9:     $desc_{next} \leftarrow new\ Descriptor(desc_{current}, +1, writeop)$
10: until $CAS(\&vector.desc, desc_{current}, desc_{next})$
11: $CompleteWrite(vector, desc_{next}.writeop)$
```

Algorithm 5 lf::vector — AllocBucket $vector, bucket$

```
1: $bucketsize \leftarrow \texttt{FBS}^{bucket+1}$
2: $mem \leftarrow new\ T[bucketsize]$
3: if $not\ CAS(\&vector.memory[bucket], NULL, mem)$ then
4:     $Free(mem)$
5: end if
```

VII.B.1.a. The ABA Problem

The ABA problem is fundamental to all CAS-based systems [125]. The semantics of the lock-free vector's operations can be corrupted by the occurrence of the ABA problem. Consider the following execution: assume a thread T_0 attempts to perform a `push_back`; in the vector's "Descriptor", `push_back` stores a write-descriptor an-

Algorithm 6 lf::vector — Size *vector*

```
1: $desc \leftarrow vector.desc$
2: $size \leftarrow desc.size$
3: if $desc.writeop.pending$ then
4:     $size \leftarrow size - 1$
5: end if
6: return $size$
```

Algorithm 7 lf::vector — Read *vector*, *i*

```
1: return $At(vector, i)\hat{}$
```

Algorithm 8 lf::vector — Write *vector*, *i*, *elem*

```
1: $At(vector, i)\hat{} \leftarrow elem$
```

Algorithm 9 lf::vector — PopBack *vector*

```
1: repeat
2:     $desc_{current} \leftarrow vector.desc$
3:     $CompleteWrite(vector, desc_{current}.writeop)$
4:     $elem \leftarrow At(vector, desc_{current}.size - 1)\hat{}$
5:     $desc_{next} \leftarrow new\ Descriptor(desc_{current}, -1)$
6: until $CAS(\&vector.desc, desc_{current}, desc_{next})$
7: return $elem$
```

nouncing that the value of the object at position i should be changed from A to B. Then a thread T_1 interrupts and reads the write-descriptor. Later, after T_0 resumes and successfully completes the operation, a third thread T_2 can modify the value at position i from B back to A. When T_1 resumes its CAS is going to succeed and erroneously execute the update from A to B. There are two particular instances when the ABA problem can affect the correctness of the vector's operations:

Algorithm 10 lf::vector — Reserve $vector, size$

```
1: $i \leftarrow HighestBit(vector.desc.size + \texttt{FBS} - 1) - HighestBit(\texttt{FBS})$
2: **if** $i < 0$ **then**
3:     $i \leftarrow 0$
4: **end if**
5: **while** $i < HighestBit(size + \texttt{FBS} - 1) - HighestBit(\texttt{FBS})$ **do**
6:     $i \leftarrow i + 1$
7:     $AllocBucket(vector, i)$
8: **end while**
```

Algorithm 11 lf::vector — At $vector, i$

```
1: $pos \leftarrow i + \texttt{FBS}$
2: $hibit \leftarrow HighestBit(pos)$
3: $idx \leftarrow pos\ xor\ 2^{hibit}$
4: **return** $\&vector.memory[hibit - HighestBit(\texttt{FBS})][idx]$
```

Algorithm 12 lf::vector — CompleteWrite $vector, writeop$

```
1: **if** $writeop.pending$ **then**
2:     $CAS(At(vector, writeop.pos), writeop.value_{old}, writeop.value_{new})$
3:     $writeop.pending \leftarrow false$
4: **end if**
```

1. the user intends to store a memory address value A multiple times.

2. the memory allocator reuses the address of an already freed object.

A universal solution to the ABA problem is to associate a version counter to each element on platforms supporting CAS2. However, because of hardware requirements of our primary application domain, we cannot currently assume availability of CAS2.

To eliminate the ABA problem of (2) (in the absence of CAS2), we have incorporated a variation of Herlihy et al.'s pass the buck algorithm [91] utilizing a separate thread to periodically reclaim unguarded objects.

The vector's vulnerability to (1) (in the absence of CAS2), can be eliminated by requiring the data structure to copy all elements and store pointers to them. Such behavior complies with the STL value-semantics [171], however it can incur significant overhead in some cases due to the additional heap allocation and object construction. In a lock-free system, both the object construction and heap allocation can execute concurrently with other operations. However, for significant applications, our vector can be used because the application programmer can avoid ABA problem (1). For example, a vector of unique elements (e.g. a vector recording live or active objects) does not suffer from this problem. Similarly, a vector that has a "growth phase" (using `push_back`) that is separate from a "write phase" (using assignment to elements) (e.g., an append-only vector) is safe. The testbed for goal driven autonomous software exhibits such behavior. In order to prevent ABA from occurring, we can statically detect source code that violates safe coding guidelines [58].

VII.C. Performance Evaluation

We ran performance tests on an Intel IA-32 SMP machine with two 1.83GHz processor cores with 512 MB shared memory and 2 MB L2 shared cache running the MAC OS 10.4.6 operating system. In our performance analysis, we compare the lock-free approach (with its integrated lock-free memory management and memory allocation) with the most recent concurrent vector provided by Intel [95] as well as an STL vector protected by a lock. For the latter scenario we applied different types of locking synchronizations - an operating system dependent mutex, a reader/writer lock, a spin lock, as well as a queuing lock. We used this variety of lock-based techniques to contrast our non-blocking implementation to the best available locking synchronization technique for a given distribution of operations. We utilize the locking synchronization provided by Intel [95].

Similarly to the evaluation of other lock-free concurrent containers [74, 123], we have designed our experiments by generating a workload of various operations (`push_back`, `pop_back`, random access `write`, and `read`). In the experiments, we varied the number of threads, starting from 1 and exponentially increased their number to 32. Every active thread executed 500,000 operations on the shared vector. We measured the CPU time (in seconds) that all threads needed in order to complete. Each iteration of every thread executed an operation with a certain probability; `push_back` (`+`), `pop_back` (`-`), random access `write` (`w`), random access `read` (`r`). We use per-thread linear congruential random number generators where the seeds preserve the exact sequence of operations within a thread across all containers. We executed a number of tests with a variety of distributions and found that the differences in the containers' performances are generally preserved. As discussed by Fraser [74], it has been observed that in real-world concurrent applications, the read operations dominate and account to about 70% to 75% of all operations. For this reason we illustrate the performance of the concurrent vectors with a distribution of `+`:15%, `-`:5%, `w`:10%, `r`:70% on Fig. 16A. Similarly, Fig. 16C demonstrates the performance results with a distribution containing predominantly writes, `+`:30%, `-`:20%, `w`:20%, `r`:30%. In these diagrams, the number of threads is plotted along the x-axis, while the time needed to complete all operations is shown along the y-axis. Both axes use logarithmic scale.

The current release of Intel's concurrent vector does not offer `pop_back` or any alternative to it. To include its performance results in our analysis, we excluded the `pop_back` operation from a number of distributions. Fig. 16B and Fig. 16D present two

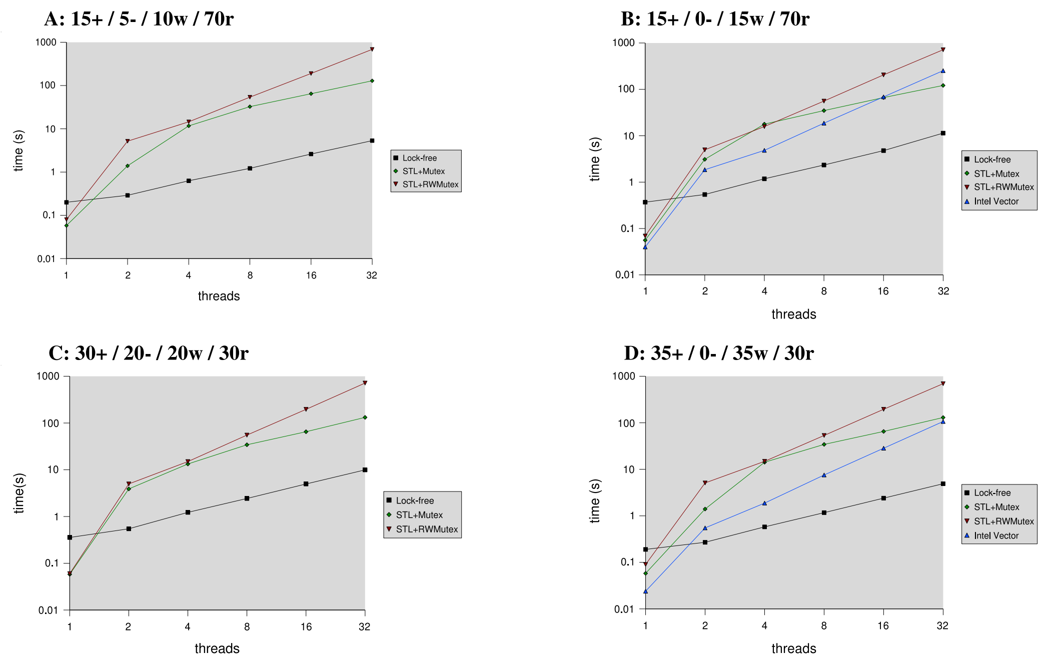

Fig. 16. Performance results - Intel Core Duo

of these distributions. For clarity we do not depict the results from the QueuingLock and SpinLock implementations. According to our observations, the QueuingLock performance is consistently slower than the other lock-based approaches. As indicated in [95], SpinLocks are volatile, unfair, and not scalable. They showed fast execution for the experiments with 8 threads or lower, however their performance significantly deteriorated with the experiments conducted with 16 or more active threads. To find a lower bound for our experiments we timed the tests with a non-thread safe STL-vector with pre-allocated memory for all operations. For example, in the scenario described in Fig. 16D, the lower bound is about a $\frac{1}{10}$ of the lock-free vector.

Under contention our non-blocking implementation consistently outperforms the alternative lock-based approaches in all possible operation mixes by a significantly large factor. It has also proved to be scalable as demonstrated by the performance analysis. Lock-free algorithms are particularly beneficial to shared data under high contention. It is expected that in a scenario with low contention, the performance gains will not be as considerable.

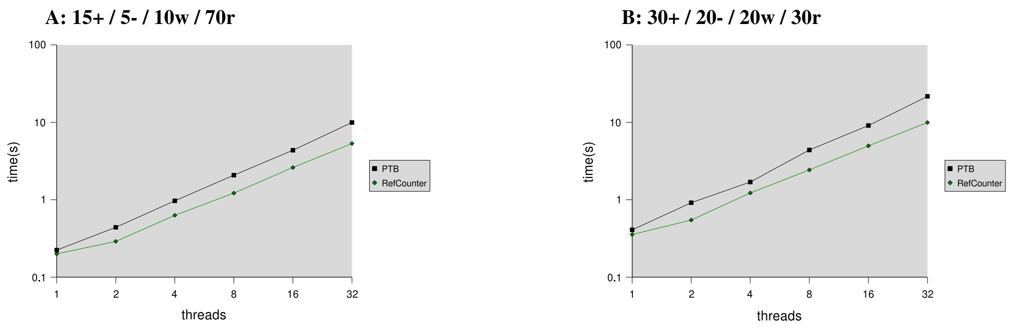

Fig. 17. Performance results - alternative memory management

We have incorporated two different memory management approaches with our lock-free implementation, namely Michael and Scott's reference counting scheme (RefCount) [126] and Herlihy et al.'s pass the buck technique (PTB) [91]. We have evaluated the vector's performance with these two different memory management schemes (Fig. 17).

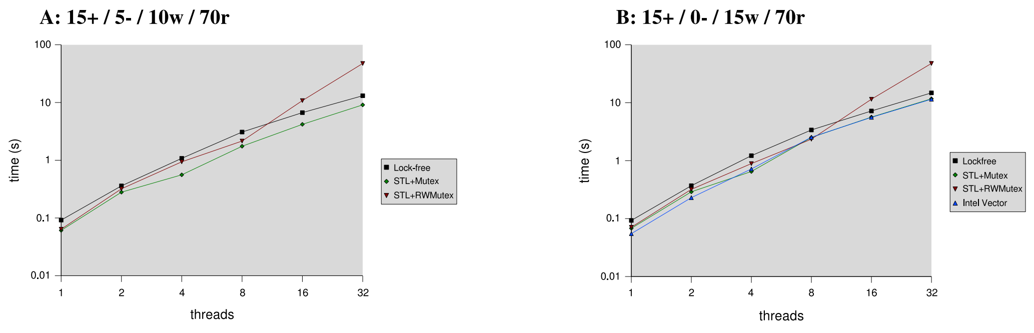

Fig. 18. Performance results - AMD 8-way Opteron

On systems without shared L2 cache, shared data structures suffer from performance degradation due to cache coherency problems. To test the applicability of our approach on such architecture we have performed the same experiments on an AMD 2.2GHz quad dual core Opteron architecture with 1 MB L2 cache and 4GB shared RAM running the MS Windows 2003 operating system. (Fig. 18). The applied lock-free memory allocation scheme is not available for MS Windows. For the sake of our performance evaluation we applied a regular lock-based memory allocator. The experimental results on this architecture lack the impressive performance gains we have observed on the dual-core L2 shared-cache system. However, the graph (Fig. 18) demonstrates that the performance of our lock-free approach on such architectures is comparable to the performance of the best lock-based alternatives.

VII.D. Future Work

In concurrent programs, the access of shared data structures is often guarded by fine-grained mutual exclusion locks. As discussed, the application of such locks can lead to problems not present in non-blocking implementations. To address such problems, we will apply our rejuvenation toolchain to the detection and replacement of lock based container implementations with one lock-free implementations.

CHAPTER VIII

TEMPLATE ANALYSIS AND CONCEPT EXTRACTION

The choice of requirements for an argument of a generic type or algorithm is a central design issue in generic programming. In the context of C++, a specification of requirements for a template argument or a set of template arguments is called a *concept*.

In this dissertation, we present a novel tool, TACE (template analysis and concept extraction), designed to help programmers understand the requirements that their code de facto imposes on arguments and help simplify and generalize those through comparisons with libraries of well-defined and precisely-specified concepts. TACE automatically extracts requirements from the body of template functions. These requirements are expressed using the notation and semantics developed by the ISO C++ standards committee. TACE converts implied requirements into concept definitions and compares them against concepts from a repository. Components of a well-defined library exhibit commonalities that allow us to detect problems by comparing requirements from many components: Design and implementation problems manifest themselves as minor variations in requirements. TACE points to source code that cannot be constrained by concepts and to code where small modifications would allow the use of less constraining concepts. For people who use a version of C++ with concept support, TACE can serve as a core engine for automated source code rejuvenation.

VIII.A. Introduction

A fundamental idea of generic programming is the application of mathematical principles to the specification of software abstractions [169]. ISO C++ [97, 171] supports generic programming through the use of templates. Unfortunately, it does not directly support the specification of requirements for arguments to generic types and functions [170]. However, research into language-level support for specifying such requirements, known as *concepts*, for C++ has progressed to the point where their impact on software can be examined [83, 27, 84]. Our work is aimed at helping programmers cope with the current lack of direct support for concepts and ease the future transition to language-supported concepts.

Templates are a compile-time mechanism to parameterize functions and classes over types and values. When the concrete template argument type becomes known to the compiler, it replaces the corresponding type parameter (template instantiation), and type checks the instantiated template body. This compilation model is flexible, type safe, and can lead to high performance code [63]. For over a decade, C++ templates have helped deliver programs that are expressive, maintainable, efficient, and organized into highly reusable components [172]. Many libraries, such as the C++ Standard Template Library (STL) [15], the BOOST graph library [161], and the parallel computation system, STAPL [13], for which adaptability and performance are paramount, rest on the template mechanism.

C++ currently does not allow the requirements for the successful instantiation of a template to be explicitly stated. Instead, such requirements must be found in documentation or inferred from the template body. For attempts to instantiate a template with types that do not meet its requirements, current compilers often fail with error messages that are hard to understand [83]. Also, C++ provides only weak support for overloaded templates. A number of programming techniques [1][9][102][160] offer partial solutions to these problems, but they tend to raise the level of complexity of template implementations and can make programs harder to understand.

Concepts [83][27][84] were developed to provide systematic remedies and deliver better support for the design and development of generic programs. As defined for C++0x, concepts improve expressiveness, make error messages more precise, and provide better control of the compile-time resolution of templates. Importantly, the use of concepts does not incur runtime overhead when compared to templates not using concepts. However, despite many years design efforts, implementation work, and experimental use, concerns about usability, scalability, and the time needed to stabilize a design prevented concepts from being included as a language mechanism in the next revision of C++ [176][175]. However, we are left with a notation and a set of concepts developed for the STL and other libraries that can be used to describe and benchmark our use of design-level concepts.

In this paper, we present a novel tool for template analysis and concept extraction, TACE, that addresses some of these concerns. TACE extracts concept requirements from industrial strength C++ code and helps apply concepts to unconstrained templated code. This dissertation offers the following contributions:

- a strategy for evolving generic code towards greater generality, greater uniformity, and more precise specification.
- type level evaluation of uninstantiated template functions and automatic extraction of sets of requirements on template arguments.
- concept analysis that takes called functions into account.

Experience with large amounts of generic C++ code and the development of C++ generic libraries, such as the generic components of the C++0x standard library [25] shows that the source code of a class template or a function template is not an adequate specification of its requirements. Such a definition is sufficient for type safe code generation, but even expert programmers find it hard to provide implementations that do not accidentally limit the applicability of a template (compared to its informal documentation). It is also hard to precisely specify template argument requirements and to reason about those.

Consequently, there is wide agreement in the C++ community that a formal statement of template argument requirements in addition to the template body is required. Using traditional type deduction techniques [6] modified to cope with C++, TACE generates such requirements directly from the code to enable the programmer to see the implications of implementation decisions. Furthermore, the set of concepts generated from an implementation is rarely the most reusable or the simplest. To help validate a library implementation TACE compares the generated (implied) concepts to pre-determined set of library concepts.

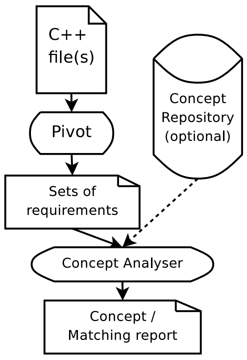

Fig. 19. The TACE tool chain

Fig. 19 shows TACE's tool chain. TACE utilizes the Pivot source-to-source transformation infrastructure [64] to collect and analyze information about C++ template functions. The Pivot's internal program representation preserves high-level information present in the source code - it represents uninstantiated templates and is ready for concepts. TACE analyzes expressions, statements, and declarations in the body of template functions. Since function instantiations do not have to be present, TACE can operate on modularily defined template libraries. It evaluates the template body on the type level and extracts the requirements on template arguments. TACE alerts programmers about code where a requirement cannot be modeled with concepts. It merges the requirements with requirements extracted from functions that the template body potentially invokes. The resulting sets of requirements can be written out as concept definitions. TACE does not try to discover semantic properties of a concept ("axioms" [65]. In general, doing so is beyond the scope of static analysis.

Our experiments demonstrate that TACE can extract requirements from individual functions. However, our goal is to find higher-level concepts that prove useful at the level of the design of software libraries. In particular, we do not just want to find the requirements of a particular implementation of an algorithm or the absolute minimal set of requirements. We want to discover candidates for concepts that are widely usable in interface specifications for algorithms. To recognize such concepts we need the "advice" of an experienced human. TACE achieves this by matching the extracted sets of requirements against concepts stored in a concept repository (e.g., containing standard concepts). In addition to reporting matches, the tool also reports close misses. In some cases, this allows programmers to reformulate their code to facilitate types that model a weaker concept. Our test results for STL indicate that our tool is effective when used in conjunction with a concept repository that contains predefined concepts.

VIII.B. Concepts for C++

Concepts as designed for C++0x [27, 83, 84] provide a mechanism to express constraints on template arguments as sets of syntactic and semantic requirements.

Syntactic requirements describe requirements such as associated functions, types, and templates that are necessary for the template instantiation to succeed. Consider the following template, which determines the distance between two iterators:

```
template<typename Iterator>
size_t distance(Iterator first, Iterator last) {
  size_t n = 0;
  while (first != last) { ++first; ++n; }
  return n;
}
```

The function `distance` requires types that substitute for the type parameter `Iterator` have a copy constructor (to copy the arguments), an inequality (`!=`) operator, and an increment (`++`) operator. A requirement's argument type can be derived from the source code. Requirements can be stated using a C++ function signature like notation.

```
concept DistanceRequirements<typename T> {
  T::T(const T&); // copy constructor
  bool operator!=(T, T);
  void operator++(T);
}
```

In order not to over-constrain templates, function signatures of types that model the concept need not match the concept signature exactly. C++'s pseudo signatures allow automatic conversions of argument types. This means that an implementation of `operator!=` can accept types that are constructable from `T`.

The return type of a functional requirement has to be named but can remain unbound. The following example shows a function with two parameters of type `T`, where `T` is a template argument constrained by the concept `TrivialIterator`. The function tests whether the return values of the `operator*` are equal. The type of the return values is irrelevant as long as there exists an `operator==` that can compare the two. The result type of the equality comparison must be convertible to `bool`.

```
template <TrivialIterator T>
bool same_elements(T lhs, T rhs) {
  return (*lhs == *rhs);
}
```

Concepts introduce *associated types* to model such types. The following concept definition of `TrvialIterator` introduces such an associated type `ValueType` to specify the return type of `operator*`. Associated types can be constrained by nested requirements (e.g., the `requires` clause).

```
concept TrivialIterator<typename T> {
  typename ValueType;
  // nested requirements
  requires EqualityComparable<ValueType>; // operator== of ValueType
  ValueType operator*(T); // deref operator*
  . . .
}
```

The compiler will use the concept definitions to type check expressions, declarations, and statements of the template body without instantiating it. Any type (or combination of types) that defines the required operations and types is a model of the concept. Those types can be used to instantiate the template bodies.

Semantic requirements describe behavioral properties, such as the equivalence of operations or runtime complexity. Types that satisfy the semantic requirements are guaranteed to work properly with a generic algorithm. Axioms model some behavioral properties. Axioms can specify the equivalence of operations. The two operations are separated by an operator `<=>`. In the following example, the axiom `indirect_deref` specifies that the operations of the left and right side of `<=>` produce the same result. This is the case for pointers or random access iterators. Compilers are free to use axioms for code optimizations.

```
concept Pointer<typename T> {
  typename data;
  data operator*(T);
  T operator+(T, size_t);
  data operator[](T, size_t);
  axiom indirect_deref(T t, size_t n) {
    t[n] <=> *(t+n);
  }
}
```

Concepts can extend one or more existing concepts and add new requirements. Any requirement of the "base" concept remains valid for its *concept refinements*. For example, consider a trivial iterator abstraction, which essentially defines an operation to access its element (`operator*`). The concept `ForwardIterator` adds operations to traverse a sequential data structure in one way (from the beginning to an end).

```
concept ForwardIterator<typename T> {
  requires TrivialIterator<T>;
  ...
}
```

Concept refinements are useful for the implementation of a family of generic functions. A base implementation constrains its template arguments with a general concept, while specialized versions exploit the stronger requirements of concept refinements to provide more powerful or more efficient implementations. Consider, the STL algorithm `advance(Iter, Size)` for which three different implementations exist. Its basic implementation is defined for input-iterators and has runtime complexity $O(Size)$. The version for bidirectional-iterators can handle negative distances, and the implementation for random access improves the runtime complexity to $O(1)$. The compiler selects the implementation according to the concept a specific type models [100].

In current C++, programmers resort to a combination of trait classes [132], tag dispatching [1], or substitution failure is not an error (SFINAE) based techniques [102] to exert influence on the overload resolution.

Concepts can be used to constrain template arguments of stand-alone functions. In such a scenario, the extracted concept requirements reflect the function implementation directly. In the context of template libraries, clustering similar sets of requirements yields reusable concepts, where each concept constrains a family of types that posses similar qualities. Clustering requirements results in fewer and easier to comprehend concepts and makes concepts more reusable. An example of concepts, refinements, and their application is STL's iterator hierarchy, which groups iterators by their access capabilities.

Concept requirements are bound to concrete operations and types by the means of *concept maps*. Concept maps can be automatically generated. Should a type's operations not exactly match the requirement definition (e.g., when a function is named differently), concept maps allow for an easy adaptation [101].

VIII.C. Requirements Extraction

TACE extracts individual concept requirements from the body of template functions by infering properties of types from declarations, statements, and expressions. Similar to the usage pattern style of concept specification [63], we derive the requirements by reading C++'s evaluation rules [62] backwards. We say backwards, because regular type checking tests whether expressions, statements, and declarations together with type (concept) constraints result in a well-typed program. In this work, we start with an empty set of constraints and derive the type (concept) constraints that make type-checking of expressions, statements, and declarations succeed. The derived constraints (ζ) reflect functional requirements and associated typenames.

VIII.C.1. Evaluation of Expressions

A functional requirement $op(arg_1,\ \ldots,\ arg_n) \rightarrow res$ is similar to a C++ signature. It consists of a list of argument types (arg) and has a result type (res). Since the concrete type of template dependent expressions is not known, the evaluator classifies the type of expressions into three groups:

Concrete types: this group comprises all types that are legal in non template context. It includes built-in types, user defined types, and templates that have been instantiated with concrete types. We denote types of this class with C.

Named template dependent types: this group comprises named but not yet known types (i.e., class type template parameters, dependent types, associated types, and instantiations that are parametrized on unknown types), and their derivatives. Derivatives are constructed by applying pointers, references, `const` and `volatile` qualifiers on a type. Thus, a template argument `T`, `T*`, `T**`, `const T`, `T&`, `typename traits<T>::value_type` are examples for types grouped into this category. We denote types of this class with T.

Requirement results: This group comprises fresh type variables. They occur in the context of evaluating expressions where one or more subexpressions have a non concrete type. The symbol R denotes types of this class. The types R are unique for each operation, identified by name and argument types. Only the fact that multiple occurrences of the same function must have the same return type, enables the accumulation of constraints on a requirement result. (e.g., the STL algorithm `search` contains two calls to `find`).

In the ensuing description, we use N for non concrete types ($T \cup R$) and A for any type ($N \cup C$). For each expression, the evaluator yields a tuple consisting of the return type and the extracted requirements. For example, $\mathit{expr} : C, \zeta$ denotes an expression that has a concrete type and where ζ denotes the requirements extracted for *expr* and its subexpressions. We use $X \rightsquigarrow Y$ to denote type X is convertible to type Y. Table XVII shows TACE's evaluation rules of expressions in a template body.

A *concrete expression* (*expr*) is an expression that does not depend on any template argument (e.g., literals, or expressions where all subexpressions (s) have concrete type). The result has a concrete type. The direct subexpressions have concrete types,

Table XVII. Evaluation rules for expressions

concrete expression	$\dfrac{\Gamma \overset{exp}{\vdash} s_1{:}C_1,\zeta_1 \;...\; s_n{:}C_n,\zeta_n}{\Gamma \overset{exp}{\vdash} expr(s_1,\ ...,\ s_n){:}C_{expr}, \bigcup\limits_{1\le i\le n} \zeta_i}$
unbound function	$\dfrac{\Gamma \overset{exp}{\vdash} s_1{:}A_1,\zeta_1 \;...\; s_n{:}A_n,\zeta_n}{\Gamma \overset{exp}{\vdash} uf(s_1,...,s_n){:}R_{uf(s1,\ ...\ sn)}, \bigcup\limits_{1\le i\le n} \zeta_i \cup \{uf(A_1,\ ...\ ,A_n) \rightarrow R_{uf(s_1,\ ...\ s_n)}\}}$
bound function	$\dfrac{\Gamma \overset{exp}{\vdash} (bf{:}(A_1^{bf},...,A_n^{bf}) \rightarrow A_r^{bf})\; s_1{:}A_1,\zeta_1 \;...\; s_n{:}A_n,\zeta_n}{\Gamma \overset{exp}{\vdash} fn(s_1,\ ...,\ s_n){:}A_r^{bf}, \bigcup\limits_{1\le i\le n} \zeta_i \cup \{A_i \rightsquigarrow A_i^{bf}\}}$
conditional operator	$\dfrac{\Gamma \overset{exp}{\vdash} s_1{:}A_1,\zeta_1\;\; s_2{:}A_2,\zeta_2\;\; s_3{:}A_2,\zeta_3}{\Gamma \overset{exp}{\vdash} (s_1?s_2{:}s_3){:}\ A_2, \bigcup\limits_{1\le i\le 3} \zeta_i \cup \{A_1 \rightsquigarrow bool\}}$
member functions	$\dfrac{\Gamma \overset{exp}{\vdash} o{:}A_o,\zeta_o\;\; s_1{:}A_1,\zeta_1\;\; s_n{:}A_n,\zeta_n}{\Gamma \overset{exp}{\vdash} o.uf(s_1,...,s_n){:}R_{uf\ (o,s_1,\ ...\ s_n)},\zeta_o \cup \bigcup\limits_{1\le i\le n} \zeta_i \cup \{uf(A_o,A_1,\ ...\ ,A_n) \rightarrow R_{uf(o,s_1,\ ...\ s_n)}\}}$
non concrete arrow	$\dfrac{\Gamma \overset{exp}{\vdash} o{:}A_o,\zeta_o}{\Gamma \overset{exp}{\vdash} o\texttt{->}{:}R_{\texttt{->}o},\{operator\texttt{->}(A_o) \rightarrow R_{\texttt{->}o}\}}$
static cast	$\dfrac{\Gamma \overset{exp}{\vdash} A_{tgt},\zeta_{tgt}\;\; o{:}A_o,\zeta_o}{\Gamma \overset{exp}{\vdash} A_{tgt},\{A_o \rightsquigarrow A_{tgt}\}}$
dynamic cast	$\dfrac{\Gamma \overset{exp}{\vdash} A_{tgt},\zeta_{tgt}\;\; o{:}A_o,\zeta_o}{\Gamma \overset{exp}{\vdash} A_{tgt},\{PolymorphicClass{<}A_o{>}\}}$
other casts	$\dfrac{\Gamma \overset{exp}{\vdash} A_{tgt},\zeta_{tgt}\;\; o{:}A_o,\zeta_o}{\Gamma \overset{exp}{\vdash} A_{tgt},\{\}}$

but indirect subexpressions can be template dependent (e.g., `sizeof(T)`). Thus, ζ is the union of subexpression requirements.

Calls to *unbound functions* (*uf*) (and unbound overloadable operators, constructors, and destructor) have at least one argument that depends on an unknown type N. Since *uf* is unknown, its result type is denoted with a fresh type variable $R_{uf\ (s_1,\ \ldots\ s_n)}$.

Bound functions are functions, where the type of the function can be resolved at the compile time of the template body. Examples of bound functions include calls to member functions, where the type of the receiver object is known, and calls, where argument dependent lookup [97] is suppressed. Calls to bound functions (*bf*) have the result type of the callee. *bf*'s specification of parameter and return types can add conversion requirements to ζ (i.e., when the type of a subexpression differs from the specified parameter type and when at least one of these types is not concrete.)

The *conditional operator* (`?:`) cannot be overloaded. The ISO standard definition requires the first subexpression be convertible to `bool`. TACE's evaluation rules require the second and the third subexpression to be the same type. Here TACE is currently stricter than the ISO C++ evaluation which allows for a conversion of one of the result types.

The other not overloadable operators (i.e., `typeid` and `sizeof`) have a concrete result type. The set of extracted requirements is the same as for their subexpressions and type names.

C++ concepts do not support modeling of member variables. Thus the application of a member selection (i.e, the `dot` or `arrow operator` can only refer to a member function name. The evaluator rejects any dot expression that occurs not in the context of evaluating the receiver of a call expression.

For *non concrete objects*, the evaluator treats the `arrow` as a unary operator that yields an object of unknown result type. The object becomes the receiver of a

Table XVIII. Evaluation rules for declarations and statements

statement context	$\dfrac{\Gamma \overset{stmt}{\vdash} \tau{\in}N,s{:}A,\zeta}{\epsilon,\zeta{+}A{\rightsquigarrow}\tau}$
default ctor	$\dfrac{\Gamma \overset{decl}{\vdash} o{:}(\Gamma,\tau{\in}N_o)}{\Gamma \overset{decl}{\vdash} o{:}\tau,\{\tau{::}ctor()\}}$
single argument ctor	$\dfrac{\Gamma \overset{decl}{\vdash} o{:}(\Gamma,\tau{\in}N_o),s_1{:}A_1,\zeta_1}{\Gamma \overset{decl}{\vdash} o{:}\tau,\zeta_1{+}\tau{::}ctor(const\tau\&){+}A_0{\rightsquigarrow}\tau}$
constructor	$\dfrac{\Gamma \overset{decl}{\vdash} o{:}(\Gamma,\tau{\in}N_o),s_1{:}A_1,\zeta_1,\ ...,\ s_n{:}A_n,\zeta_n}{\Gamma \overset{decl}{\vdash} o{:}\tau,\ \bigcup_{1\leq i\leq n} \zeta_i{+}\tau{::}ctor(A_1,\ ...,\ A_n)}$
parameter	$\dfrac{\Gamma \overset{decl}{\vdash} p{:}(\Gamma,\tau{\in}N_o)}{\Gamma \overset{decl}{\vdash} p{:}\tau,\{\tau{::}ctor(A_1,\ ...,\ A_n)\}}$

subsequent call to an unbound member function.

`Cast` expressions are evaluated according to the rules specified in Table XVII. The target type of a cast expression is also evaluated and can add dependent name requirements to ζ. A `static_cast` requires the source type be convertible to the target type. A `dynamic_cast` requires the source type to be a polymorphic class (`PolymorphicClass` is part of C++ with concepts).

The evaluation of operations on pointers follows the regular C++ rules, thus the result of dereferencing $T*$ yields $T\&$, the arrow operator yields a member function selection of T, taking the address of $T*$ yields T`**`, and any arithmetic expression on $T*$ has type $T*$. *Variables* in expressions are typed as lvalues of their declared type.

VIII.C.2. Evaluation of Declarations and Statements

Statements and declarations are evaluated according to the rules shown in Table XVIII.

Statements: The condition expressions of `if`, `while`, `for` require the expression to be convertible to `bool`. The `return` statement requires convertibility of the expression to the function return type. The expression of the `switch` statement is either convertible to `signed` or `unsigned` integral types. We introduce an artificial type `Integer` that subsumes both types. The type will be resolved later, if more information becomes available.

Object declarations: Variable declarations of object type require the presence of a constructor. Constructions with a single argument (i.e., $T\ t\ =\ arg$) are modeled to require $A_{arg} \rightsquigarrow T$ and a copy constructor on T.

References: Bindings to lvalue (mutable) references (i.e., declarations, function calls, and `return` statements) imposer stricter requirements. Instead of convertibility, they require the result type of an expression be an exact type (instead of a convertible type).

VIII.C.3. Evaluation of Class Instantiations

The current implementation focuses on extracting requirements from functions, and thus treats any instantiation of classes that have data members and where the template parameter is used as template argument as concrete type (e.g. `pair`, `reverse_iterator`); ζ remains unchanged. To allow the analysis of real world C++ template functions, TACE analyzes classes that contain static members (types, functions, and data). Particularly, trait classes can add dependent type requirements to ζ. For example, the instantiation of `iterator_traits<T>::value_type` leads to the type constraint `T::value_type`.

Static (templated) member functions of templated classes (e.g.: the various variants of `sort`) are treated as if they were regular template functions. The template parameters of the surrounding class extend the template parameters of the member function. For example:

```
template <class T>
struct S { template <class U> static T bar(U u); };
```

```
template<typename FwdIter1, typename FwdIter2>   | concept Search <typename FwdIter1,
FwdIter1                                         |                 typename FwdIter2> {
search(FwdIter1 first1, FwdIter1 last1,          |   // argument construction
       FwdIter2 first2, FwdIter2 last2) {        |   FwdIter1::FwdIter1(const FwdIter1&);
                                                 |   FwdIter2::FwdIter2(const FwdIter2&);
                                                 |   // if statement and return
  if (first1 == last1 || first2 == last2)        |   typename r1;
    return first1;                               |   r1 operator==(FwdIter1&, FwdIter1&);
                                                 |   typename r2;
                                                 |   r2 operator==(FwdIter2&, FwdIter2&);
                                                 |   typename r3;
                                                 |   r3 operator||(r1, r2);
                                                 |   operator bool(r3);
                                                 |   // second range has length 1
  FwdIter2 tmp(first2);                          |   r5 operator++(FwdIter2&);
  ++tmp;                                         |   operator bool(r2);
  if (tmp == last2)                              |   typename r7;
    return find(first1, last1, *first2);         |   r7 operator*(FwdIter2&);
                                                 |   typename r8;
                                                 |   r8 find(FwdIter1&, FwdIter1&, r7);
                                                 |   operator FwdIter1(r8);
                                                 |   // while loop
  FwdIter2 p1, p;                                |   FwdIter2::FwdIter2(); // default constructor
  p1 = first2;                                   |   void operator=(FwdIter2&, FwdIter2&);
  ++p1;                                          |   typename r12;
  FwdIter1 current = first1;                     |   r12 operator!=(FwdIter1&, FwdIter1&);
  while (first1 != last1) {                      |   operator bool(r12);
    first1 = find(first1, last1, *first2);       |   void operator=(FwdIter1&, r8);
    // ...                                       |   // ...
```

Fig. 20. Requirement extraction

is treated as:

template <**class** T, **class** U> T bar(U u);

VIII.C.4. Examples

We use the beginning of GCC's implementation (4.1.3) of the STL function `search` to illustrate our approach. Fig. 20 shows a portion of the implementation and the requirements that get extracted from it.

We begin by extracting the requirements from the argument list. Their types are `FwdIter1` and `FwdIter2`. They are passed by value. According to the evaluation rule for parameters, their type has to support copy construction.

The condition of the `if` statement is evaluated bottum up. The right hand side of the `operator||` is an equality comparison (`operator==`) of two parameters of type `FwdIter1&`. Since `FwdIter1` is an unknown type, the operation is evaluated according to the unbound function rule. The result type of the operation is a fresh type variable ($r1$). The set of requirements consists of the equality comparison `operator==(FwdIter1&, FwdIter1&)` $\to$ $\mathtt{r}_1$. Similarily, the evaluation of the comparison of `first2` and `last2` yields $\mathtt{r}_2$ and the requirement `operator==(FwdIter2&, FwdIter2&)` $\to$ $\mathtt{r}_2$

The evaluation proceeds with the `operator||`. Both arguments have an undetermined type ($\mathtt{r}_1$ and $\mathtt{r}_2$). Thus, the operation is evaluated according to the unbound function rule. The result type is $\mathtt{r}_3$. `operator==(`$\mathtt{r}_1, \mathtt{r}_2$`)` $\to$ $\mathtt{r}_3$ and the requirements extracted from the subexpressions form the set of requirements. $\mathtt{r}_3$ is evaluated by an if statement. According to the rule statement context, $\mathtt{r}_3$ has to be convertible to `bool`.

The return statement does not produce any new requirement, because the copy constructor of `FwdIter1` is already part of $\zeta_{\mathtt{search}}$.

The next source line declares a local variable `tmp`. Its initial value is constructed from a single argument of the same type. Thus, this line requires a copy constructor on `FwdIter2` (evaluation rule for constructors).

The next line moves the iterator `tmp` forward by one. The source line is evaluated according to the unbound function rule. The return type is $\mathtt{r}_5$, the extracted requirement is `operator++(FwdIter2&)` $\to$ $\mathtt{r}_5$.

The expression of the following `if` statement compares two expressions of type `FwdIter2&`. Unification with the already extracted requirements in $\zeta_{\mathtt{search}}$ yields the result type $\mathtt{r}_2$. Evaluating $\mathtt{r}_2$ according to the statement context rule yields an additional conversion requirement $\mathtt{r}_2 \rightsquigarrow$ `bool`.

The next line of code returns the result of a function call. First, the argument list of the call is processed. The first two arguments are references to parameters of type `FwdIter1`; the third argument dereferences a parameter of type `FwdIter2`. According to the unbound function rule, this expression yields to a new result type $\mathtt{r}_7$ and the requirement `operator*(FwdIter2&)` $\rightarrow \mathtt{r}_7$. Then TACE applies the unbound function rule to the function call itself. This yields the result type $\mathtt{r}_8$ and the requirement `find(FwdIter1&, FwdIter1&,` $\mathtt{r}_7$`)` $\rightarrow \mathtt{r}_8$. From the statement context, we infer $\mathtt{r}_8 \rightsquigarrow$ `FwdIter1`.

From the declarations of `p1` and `p`, `FwdIter2` is required to support default construction (`FwdIter2::FwdIter2()`).

We skip the remaining code and requirements.

VIII.D. From Requirements to Concepts

This section discusses the analysis and manipulation of the extracted requirements with the goal to print out *function specific concepts*.

VIII.D.1. Function Calls

Consider the requirements that were extracted from `search` (§VIII.C.4). It contains two calls to a function `find`, an unbound non-member call that potentially (or likely) resolves to STL's templated function. We have a choice, how we can handle such functions.

• A simplistic approach (§VIII.C.4) could print the concept `Search` and represent the calls to find as functional requirement.

```
typename r8;
r8 find(FwdIter1&, FwdIter1&, r7);
operator FwdIter1(r8);
void operator=(FwdIter1&, r8);
```

• Another approach would replace the requirements related to `find` with a simple refinement clause, and eliminate the requirements on `r8` (the conversion requirement becomes obsolete, and `operator=(FwdIter1, FwdIter1)` already exists in `search`).

```
requires Find<FwdIter1, r7>;
```

Both approaches lead to (deeply) nested requirements. `Search` does not expose the requirements on `FwdIter1` (or on the combination of `FwdIter1` with `r7`) that stem from `Find`. Thus, we would expect requirement errors that stem from deeply nested templated function calls to remain hard to understand for programmers.

• TACE's approach is to merge the requirements of the callee into the caller, if a callee exists. A callee exists, when there is a template function with the same name defined in the same namespace, and when that function's parameter are at least as general as the arguments of the call expression. (e.g., `search` calls `find`).

VIII.D.2. Result Type Reduction

In the extracted set of requirements, any result of an operation is represented as an unnamed type requirement (i.e., associated types such as `r1` and `r2` in §VIII.C.4). However, the evaluation context contributed more information about these types in the form of conversion requirements. TACE invokes a function *reduce* that propagates the target types of conversions.

$$\mathit{reduce}(\zeta) \rightarrow \zeta'$$

Should a requirement result have more than one conversion targets (for example, an unbound function was evaluated in the context of `bool` and `int`), we apply the following subsumption rule: assuming n conversion requirements with the same input type (R) but distinct target types A_i.

$$R' = \begin{cases} A_j & \text{if } \exists_j \forall_i \text{ such that } A_j \rightsquigarrow A_i \\ R & \text{otherwise} \end{cases}$$

Note, that the $A_j \rightsquigarrow A_i$ must be part of ζ, or defined for C++ built in types. If such an A_j exists, all operations that depend on R are updated, and become dependent on A_j. Any conversion requirement on R is dropped from ζ. When R is not evaluated by another function it gets the result type `void`. If R is evaluated by another expression, but no conversion requirement exists, the result type R' remains unnamed (i.e., becomes an associated type).

After the return type has been determined, the new type R' is propagated to all operations that use it as argument type. By doing so, the set of requirements can be further reduced (e.g., if all argument types of an operation are in C, the requirement can be eliminated, or in case the operation does not exist, an error reported) and more requirement result types become named (if an argument type becomes T, another operation on T might already exist). Reduction is a repetitive process that stops when a fixed point is reached. The size of the reduction depends on how much context information is available in code.

For example, *reduce* reduces the set of requirements that we got from merging `search` and `find`:

```
concept search <typename FwdIter1, typename FwdIter2> {
  FwdIter1::FwdIter1(const FwdIter1&);
  FwdIter2::FwdIter2(const FwdIter2&);
  bool operator==(FwdIter1&, FwdIter1&);
  bool operator==(FwdIter2&, FwdIter2&);
  typename r4;
  r4 operator++(FwdIter2&);
  typename r5;
  r5 operator*(FwdIter2&);
  FwdIter2::FwdIter2();
  void operator=(FwdIter2&, FwdIter2&);
  bool operator!=(FwdIter1&, FwdIter1&);
  void operator=(FwdIter1&, FwdIter1&);
  typename r11;
  r11 operator++(FwdIter1&);
  bool operator==(r11, FwdIter1&);
  typename r13;
  r13 operator*(FwdIter1&);
  bool operator==(r13, r5);
  bool operator==(r4, FwdIter2&);
  typename r529;          // from find
  r529 operator==(r13, const r5&);
  typename r530;
  r530 operator!(r529);
  bool operator&&(bool, r530);
}
```

Due to the presence of the conversion `operator FwdIter1(r8)`, `FwdIter1&` substitutes for `r8` in `void operator=(FwdIter1&, r8)`. Similar `r1` and `r2` become convertible to `bool`, thus the requirement `r3 operator||(r1, r2);` is dropped.

The result of `reduce` may constrain the type system more than the original set of requirements. Thus, `reduce` has to occur after merging all requirements from potential callees, when all conversion requirements on types are available.

VIII.E. Concept Matching with Repository

Template libraries utilize concepts to constrain the template arguments of a group of functions that operate on types with similar capabilities. To make this manageable in real code, this requires clustering similar sets of requirements into relatively few concepts. These concepts then provide a design vocabulary for the application domain of the library and help provide a degree of pluggability among algorithms and types. We tackle this by using a concept repository, which contains a number of predefined concept definitions (e.g., core concepts or concepts that users define for specific libraries). The use of a concept repository offers users the following benefits:

- reduces the number of concepts

- improves the structure of concepts
- exposes the refinement relationships of concepts, which allows for more exact analysis
- replaces requirement results with named types (concrete, template dependent, or associated typenames)

The repository we use to drive the examples in this sections contains the following concepts: `IntegralType<T>`, `RegularType<T>`, `TrivialIterator<T>`, `ForwardIterator<T>`, `BidirectionalIterator<T>`, `RandomaccessIterator<T>`, and `EqualityComparable<T>`. `IntegralType` specifies operations that are defined on type `int`. `RegularType` specifies operations that are valid for all built-in types (i.e., default construction, copy construction, destruction, assignment, equality comparison, and address of). `TrivialIterator` specifies the dereference operation and associated iterator types. The other iterators have the operations defined in the STL.

VIII.E.1. Concept Kernel

In order to match the extracted requirements of each template argument against concepts in the repository that depend on fewer template arguments, we partition the unreduced set into smaller sets called *kernels*. We define a concept kernel over a set of template arguments $\widehat{T}$ to be a subset of the original set of requirements ζ.

$$kernel(\zeta_{function}, \widehat{T}) \rightarrow \zeta_{kernel}$$

ζ_{kernel} is a projection that captures all operations on types that directly or indirectly originate from the template arguments in $\widehat{T}$.

$$\zeta_{kernel} \Leftrightarrow \{op | op \in \zeta_{src}, \phi_{\widehat{T}}(op)\}$$

For the sake of brevity, we also say that a type is in ζ_{kernel}, if the type refers to a result R of an operation in ζ_{kernel}.

$$\phi_{\widehat{T}}(op) = \begin{cases} 1 & \text{true for a } op(arg_1, \ldots, arg_n) \rightarrow res \\ & \text{if } \forall_i \ arg_i \in \widehat{T} \cup \zeta_{kernel} \cup C \\ 0 & \text{otherwise} \end{cases}$$

$\phi_{\widehat{T}}(op)$ is true, if all arguments of *op* either are in $\widehat{T}$, are result types from operations in ζ_{kernel}, or are concrete.

As an example, we give the concept kernel for the first template argument of `search`.

```
FwdIter1::FwdIter1(const FwdIter1&);
typename r1652;
r1652 operator==(FwdIter1&, FwdIter1&);
typename r1654;
r1654 operator!=(FwdIter1&, FwdIter1&);
operator bool (r1654);
operator bool (r1652);
void operator=(FwdIter1&, FwdIter1&);
typename r1658;
r1658 operator++(FwdIter1&);
typename r1659;
r1659 operator==(r1658, FwdIter1&);
operator bool (r1659);
typename r1661;
r1661 operator*(FwdIter1&);
```

VIII.E.2. Concept Matching

For each function, TACE type checks the kernels against the concepts in the repository. The mappings from a kernel's template parameters to a concept's template parameters are generated from the arguments of the first operation in a concept kernel and the parameter declarations of operations with the same name in the concept.

Matching the requirements against the definitions in the concept repository is similar to the Hindley-Milner-Damas (HMD) type inference for functional languages [55]. The HMD algorithm and its variations derive a type scheme for untyped entities of a function from type annotations and the utilization of these entities in the context of defined functions. Wadler and Blott [191] discuss problems of type inference for ad-hoc polymorphic functions in languages that support overloaded operators. They discuss a number of solutions that have been utilized by various functional languages. Subsequently, Wadler and Blott develop the notion of a type class as a structured solution to the problem. A type class defines (overloaded) operations for a set of types. Any type that provides the specified operations can become an instance of the type class. Peterson and Jones [140] present an extension to the HMD algorithm, which utilizes type classes to derive an unambiguous type scheme for polymorphic functions. Jones [103] formalizes the definition of Haskell's type inference algorithm by providing an implementation in Haskell.

We describe the necessary adaptations for checking C++ template functions against concept definitions below:

- C++ template code follows regular C++ programming style, where variables have to be declared before they can be used. The type of a variable can be template argument dependent. The types of variable declarations is useful for the resolution of for overloaded function requirement (e.g., random access iterator's subtraction and difference requirement).

- C++'s type system allows type coercions. Based on C++ binding rules and conversion (and constructor) requirements that are defined in the concept, TACE generates all possible type combinations that a specific function requirement can handle. For example, if a signature is defined over `const T&` another signature for `T&` is added to the concept. Consequently, checking whether a requirement kernel is expressible by a concept in the repository relies on signature unification. The result type of a function is inferred from the requirement specification in the repository.

- in C++, type checking of expressions is context dependent. The disambiguation of overloaded functions relies on context information. For a call, the argument list provides this context. In an assignment to a function pointer, the type of the pointer determines the function type. When code suppresses argument dependent lookup, the rules become more subtle. In this circumstance, the overload set only contains functions that were available at template definition time.

- Haskell type checking utilizes context reduction. For example, an equality implementation on lists may require that the list elements be comparable. Extending TACE with context reduction would be useful for deriving requirements from templated data-structures (see §VIII.C.3).

For any requirement in the kernel, a concept has to contain a single best matching requirement (multiple best matching signatures indicate an ambiguity). TACE checks the consistency of a concept requirement's result type with all conversion requirements in the kernel.

For each kernel and concept pair, TACE partitions the requirements into satisfiable, unsatisfiable, associated, and available functions. An empty set of unsatisfiable requirements indicates a match. TACE can report small sets of unsatisfiable requirements (i.e., near misses), thereby allowing users to modify the function implementation (or the concept in the repository) to make a concept-function pair work together. The group of associated requirements contains unmatched requirements on associated types. For example, any iterator requires the value type to be regular. Besides regularity, some functions such as `lower_bound` require less than comparability of container elements. Associated requirements are subsequently matched. The group of available functions contains requirements, where generic implementations exist.

This produces a set of candidate concepts. For example, the three iterator categories match the template parameters of `search`. `find` has two template arguments. Any iterator concept matches the first argument. Every concept in the repository matches the second argument.

For functions with more than one template argument, TACE generates the concept requirements using a Cartesian join of the results of the individual kernels.

The final step in reducing the candidate concepts is the elimination of candidates by eliminating all refinements, for which the refinee is also a candidate. In our case, the concepts for bidirectional and random access iterators are eliminated from the candidate set.

The following code snippet shows the result for `search`:

```
concept Search<typename FwdIter1, typename FwdIter2> {
  requires ForwardIterator<FwdIter1>;
  requires ForwardIterator<FwdIter2>;
  bool operator==( iterator_traits<FwdIter1>::value_type&,
                   iterator_traits<FwdIter2>::value_type&);
}
```

Note, that matching currently does not generate extra conversion requirements. Thus, the `operator==` with two different argument types does not match the `operator==` defined in `EqualityComparable<T>`.

VIII.E.3. Families of Functions

A generic function can consist of a family of different implementations, where each implementation exploits concept refinements (e.g., `advance` in §VIII.B).

A template that calls a generic function needs to incorporate the minimal requirements of the generic function in its concept. To do so, it is necessary to determine the most general implementation. Finding the base implementation is non trivial with real code. Consider STL's `advance` family. TACE extracts the following requirements:

```
// for Input−Iterators
concept AdvInputIter <typename Iter, typename Dist> {
  Dist::Dist(const Dist&);
  void operator++(Iter&);
  bool operator−−(Dist&, int);
}
// for Bidirectional−Iterators
concept AdvBidirectIter <typename Iter, typename Dist> {
  Dist::Dist(const Dist&);
  void operator++(Iter&);
  void operator−−(Iter&);
  bool operator++(Dist&, int);
  bool operator−−(Dist&, int);
}
// for Randomaccess−Iterators
concept AdvRandomaccessIter <typename Iter, typename Dist> {
  Dist::Dist(const Dist&);
  void operator+=(Iter&, Dist&);
}
```

The sets of extracted requirements for the implementations based on input- and bidirectional-iterator are in a subset/superset relation, the set of requirements for the random access iterator based implementation is disjoint with the former sets.

If calls to generic functions where a refinement relationship cannot be inferred occur under scenario §VIII.D, TACE requires the user mark the least specific function implementation. A concept repository helps infer the correct refinement relationship.

However, detecting the least specific implementation can still be problematic for generic functions with a too general definition. Consider the implementations of `advance` for input- and random access iterators.

```
// template <class InputIterator, Class Distance>
// void advance(InputIterator& iter, Distance dist);
```

```
void operator++(InputIterator&); // kernel(InputIterator)
Distance operator−−(Distance&,int); // kernel(Distance)
operator bool(Distance&); // kernel(Distance)
// template <class RandomaccessIter, Class Distance>
// void advance(RandomaccessIter& iter, Distance dist);
void operator+=(InputIterator&, Distance&); // multi−parametric
```

The kernel for `RandomaccessIter` is empty, thus any concept matches. After eliminating false positive candidates, any iterator concept matches; the requirement `operator+=` becomes an operation that is defined over two independent template arguments. Recent concept based STL implementations make the type dependence between the two parameters explicit and TACE correctly identifies the random access iterator concept.

VIII.F. Results

We validated the approach by matching the functions defined in GCC's header file `algorithm`. The file contains more than 9000 non-comment (and non empty) lines of code and defines 115 algorithms plus about 100 helper functions. The `algorithm` header exercises some of the most advanced language features and design techniques used for generic programming in C++.

The success rate of the concept recovery depends on the concepts in the repository. A repository containing syntactically similar concepts will lead to ambiguous results. We ran the tests against the repository introduced in §VIII.E plus concepts that are defined over multiple template arguments (`UnaryFunction`, `UnaryPredicate`, `BinaryFunction` and `BinaryPredicate`). The predicates refine the functions and require the return type be convertible to `bool`.

TACE found a number of functions, where the annotations in code overly constrain the template arguments, such as `__unguarded_linear_insert` (STL's specifications are meaningful though, as the identified functions are helpers of algorithms requiring random access.) Static analysis tools, such as TACE, are limited to recovering syntactic requirements (such as, operations used and typenames referred to), but cannot deduce semantic details from code. For example, a forward iterator differs only semantically from an input iterator (a forward iterator can be used in multi-pass algorithms). Also, consider `find_end` for bidirectional iterators, which takes two different iterator types. The second iterator type requires only forward access and the existence of `advance(it, n)`. `n`'s possible negativity is what requires bidirectional access. Over the entire test set, TACE currently recognizes about 70% of iterator concepts correctly and unambiguously. For about 10% TACE produces a false positive match (e.g., `IntegralType`) alongside the correct iterator concept. For the input set, TACE classifies all functions and predicates (including template arguments marked as `Compare` and `StrictWeakOrding`) correctly, but due to the reason stated at the end of §VIII.E.2 does not generate the conversion requirement on result types.

VIII.G. Related Work

Dos Reis and Stroustrup [63] present an alternative idea for concept specification and checking. Their approach states concepts in terms of usage patterns, a form of requirement specification that mirrors the declarations and expressions in the template body that involve template arguments. If type checking of a concrete type against the usage pattern succeeds, then template instantiation will succeed too. In essence, TACE reverses this process and derives the requirements from C++ source code and converts them into signature based concepts.

The aim of type inference for dynamically typed languages, the derivation of type annotations from dynamically typed code, is somewhat similar to concept recovery.

Agesen et al. [6]'s dynamic type inference on SELF generates local constraints on objects from method bodies. By analyzing edges along trace graphs their analysis derives global constraints from local constraints. This kind of analysis differs from our work in a number of factors. Agesen et al start at a specific entry point of a complete program (i.e., function `main` in a C++ program). This provides concrete information on object instantiations from prototypes. Concept recovery neither depends on a single entry point, nor does the analyzed program have to be complete (instantiations are not required). Moreover, concept recovery is not concerned with finding concrete type annotations, but with finding higher level abstractions that describe multiple types (concepts). On the semantic level, C++'s type system differs from dynamically typed languages. C++ allows automatic type conversions, overloading, and function results are typed.

VIII.H. Future Work

The framework described in this section can be adapted for related analyses. In this section, we describe two extensions of our framework. One extension analyzes the actual number of a concept's functions used in instantiations. Another extension supports programmers in generifying concrete C++ functions by automating the lifting process.

VIII.H.1. Concept Utilization Analysis

The design and formalization of C++ concepts have received significant attention [83, 63]. Recent studies [179] on concept use in practice provides empirical data to language designers.

The goal of our studies is to support design decisions that relate to the automatic instantiation of concept maps with real world data. To do so, we extend our analysis tool to extract information on how many requirements of a concept are actually necessary to instantiate a template function. This preliminary study reports our results for the use of iterator requirements in STL algorithms (i.e., `algorithm` header file).

The concept-use analysis tool extends the framework presented in this chapter. Our tool uses the Pivot to extract requirements from template functions and generates the internal model. Then it reads the concept definition from a repository file. In order to get the actual requirement use, we instantiate free-standing template functions with the concept definitions. To circumvent the lack of concept support in the language, we utilize STL's naming conventions of template parameters to find the proper concept definition for each instantiation.

The analysis processes each generic function separately. For each template parameter, TACE looks up the concept from the repository according to the name of the template parameter. Then, our tool "instantiates" the extracted functional requirements with the found concepts. Note that TACE produces partial instantiations if some concepts are not present in the repository.

A functional requirement that invokes another generic function is replaced with the extracted requirements for the latter function. In order to discern among multiple implementations (e.g., various implementations of `advance`), TACE again utilizes the template parameters' name. If multiple implementations exist for the same concept (e.g., `__destroy` has two implementations, where one is optimized to treat POD value types.), TACE chooses the less specialized. The integration of functional requirements is repeated until all function invocations within the strongly connected component in the call graph have been processed.

Functions that are under constrained, such as `advance` for random access iterators (discussed in §VIII.E.3), pose the same challenges for this analysis task. TACE cannot currently handle such functions in isolation. However, TACE can handle calls to such functions by instantiating them with the types provided at the call site.

Preliminary results: We report the results for functional requirements that are defined over the iterator argument. Specifically, the base set excludes all associated type requirements, and functional requirements that are defined exclusively over associated types (and concrete types). These requirements are excluded, because the number of operations defined on, for example, the difference type would dilute the results of iterators.

We report results for forward, bidirectional, and random access iterators. The forward iterator concept contains nine functional requirements (default constructor, copy constructor, destructor, assignment, equality, inequality, pre-increment, post-increment, and dereference). The bidirectional iterator concept contains eleven requirements. It adds the pre-decrement and post-decrement operation. The random access iterator contains 20 operations. It adds requirements for less-than, greater-than, 2x iterator subtraction, 3x iterator addition, difference, and element reference (`[]`).

Table XIX shows the results obtained for some algorithms. For each concept we included the algorithm that utilizes most functional requirements. The requirement utilization is reported in absolute numbers and percentage in parenthesis.

The utilization of a concept's requirements range from 11.1% to 66.6%, from 18.2% to 45.5%, and from 20% (the true low would be 5% for `advance`) to 70%, for `Forward`, `Bidirectional`, and `Randomaccess` iterators respectively. Smaller and more fine grained concepts — in particular `UnaryFunction`, `BinaryFunction`, `UnaryPredicate`, and `BinaryPredicate` — have a 100% utilization.

Table XIX. Preliminary results on concept use

	Forward	Bidirectional	Random access
advance	1 (11.1%)	2 (18.2%)	-
find	4 (44.4%)	n/a	4 (20.0%)
lexicographical_compare	5 (55.5%)	n/a	n/a
remove	6 (66.6%)	n/a	n/a
reverse	n/a	4 (36.4%)	5 (25.0%)
rotate	6 (66.6%)	5 (45.5%)	14 (70.0%)
sort	n/a	n/a	14 (70.0%)

The results confirm the intuition that concepts hierarchies with fine grained refinements exhibit higher utilization rate (e.g., Predicates, Functions). For larger concepts, such as the iterators, we may conclude that auto concepts (or the automatic generation of concept maps) are useful. Such mechanisms help programmers avoid implementing functional requirements that will remain unused.

However, such conclusion requires validation by deeper and broader analysis. Filtering out requirements for which a concept provides a default implementation will be meaningful. For example, the random access iterator's element reference operation can be expressed by using a combination of iterator addition and dereference. Also, these preliminary results were obtained from a specific implementation of STL's `algorithm` header file. In order to provide representative data to language designers, the analysis of other template based libraries (e.g., STAPL, BOOST graph library, etc.) is essential.

The current analysis obtains concept annotations by analyzing a template parameter's name. This limits the analysis to single parametric concepts. In order to analyze more complex template libraries, TACE will need to recover other concept annotations, for example by recognizing BOOST concept checking classes.

VIII.H.2. Automatic Lifting of Non-Template Code

TACE together with the concept repository can be used to automate lifting [61] — a process to generify non template functions. Siff and Reps [162] (see also §II.C) discuss automatic generification, using predefined functional constraints. The use of a concept repository offers a promising approach to obtain more cohesive results.

CHAPTER IX

CONCLUSION AND FUTURE WORK

The premise of this dissertation is that practical programming languages evolve over time. The enhancements of a programming language are geared towards letting programmers express their intent more clearly and more concisely. I have illustrated such improvements by comparing source code using new language facilities with source code using work-around code. For example, when code with open-methods is compared with code that uses other type based dispatch techniques, code with open-methods is shorter, exhibits better performance, and is comparable in executable size.

While newly written code will benefit from language and library enhancements, existing software will not. Consequently, I have discussed source code rejuvenation, a process that assists or automates source code changes to benefit from improvements to programming languages and their libraries.

In order to be applicable to production size codes, rejuvenation requires tool support. We used the Pivot infrastructure as the basic framework for the development of rejuvenation tools. To simplify querying the internal program representation, I have described a pattern match generator. This generator helps to write analysis software that can discover replaceable implementation techniques. We have utilized the pattern matcher for the discovery of source code that would benefit from using initializer lists.

Additionally, this dissertation has discussed a technique to extract requirements on template arguments from C++ template functions. We have used a repository of concepts to identify reusable sets within the extracted set of requirements.

The work presented in this dissertation can be extended in multiple ways. The respective chapters describe possible future directions in detail. In general, tool support for a catalog of work-around techniques will be useful for the transition of source code from C++ to C++0x. In addition, the practical value of such a tool can be extended to people learning the new language facilities.

Lastly, the ability to detect coding patterns is crucial for many similar research areas. My rejuvenation framework can be adapted for related analysis tools, such as bug detection in source code.

REFERENCES

[1] D. Abrahams, A. Gurtovoy, C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series), Addison-Wesley Professional, Reading, MA, 2004.

[2] P. Abrahamsson, O. Salo, J. Ronkainen, Agile Software Development Methods: Review and Analysis, Technical Report 478, VTT Electronics, 2002.

[3] S. Adamczyk, G. Dos Reis, B. Stroustrup, Initializer List Wording, Technical Report N2531A, JTC1/SC22/WG21 C++ Standards Committee, International Standardization Organization (ISO), February 2008.

[4] Adobe System Inc., Adobe Source Library, http://opensource.adobe.com/, 2005.

[5] E. Agerbo, A. Cornils, How to Preserve the Benefits of Design Patterns, in OOPSLA '98: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications, 1998, pp. 134–143.

[6] O. Agesen, J. Palsberg, M. Schwartzbach, Type Inference of SELF, in: ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming, Springer, London, UK, 1993, pp. 247–267.

[7] R. Agrawal, L. G. Demichiel, B. G. Lindsay, Static Type Checking of Multi-Methods, in: OOPSLA '91: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications, ACM, New York, NY, 1991, pp. 113–128.

[8] A. V. Aho, M. J. Corasick, Efficient String Matching: An Aid to Bibliographic Search, Commun. ACM 18 (6), (1975) 333–340.

[9] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2001.

[10] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., S. Tobin-Hochstadt, The Fortress Language Specification version 1.0, Sun Microsystems, Inc., Santa Clara, CA, March 2008.

[11] E. Allen, J. Hallett, V. Luchangco, S. Ryu, J. Guy L. Steele, Modular Multiple Dispatch with Multiple Inheritance, in: SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing, ACM, New York, NY, 2007, pp. 1117–1121.

[12] E. Amiel, O. Gruber, E. Simon, Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables, in: OOPSLA '94: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, New York, NY, 1994, pp. 244–258.

[13] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, L. Rauchwerger, STAPL: A Standard Template Adaptive Parallel C++ Library, in: LCPC '01: Proceedings of the 14th International Conference on Languages and Compilers for Parallel Computing, Cumberland Falls, KY, Vol. 2624 of LNCS, Springer, 2001, pp. 193–208.

[14] K. Arnold, J. Gosling, D. Holmes, The Java Programming Language, 4th ed., Prentice-Hall, Inc., Upper Saddle River, NJ, 2005.

[15] M. H. Austern, Generic Programming and the STL: Using and Extending the C++ Standard Template Library, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1998.

[16] O. S. Bagge, M. Haveraaen, Domain-Specific Optimisation with User-Defined Rules in CodeBoost, in: J.-L. Giavitto, P.-E. Moreau (eds.), in RULE'03: Proceedings of the 4th International Workshop on Rule-Based Programming, Vol. 86/2 of Electronic Notes in Theoretical Computer Science, Elsevier, Valencia, Spain, 2003, pp. 119–133.

[17] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, E. Visser, Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++ Programs, in: D. Binkley, P. Tonella (eds.), in: SCAM '04: The 3rd International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society Press, Amsterdam, The Netherlands, 2003, pp. 65–75.

[18] I. Balaban, F. Tip, R. Fuhrer, Refactoring Support for Class Library Migration, in: OOPSLA '05: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications, ACM, New York, NY, 2005, pp. 265–279.

[19] Z. Balanyi, R. Ferenc, Mining Design Patterns from C++ Source Code, in: ICSM '03: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, 2003, pp. 305–314.

[20] E. Balland, Y. Boichut, T. Genet, P.-E. Moreau, Towards an Efficient Implementation of Tree Automata Completion, in: AMAST '08: Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 67–82.

[21] G. Barnes, A Method for Implementing Lock-Free Shared-Data Structures, in: SPAA '93: Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, NY, 1993, pp. 261–270.

[22] J. J. Barton, L. R. Nackman, Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1994.

[23] I. D. Baxter, DMS: Program Transformations for Practical Scalable Software Evolution, in: IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution, ACM, New York, NY, 2002, pp. 48–51.

[24] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone Detection Using Abstract Syntax Trees, in: ICSM '98: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, 1998, pp. 368–377.

[25] P. Becker, The C++ Standard Library Extensions: A Tutorial and Reference, 1st ed., Addison-Wesley Professional, Boston, MA, 2006.

[26] The Boost C++ Libraries, http://www.boost.org/, retrieved July 2008.

[27] P. Becker, Working Draft, Standard for Programming Language C++, Technical Report N2914, JTC1/SC22/WG21 C++ Standards Committee, International Standardization Organization (ISO), June 2009.

[28] T. Becker, Type Erasure in C++: The Glue between Object Oriented and Generic Programming, in: K. Davis, J. Striegnitz (eds.), in: MPOOL '07: Proceedings of the Multiparadigm Programming Workshop at ECOOP, 2007.

[29] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, D. Engler, A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World, Commun. ACM 53 (2), (2010) 66–75.

[30] L. Bettini, S. Capecchi, B. Venneri, Double Dispatch in C++, Software - Practice and Experience 36 (6), (2006) 581–613.

[31] G. M. Birtwistle, O. Dahl, B. Myhrhaug, K. Nygaard, Simula BEGIN, Auerbach Press, Philadelphia, 1973.

[32] D. Bonniot, B. Keller, F. Barber, The Nice User's Manual, 2008. http://nice.sourceforge.net/manual.html, retrieved September 2009.

[33] L. Bourdev, J. Järvi, Efficient Run-Time Dispatching in Generic Programming with Minimal Code Bloat, in: LCSD '06: Workshop of Library-Centric Software Design at OOPSLA'06, Chalmers University, Göteborg, Sweden, 2006, pp. 15–24.

[34] J. Boyland, G. Castagna, Type-Safe Compilation of Covariant Specialization: A Practical Case, in: ECCOP '96: Proceedings of the 10th European Conference on Object-Oriented Programming, Springer-Verlag, London, UK, 1996, pp. 3–25.

[35] J. Boyland, G. Castagna, Parasitic Methods: An Implementation of Multi-Methods for Java, in: OOPSLA '97: Proceedings of the 12th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, 1997, pp. 66–76.

[36] M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.16: Components for Transformation Systems, in: PEPM '06: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM, New York, NY, 2006, pp. 95–99.

[37] W. J. Brown, R. C. Malveau, H. W. McCormick, III, T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, John Wiley & Sons, Inc., New York, NY, 1998.

[38] K. Bruce, L. Cardelli, G. Castagna, G. T. Leavens, B. Pierce, On Binary Methods, Theor. Pract. Object Syst. 1 (3) (1995) 221–242.

[39] S. Ceri, G. Gottlob, L. Tanca, What You Always Wanted to Know about Datalog (and Never Dared to Ask), IEEE Transactions on Knowledge and Data Engineering 1, 1989, pp. 146–166.

[40] D. Chamberlin, XQuery: An XML Query Language, IBM Syst. J. 41 (4) (2002) 597–615.

[41] C. Chambers, Object-Oriented Multi-Methods in Cecil, in: ECOOP '92: Proceedings of the European Conf. on Object-Oriented Programming, Springer-Verlag, London, UK, 1992, pp. 33–56.

[42] C. Chambers, The Cecil Language: Specification and Rationale v3.2, University of Washington, Seattle, WA, 2004.

[43] C. Chambers, The Diesel Language, Specification and Rationale v0.2, University of Washington, Seattle, WA, 2006.

[44] C. Chambers, W. Chen, Efficient Multiple and Predicated Dispatching, in: OOPSLA '99: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, 1999, pp. 238–255.

[45] B. Chelf, D. Engler, S. Hallem, How to Write System-Specific, Static Checkers in Metal, in: PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT

Workshop on Program Analysis for Software Tools and Engineering, ACM, New York, NY, 2002, pp. 51–60.

[46] K. Chen, D. Wagner, Large-scale Analysis of Format String Vulnerabilities in Debian Linux, in: PLAS '07: Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, ACM, New York, NY, 2007, pp. 75–84.

[47] C. Cleeland, D. Schmidt, T. Harrison, External Polymorphism – an Object Structural Pattern for Transparently Extending Concrete Data Types, in: Pattern Languages of Program Design (R. Martin, F. Buschmann, and D. Riehle, eds.), Addison-Wesley, Reading, MA, 1997.

[48] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein, MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java, in: OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, 2000, pp. 130–145.

[49] C. Clifton, T. Millstein, G. T. Leavens, C. Chambers, MultiJava: Design Rationale, Compiler Implementation, and Applications, ACM Trans. Program. Lang. Syst. 28 (3) (2006) 517–575.

[50] M. L. Collard, J. I. Maletic, A. Marcus, Supporting Document and Data Views of Source Code, in: DocEng '02: Proceedings of the 2002 ACM Symposium on Document Engineering, ACM, New York, NY, 2002, pp. 34–41.

[51] S. Cook, D. Dechev, P. Pirkelbauer, IQL: An IPR Query Language, Course Project, C++ Template Meta Programming, Texas A&M University, College Station, TX, 2004.

[52] J. R. Cordy, Source Transformation, Analysis and Generation in TXL, in: PEPM '06: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM, New York, NY, 2006, pp. 1–11.

[53] J. R. Cordy, The TXL Source Transformation Language, Sci. Comput. Program. 61 (3) (2006) 190–210.

[54] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, 2001.

[55] L. Damas, R. Milner, Principal Type-Schemes for Functional Programs, in: POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, 1982, pp. 207–212.

[56] D. Dechev, P. Pirkelbauer, N. Rouquette, B. Stroustrup, Semantically Enhanced Containers for Concurrent Real-Time Systems, Engineering of Computer-Based Systems, IEEE International Conference, 2009, pp. 48–57.

[57] D. Dechev, P. Pirkelbauer, B. Stroustrup, Lock-Free Dynamically Resizable Arrays., in: A. A. Shvartsman (ed.), OPODIS, Vol. 4305 of Lecture Notes in Computer Science, Springer, 2006, pp. 142–156.

[58] D. Dechev, N. Rouquette, P. Pirkelbauer, B. Stroustrup, Verification and Semantic Parallelization of Goal-Driven Autonomous Software, in: Autonomics '08: Proceedings of 2nd International Conference on Autonomic Computing and Communication Systems, 2008, pp. 33:1–33:8.

[59] J. C. Dehnert, A. A. Stepanov, Fundamentals of Generic Programming, in:

Selected Papers from the International Seminar on Generic Programming, Springer-Verlag, London, UK, 2000, pp. 1–11.

[60] D. v. Dincklage, A. Diwan, Converting Java Classes to Use Generics, in: OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, New York, NY, 2004, pp. 1–14.

[61] G. Dos Reis, Personal Communication, December 2009.

[62] G. Dos Reis, B. Stroustrup, A C++ Formalism, Technical Report N1885, JTC1/SC22/WG21 C++ Standards Committee, International Standardization Organization (ISO), 2005.

[63] G. Dos Reis, B. Stroustrup, Specifying C++ Concepts, in: POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, New York, NY, 2006, pp. 295–308.

[64] G. Dos Reis, B. Stroustrup, A Principled, Complete, and Efficient Representation of C++, in: M. Suzuki, H. Hong, H. Anai, C. Yap, Y. Sato, H. Yoshida (eds.), The Joint Conference of ASCM 2009 and MACIS 2009, Vol. 22 of MI Lecture Note Series, COE, Fukuoka, Japan, 2009, pp. 151–166.

[65] G. Dos Reis, B. Stroustrup, A. Meredith, Axioms: Semantics Aspects of C++ Concepts, Technical Report N2887, JTC1/SC22/WG21 C++ Standards Committee, International Standardization Organization (ISO), June 2009.

[66] S. Ducasse, M. Lanza, Towards a Methodology for the Understanding of Object-Oriented Systems, Technique et Science Informatiques, Hermes Publications, Vol. 6 (2001), 539–566.

[67] Eclipse Framework, MoDisco Project, http://www.eclipse.org/gmt/modisco/, retrieved September 2009.

[68] ECMA, The C# Language Specification, Technical Report ECMA-334, ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, June 2006.

[69] Edison Design Group, C++ Front End, http://www.edg.com/, retrieved July 2008.

[70] R. Ferenc, I. Siket, T. Gyimóthy, Extracting Facts from Open Source Software, in: ICSM '04: Proceedings of the 20th International Conference on Software Maintenance, IEEE Computer Society, 2004, pp. 60–69.

[71] C. B. Flynn, D. Wonnacott, Reconciling Encapsulation and Dynamic Dispatch via Accessory Functions, Technical Report 387, Haverford College, 1999.

[72] B. Foote, R. E. Johnson, J. Noble, Efficient Multimethods in a Single Dispatch Language, in: ECOOP '05: Proceedings of the European Conference on Object-Oriented Programming, Glasgow, Scotland, 2005, pp. 337–361.

[73] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1999.

[74] K. Fraser, Practical Lock-Freedom, Technical Report UCAM-CL-TR-579, Unversity of Cambridge, February 2004.

[75] E. Fredkin, Trie Memory, Commun. ACM 3 (9) (1960) 490–499.

[76] Free Software Foundation, GCC, the GNU Compiler Collection, http://gcc.gnu.org/, retrieved April 2010.

[77] C. Frost, T. Millstein, Modularly Typesafe Interface Dispatch in JPred, in: FOOL/WOOD '06: 2006 International Workshop on Foundations and Development of Object-Oriented Languages, Charleston, SC, 2006.

[78] J. Fulara, K. Jakubczyk, Practically Applicable Formal Methods, in: J. v. Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, B. Rumpe (eds.), SOFSEM, Vol. 5901 of Lecture Notes in Computer Science, Springer, 2010, pp. 407–418.

[79] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995.

[80] M. Gibbs, B. Stroustrup, Fast Dynamic Casting, Softw. Pract. Exper. 36 (2) (2006) 139–156.

[81] J. Y. Gil, I. Maman, Micro Patterns in Java Code, in: OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, New York, NY, 2005, pp. 97–116.

[82] A. Goldberg, D. Robson, Smalltalk-80: The Language and Its Implementation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1983.

[83] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, A. 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, ACM Press, New York, NY, 2006, pp. 291–310.

[84] D. Gregor, B. Stroustrup, J. Siek, J. Widman, Proposed Wording for Concepts

(rev 4), Technical Report N2501, JTC1/SC22/WG21 C++ Standards Committee, International Standardization Organization (ISO), February 2008.

[85] G. Greif, Chinese Dispatch – Unpublished Presentation Notes, Dylan Hackers Conference, Berlin, Germany, July 2002.

[86] D. Grune, C. J. H. Jacobs, Parsing Techniques (Monographs in Computer Science), Springer-Verlag New York, Inc., Secaucus, NJ, 2006.

[87] T. L. Harris, A Pragmatic Implementation of Non-Blocking Linked-Lists, in: DISC '01, Springer-Verlag, London, UK, 2001, pp. 300–314.

[88] M. Harsu, Re-Engineering Legacy Software through Language Conversion, Ph.D. Thesis, University of Tampere, Tampere, Finland, 2000.

[89] D. Hendler, N. Shavit, L. Yerushalmi, A Scalable Lock-Free Stack Algorithm, in: SPAA '04: Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures, ACM Press, New York, NY, 2004, pp. 206–215.

[90] M. Herlihy, A Methodology for Implementing Highly Concurrent Data Objects, ACM Trans. Program. Lang. Syst. 15 (5) (1993) 745–770.

[91] M. Herlihy, V. Luchangco, P. Martin, M. Moir, Nonblocking Memory Management Support for Dynamic-Sized Data Structures, ACM Trans. Comput. Syst. 23 (2), 2005, pp. 146–196.

[92] D. Hovemeyer, W. Pugh, Finding Bugs is Easy, in: OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM, New York, NY, 2004, pp. 132–136.

[93] S. S. Huang, Y. Smaragdakis, Expressive and Safe Static Reflection with MorphJ, in: PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, 2008, pp. 79–89.

[94] Intel, IA-32 Intel Architecture Software Developer's Manual, Vol. 3: System Programming Guide, Mt. Prospect, IL, 2004.

[95] Intel, Reference for Intel Threading Building Blocks, version 1.0, Mt. Prospect, IL, April 2006.

[96] International Standardization Organization, ISO/IEC 10918-1:1994: Information Technology –- Digital Compression and Coding of Continuous-Tone Still Images: Requirements and Guidelines, pub-ISO, pub-ISO:adr, Geneva, Switzerland, 1994.

[97] ISO/IEC 14882 International Standard, Programming Languages: C++, American National Standards Institute, Washington, DC, 1998.

[98] The Itanium C++ ABI, http://www.codesourcery.com/public/cxx-abi/abi.html, retrieved August 2006.

[99] M. Iwaihara, Y. Inoue, Bottom-up Evaluation of Logic Programs Using Binary Decision Diagrams, in: ICDE '95: Proceedings of the 11th International Conference on Data Engineering, IEEE Computer Society, Washington, DC, 1995, pp. 467–474.

[100] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, J. Siek, Algorithm Specialization in Generic Programming: Challenges of Constrained Generics in C++, in:

PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, 2006, pp. 272–282.

[101] J. Järvi, M. A. Marcus, J. N. Smith, Library Composition and Adaptation Using C++ Concepts, in: GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, ACM Press, New York, NY, 2007, pp. 73–82.

[102] J. Järvi, J. Willcock, H. Hinnant, A. Lumsdaine, Function Overloading Based on Arbitrary Properties of Types., C/C++ Users Journal (21(6)), CMP Media, Manhasset, NY, (2003) 25–32.

[103] M. Jones, Typing Haskell in Haskell, http://www.cse.ogi.edu/ mpj/thih/, November 2000, retrieved September 2010.

[104] L. C. L. Kats, M. Bravenboer, E. Visser, Mixing Source and Bytecode: A Case for Compilation by Normalization, in: OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, ACM, New York, NY, 2008, pp. 91–108.

[105] J. Kerievsky, Refactoring to Patterns, Pearson Higher Education, Upper Saddle River, NJ, 2004.

[106] S. Kuznetsov, S. Obiedkov, Algorithms for the Construction of Concept Lattices and Their Diagram Graphs, in: Principles of Data Mining and Knowledge Discovery, Vol. 2168 of LNCS, Springer-Verlag, 2001, pp. 289–300.

[107] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, C. Unkel, Context-Sensitive Program Analysis as Database Queries, in: PODS

'05: Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM, New York, NY, 2005, pp. 1–12.

[108] R. Lämmel, Towards Generic Refactoring, in: RULE '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming, ACM, New York, NY, 2002, pp. 15–28.

[109] A. Langer, K. Kreft, Secrets of Equals, C/C++ Users Journal (20(4)), Java Supplement, CMP Media, Manhasset, NY, April 2002.

[110] C. Lattner, V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, in: CGO'04: Proceedings of the 2004 International Symposium on Code Generation and Optimization, Palo Alto, California, 2004, pp. 75–86.

[111] B. Liskov, Keynote Address - Data Abstraction and Hierarchy, in: OOPSLA '87: Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (Addendum), ACM Press, New York, NY, 1987, pp. 17–34.

[112] Lockheed Martin, Joint Strike Fighter, Air Vehicle, C++ Coding Standard, Technical Report, Lockheed Martin, Fort Worth, TX, December 2005.

[113] J. I. Maletic, M. L. Collard, A. Marcus, Source Code Files as Structured Documents, in: IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension, IEEE Computer Society, Washington, DC, 2002, pp. 289–292.

[114] M. Marcus, J. Järvi, S. Parent, Runtime Polymorphic Generic Programming–-Mixing Objects and Concepts in ConceptC++, in: K. Davis, J. Striegnitz (eds.),

MPOOL '07: Proceedings of the Multiparadigm Programming Workshop at ECOOP, 2007.

[115] M. Martin, B. Livshits, M. S. Lam, Finding Application Errors and Security Flaws Using PQL: A Program Query Language, in: OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, New York, NY, 2005, pp. 365–383.

[116] B. McCloskey, E. Brewer, ASTEC: A New Approach to Refactoring C, SIGSOFT Softw. Eng. Notes 30 (5) (2005), 21–30.

[117] S. McPeak, Elsa: An Elkhound Based C++ Parser, August 2008. http://scottmcpeak.com/elkhound/, retrieved May 2010.

[118] S. G. McPeak, Elkhound: A Fast, Practical GLR Parser Generator, Technical Report UcB/cSD-2-1214, UC Berkeley, CA, 2002.

[119] T. Mens, T. Tourwé, A Survey of Software Refactoring, IEEE Trans. Softw. Eng. 30 (2) (2004), 126–139.

[120] J. Merrill, D. Vandevoorde, Initializer Lists — Alternative Mechanism and Rationale, Technical Report N2640, JTC1/SC22/WG21 C++ Standards Committee, International Standardization Organization (ISO), 2008.

[121] R. Metzger, Z. Wen, Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization, MIT Press, Cambridge, MA, 2000.

[122] B. Meyer, Eiffel: The Language, Prentice-Hall, Inc., Upper Saddle River, NJ, 1992.

[123] M. M. Michael, High Performance Dynamic Lock-Free Hash Tables and List-Based Sets, in: SPAA '02: Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, NY, 2002, pp. 73–82.

[124] M. M. Michael, CAS-Based Lock-Free Algorithm for Shared Deques, in: Euro-Par '03, Vol. 2790 of LNCS, 2003, pp. 651–660.

[125] M. M. Michael, Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects, IEEE Trans. Parallel Distrib. Syst. 15 (6) (2004) 491–504.

[126] M. M. Michael, M. L. Scott, Correction of a Memory Management Method for Lock-Free Data Structures, Technical Report 599, University of Rochester, Rochester, NY, 1995.

[127] A. Miller, Proposed Language Extensions for Java 7, http://tech.puredanger.com/java7/, retrieved July 2009.

[128] T. Millstein, C. Chambers, Modular Statically Typed Multimethods, Information and Computation 175 (1) (2002) 76–118.

[129] T. Millstein, M. Reay, C. Chambers, Relaxed MultiJava: Balancing Extensibility and Modular Typechecking, in: OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN Conf. on Object-Oriented Programing, Systems, Languages, and Applications, ACM Press, New York, NY, 2003, pp. 224–240.

[130] T. D. Millstein, C. Chambers, Modular Statically Typed Multimethods, in: ECOOP '99: Proceedings of the 13th European Conf. on Object-Oriented Programming, Vol. 1628 of LNCS, Springer-Verlag, London, UK, 1999, pp. 279–303.

[131] R. Muschevici, A. Potanin, E. Tempero, J. Noble, Multiple Dispatch in Practice, in: OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, ACM, New York, NY, 2008, pp. 563–582.

[132] N. C. Myers, Traits: A New and Useful Template Technique, in: C++ Gems, SIGS Publications, Inc., New York, NY, 1996, pp. 451–457.

[133] W. F. Opdyke, Refactoring Object-Oriented Frameworks, Ph.D. Thesis, University of Illinois at Urbana-Champaign, Champaign, IL, uMI Order No. GAX93-05645, 1992.

[134] W. F. Opdyke, R. E. Johnson, Creating Abstract Superclasses by Refactoring, in: CSC '93: Proceedings of the 1993 ACM Conference on Computer Science, ACM, New York, NY, 1993, pp. 66–73.

[135] E. Panizzi, B. Pastorelli, Multimethods and Separate Static Typechecking in a Language with C++-Like Object Model, The Computing Research Repository (CoRR) cs.PL/0005033, 2000.

[136] S. Parent, Beyond Objects: Understanding the Software We Write, Presentation at C++ Connections, November 2005, http://stlab.adobe.com/wiki/index.php/Image:Regular_object_presentation.pdf, retrieved July 2006.

[137] S. Parent, Concept-Based Runtime Polymorphism, Presentation at BoostCon, May 2007, http://stlab.adobe.com/wiki/index.php/Image:Boost_poly.pdf, retrieved October 2007.

[138] C. Parnin, C. Görg, O. Nnadi, A Catalogue of Lightweight Visualizations to

Support Code Smell Inspection, in: SoftVis '08: Proceedings of the 4th ACM Symposium on Software Visualization, ACM, New York, NY, 2008, pp. 77–86.

[139] T. J. Parr, R. W. Quong, ANTLR: A Predicated-LL(k) Parser Generator, Software Practice and Experience 25 (1994) 789–810.

[140] J. Peterson, M. Jones, Implementing Type Classes, in: PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, ACM, New York, NY, 1993, pp. 227–236.

[141] B. C. Pierce, Types and Programming Languages, MIT Press, Cambridge, MA, 2002.

[142] P. Pirkelbauer, D. Dechev, B. Stroustrup, Source Code Rejuvenation Is Not Refactoring, in: J. v. Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, B. Rumpe (eds.), SOFSEM, Vol. 5901 of Lecture Notes in Computer Science, Springer, 2010.

[143] P. Pirkelbauer, S. Parent, M. Marcus, B. Stroustrup, Runtime Concepts for the C++ Standard Template Library, in: SAC '08: Proceedings of the 2008 ACM Symposium on Applied Computing, ACM, New York, NY, 2008, pp. 171–177.

[144] P. Pirkelbauer, S. Parent, M. Marcus, B. Stroustrup, Dynamic Algorithm Selection for Runtime Concepts, Science of Computer Programming 75 (9) (2010) 773–786.

[145] P. Pirkelbauer, Y. Solodkyy, B. Stroustrup, Open Multi-Methods for C++, in: GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, ACM Press, New York, NY, 2007, pp. 123–134.

[146] P. Pirkelbauer, Y. Solodkyy, B. Stroustrup, Report on Language Support for Multi-Methods and Open-Methods for C++, Technical Report N2216, JTC1/SC22/WG21 C++ Standards Committee, International Standardization Organization (ISO), 2007.

[147] P. Pirkelbauer, Y. Solodkyy, B. Stroustrup, Design and Evaluation of C++ Open Multi-Methods, Science of Computer Programmming 75 (7) (2010) 638–667.

[148] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. de Supinski, D. J. Quinlan, Transforming MPI Source Code Based on Communication Patterns, Future Gener. Comput. Syst. 26 (1) (2010) 147–154.

[149] D. Quinlan, M. Schordan, Q. Yi, B. R. d. Supinski, Semantic-Driven Parallelization of Loops Operating on User-Defined Containers, in: LCPC'03, Proceedings of the Workshop on Languages and Compilers for Parallel Computing, LNCS 2958, Springer, 2004, pp 524–538.

[150] D. J. Quinlan, R. W. Vuduc, G. Misherghi, Techniques for Specifying Bug Patterns, in: PADTAD '07: Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging, ACM, New York, NY, 2007, pp. 27–35.

[151] R. Ramesh, I. V. Ramakrishnan, Nonlinear Pattern Matching in Trees, J. ACM 39 (2) (1992) 295–316.

[152] D. M. Ritchie, The Development of the C Language, in: HOPL-II: The 2nd ACM SIGPLAN Conference on History of Programming Languages, ACM, New York, NY, 1993, pp. 201–208.

[153] A. Robison, Personal Communication, April 2006.

[154] J. E. Sammet, The Early History of COBOL, SIGPLAN Not. 13 (8) (1978) 121–161.

[155] M. Schordan, D. Quinlan, A Source-to-Source Architecture for User-Defined Optimizations, in: JMLC'03: Joint Modular Languages Conference, Vol. 2789 of LNCS, Springer, 2003, pp. 214–223.

[156] O. Shalev, N. Shavit, Split-Ordered Lists: Lock-Free Extensible Hash Tables, in: PODC '03: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing, ACM Press, New York, NY, 2003, pp. 102–111.

[157] A. Shalit, D. Moon, O. Starbuck: The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language (Apple Press Series), Addison-Wesley Publishing Co., New York, NY, 1996.

[158] N. Shi, R. A. Olsson, Reverse Engineering of Design Patterns from Java Source Code, in: ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, Washington, DC, 2006, pp. 123–134.

[159] D. Shopyrin, Multimethods Implementation in C++ Using Recursive Deferred Dispatching, IEEE Softw. 23 (3) (2006) 62–73.

[160] J. Siek, A. Lumsdaine, Concept Checking: Binding Parametric Polymorphism in C++, in: 1st Workshop on C++ Template Programming, Erfurt, Germany, 2000.

[161] J. G. Siek, L.-Q. Lee, A. Lumsdaine, The Boost Graph Library: User Guide and Reference Manual, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2002.

[162] M. Siff, T. Reps, Program Generalization for Software Reuse: from C to C++, in: SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM, New York, NY, 1996, pp. 135–146.

[163] M. Siff, T. Reps, Identifying Modules via Concept Analysis, IEEE Trans. Softw. Eng. 25 (6) (1999) 749–768.

[164] M. B. Siff, Techniques for Software Renovation, Ph.D. Thesis, University of Wisconsin at Madison, 1999.

[165] Y. Smaragdakis, D. S. Batory, Mixin-Based Programming in C++, in: GCSE '00: Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering-Revised Papers, Springer-Verlag, London, UK, 2001, pp. 163–177.

[166] J. Smith, Draft Proposal for Adding Multimethods to C++, Technical Report N1463, JTC1/SC22/WG21 C++ Standards Committee, International Standardization Organization (ISO), 2003.

[167] A. Snyder, Encapsulation and Inheritance in Object-Oriented Programming Languages, in: OOPLSA '86: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, ACM, New York, NY, 1986, pp. 38–45.

[168] G. L. Steele, Jr, Common LISP: The Language (2nd ed.), Digital Press, Newton, MA, 1990.

[169] A. Stepanov, P. McJones, Elements of Programming, Addison-Wesley Professional, Reading, MA, 2009.

[170] B. Stroustrup, The Design and Evolution of C++, ACM Press/Addison-Wesley Publishing Co., New York, NY, 1994.

[171] B. Stroustrup, The C++ Programming Language, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2000.

[172] B. Stroustrup, Abstraction and the C++ Machine Model, in: ICESS'04: 1st International Conference on Embedded Software and Systems, Vol. 3605 of LNCS, Springer, 2004, pp. 1–13.

[173] B. Stroustrup, The Design of C++0x, C/C++ Users Journal (23(5)), CMP Media, Manhasset, NY, May 2005.

[174] B. Stroustrup, Evolving a Language in and for the Real World: C++ 1991-2006, in: HOPL III: Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages, ACM, New York, NY, 2007, pp. 4-1–4-59.

[175] B. Stroustrup, The C++0x "Remove Concept" Decision, Dr.Dobb's Journal 92, August 2009, http://www.drdobbs.com/cpp/218600111, retrieved August 2009.

[176] B. Stroustrup, Expounds on Concepts and the Future of C++, Interview with Danny Kalev, www.devx.com/cplus/Article/42448/0/page/1, August 2009, retrieved October 2009.

[177] B. Stroustrup, G. Dos Reis, Supporting SELL for High-Performance Computing, in: 18th International Workshop on Languages and Compilers for Parallel Computing, Vol. 4339 of LNCS, Springer-Verlag, 2005, pp. 458–465.

[178] H. Sundell, P. Tsigas, Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap, in: OPODIS 2004: Principles of Distributed Systems, 8th Int. Conf., Vol 3544 of LNCS, 2005, pp. 240–255.

[179] A. Sutton, Empirically Motivating C++ Concepts, Invited Talk, Department of Computer Science and Engineering at Texas A&M University, April 2010.

[180] A. Sutton, J. I. Maletic, Recovering UML Class Models from C++: A Detailed Explanation, Inf. Softw. Technol. 49 (3) (2007) 212–229.

[181] A. Sutton, J. I. Maletic, Automatically Identifying C++0x Concepts in Function Templates, in: ICSM '08: 24th IEEE International Conference on Software Maintenance, 2008, Beijing, China, IEEE, 2008, pp. 57–66.

[182] W. Tansey, E. Tilevich, Annotation Refactoring: Inferring Upgrade Transformations for Legacy Applications, in: OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, Vol. 43, ACM, New York, NY, 2008, pp. 295–312.

[183] D. Thomas, A. Hunt, Programming Ruby: The Pragmatic Programmer's Guide, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2000.

[184] S. Thompson, G. Brat, Verification of C++ Flight Software with the MCP Model Checker, in: IEEE Aerospace Conference, Big Sky, MT, 2008, pp. 1–9.

[185] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, V. Sundaresan, Soot - a Java bytecode Optimization Framework, in: CASCON '99: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, IBM Press, 1999, pp. 13–23.

[186] G. van Rossum, The Python Language Reference Manual, Network Theory Ltd., Bristol, UK, 2003, paperback.

[187] T. L. Veldhuizen, Arrays in Blitz++, in: ISCOPE '98: Proceedings of the 2nd

International Symposium on Computing in Object-Oriented Parallel Environments, Springer-Verlag, London, UK, 1998, pp. 223–230.

[188] A. Venet, A Practical Approach towards Formal Software Verification through Static Analysis, ADA letters XXVIII (1) (2008) 92–95.

[189] Video Codec and Pixel Format Definitions, http://www.fourcc.org/, retrieved February 2007.

[190] J. Visser, Visitor Combination and Traversal Control, in: OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, 2001, pp. 270–282.

[191] P. Wadler, 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, ACM, New York, NY, 1989, pp. 60–76.

[192] L. Wagner, Traversal, Case Analysis, and Lowering for C++ Program Analysis, M.S. Thesis, Texas A&M University, College Station, TX, 2009.

[193] D. Wasserrab, T. Nipkow, G. Snelting, F. Tip, An Operational Semantics and Type Safety Proof for Multiple Inheritance in C++, in: OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, New York, NY, 2006, pp. 345–362.

[194] J. J. Willcock, A. Lumsdaine, D. J. Quinlan, Reusable, Generic Program Analyses and Transformations, in: GPCE '09: Proceedings of the Eighth International Conference on Generative Programming and Component Engineering,

ACM, New York, NY, 2009, pp. 5–14.

[195] D. Wonnacott, Using Accessory Functions to Generalize Dynamic Dispatch in Single-Dispatch Object-Oriented Languages, in: COOTS'01: 6th USENIX Conference on Object-Oriented Technologies and Systems, San Antonio, TX, USENIX, 2001, pp. 93–102.

[196] A. Wöß, M. Löberbauer, H. Mössenböck, LL(1) Conflict Resolution in a Recursive Descent Compiler Generator, in: JMLC'03: Joint Modular Languages Conference, Vol. 2789 of LNCS, Springer, 2003, pp. 192–201.

APPENDIX A

INPUT LANGUAGE OF THE PATTERN MATCH GENERATOR

Extensions to the XPR grammar that support the definition of patterns. The start production is `query_seq`.

```
query_seq
   : query_obj query_seq
   | /* empty */
   ;

query_obj
   : PATTERN
      query_head
      opt_hole_decls
      '{' pattern_seq '}'
      ;

hole_init_entry
  : type_id hole_init
  | built_in_type
  | literal
  | ELLIPSIS
  ;

hole_init_seq
  : hole_init_entry ',' hole_init_seq
  | hole_init_entry
  ;

hole_init
  : '(' hole_init_seq ')'
  | '(' ')'
  | /* empty */
  ;

hole_decl
   : ID ':' hole_init_entry ';'
   ;

opt_hole_decls
   : hole_decl opt_hole_decls
   | /* empty */
   ;

list_suffix
   : ELLIPSIS
   | /* empty */
   ;

action
   : PAM_QUERY
   | PAM_QUERY block
   ;

expr_pattern
   : expr action
   ;

pam_stmt
```

```
   : expr eod
   ;

stmt_pattern
   : pam_stmt action
   ;

type_pattern
   : type_name action
   ;

exprlist_pattern
   : pam_expr_list action
   ;

expr_pattern_seq
   : expr_pattern ';' expr_pattern_seq
   | /* empty */
   ;

exprlist_pattern_seq
   : exprlist_pattern ';' exprlist_pattern_seq
   | /* empty */
   ;

stmt_pattern_seq
   : stmt_pattern ';' stmt_pattern_seq
   | /* empty */
   ;

type_pattern_seq
   : type_pattern ';' type_pattern_seq
   | /* empty */
   ;

pattern_seq
   : PAM_EXPR expr_pattern_seq
   | PAM_EXPRLIST exprlist_pattern_seq
   | PAM_STMT stmt_pattern_seq
   | PAM_TYPE type_pattern_seq
   ;

path_selector
   : /* empty.
          The rule's semantic action pushes a token indicating the parse
          context (PAM_EXPR, PAM_EXPRLIST, PAM_STMT, PAM_TYPE) back on the
          token stream. The token determines the alternative in pattern_seq. */
   ;

query_param_entry
   : ID ':' type_name
   | ID ':' type_name '=' value_expr
   ;

query_param_seq
   : query_param_entry ',' query_param_seq
   | query_param_entry
   ;

query_params
   : '(' query_param_seq ')'
   | '(' ')'
   ;

query_head
   : ID '<' type_id '>' query_params
   ;

pam_expr_list
      : expr_list list_suffix
      | list_suffix
      ;
```

XPR grammer for parsing expressions and statements. To support the definition of patterns on expression lists, any reference to the rule `expr_list` has been replaced by a reference to `pam_expr_list`.

```
stmts
      : stmt_list
      | /* empty */
      ;

stmt_list
      : expr eod
      | expr eod stmt_list
      ;

eod : BCPL_COMMENT '{' attributes '}'
      | ';'
      ;

type_name
      : basic_type
      | type_opers basic_type
      ;

type_opers
      : type_oper
      | type_oper type_opers
      ;

type_oper
      : '*'
      | '&'
      | CONST
      | VOLATILE
      | '[' expr ']'
      | '[' ']'
      | '(' function_parameters ')' opt_spec
      | TL template_parameters TG
      ;

opt_spec
      : THROW '(' ')'
      | THROW '(' ELLIPSIS ')'
      | THROW '(' type_id_list ')'
      | /* no exception specification */
      ;

type_id_list
      : type_id
      | type_id ',' type_id_list
      ;

template_parameters
      : template_parameter
      | template_parameter ',' template_parameters
      | /* empty */
      ;

template_parameter
      : ID
      | ID ':' type_id
      | ID ':' TYPENAME
      | TYPENAME
      ;

function_parameters
      : function_parameter
      | function_parameter ',' function_parameters
      | /* empty */
      ;

function_parameter
      : type_name opt_initializer
      | id ':' type_name opt_initializer
      | ELLIPSIS
      ;

built_in_type
      : VOID
      | BOOL
      | CHAR
      | INT
      | SHORT
      | FLOAT
      | LONG
      | LONG LONG /* not C++ */
      | DOUBLE
      | LONG DOUBLE
      | SIGNED basic_type
      | UNSIGNED CHAR
      | UNSIGNED INT
      | UNSIGNED SHORT
      | UNSIGNED LONG
      | UNSIGNED LONG LONG /* not C++ */
      ;

basic_type
      : built_in_type
      | type_id
      ;

special_id
      : OPERATOR cast_oper TL basic_type TG
      | OPERATOR cast_oper TL basic_type TG AT INT_VALUE
      | OPERATOR oper
      | OPERATOR oper AT INT_VALUE
      | special_name
      ;

id
      : ID
      | ID AT INT_VALUE
      | THIS
      | THIS AT INT_VALUE
      ;

tid
      : id
      | id TL pam_expr_list TG
      ;

type_id
      : tid
      | QUAL tid
      | type_id QUAL tid
      ;

eid
      : special_id
      | tid
      ;

expr_id
      : eid
      | QUAL eid
      | type_id QUAL eid
      ;
```

```
arg_list
      : '(' pam_expr_list ')'
      ;

expr_list
      : expr
      | expr ',' expr_list
      ;

primary_expr
      : literal
      | expr_id
      | '(' expr ')'
      ;

postfix_expr
      : primary_expr
      | postfix_expr '[' expr ']'
      | postfix_expr arg_list
      | postfix_expr DOT type_id
      | postfix_expr DOT_REF type_id
      | postfix_expr ARROW type_id
      | postfix_expr ARROW_REF type_id
      | postfix_expr INCR
      | postfix_expr DECR
      ;

unary_oper
      : '*'
      | '&'
      | '+'
      | '−'
      | INCR
      | DECR
      | '~'
      | '!'
      ;

value_expr
      : unary_oper value_expr
      | postfix_expr
      | postfix_expr binary_oper value_expr
      | postfix_expr '?' value_expr ':' value_expr
      | TYPEID '(' expr ')'
      | SIZEOF TL type_name TG '(' ')'
      | SIZEOF '(' expr ')'
      | cast_oper TL type_name TG '(' value_expr ')'
      | NEW type_id opt_initializer
      | RETURN value_expr
      ;

block
      : '{' stmts '}'
      ;

expr
      : value_expr
      | IF cond block ELSE block
      | GOTO ID
      | LABEL ID ':' expr
      | SWITCH cond expr
      | CASE value_expr ':' expr
      | DEFAULT ':' expr
      | BREAK
      | DO expr WHILE cond
      | WHILE cond expr
      | FOR '(' opt_expr ';' opt_expr ';' opt_expr ')' expr
      | DELETE value_expr
      | DELETE '[' ']' value_expr
      | THROW value_expr
      | THROW
      ;

opt_expr
      : expr
      | /* empty */
      ;

cond
      : '(' expr ')'
      ;

opt_initializer
      : initializer
      | /* empty */
      ;

initializer
      : '(' expr ')'
      ;

cast_oper
      : CAST
      | STATIC_CAST
      | DYNAMIC_CAST
      | REINTERPRET_CAST
      | CONST_CAST
      ;

binary_oper
      : '*'
      | '/'
      | '%'
      | '+'
      | '-'
      | '<'
      | '>'
      | EQ
      | NE
      | LE
      | GE
      | '='
      | LS
      | RS
      | '&'
      | '^'
      | '|'
      | LAND
      | LOR
      | MUL_EQ
      | DIV_EQ
      | MOD_EQ
      | PLUS_EQ
      | MINUS_EQ
      | LS_EQ
      | RS_EQ
      | AND_EQ
      | OR_EQ
      | XOR_EQ
      | COMMA
      ;

literal
      : INT_VALUE
      | STRING_VALUE
      | FLOAT_VALUE
      | CHAR_VALUE
      | BOOL_VALUE
      ;

attributes
      : attribute
      | attributes attribute
      ;

attribute
      : '(' ID ',' literal ')'
      ;

/* BITFIELD */

special_name
      : CTOR
      | CTOR AT INT_VALUE
      | DTOR
      | DTOR AT INT_VALUE
      ;

oper
      : binary_oper
      | UNARY unary_oper
      | INCR
      | DECR
      | '(' ')'
      | '[' ']'
      | DOT
      | DOT_REF
      | ARROW
      | ARROW_REF
      ;
```

VITA

Name:	Peter Mathias Pirkelbauer
Address:	Lawrence Livermore National Laboratory 7000 East Ave. Livermore, CA, 94551
Email Address:	pirkelbauer@acm.org
Education:	Dipl.-Ing. Informatik, Johannes-Kepler Universität, Linz, Austria, 1997 M.B.A., Texas A&M University, 2003