University of Alberta

Library Release Form

Name of Author: Daqing Hou

Title of Thesis: FCL: Automatically Detecting Structural Errors in Framework-Based Development

Degree: Doctor of Philosophy

Year this Degree Granted: 2004

Daqing Hou
9-203, Yang Shu Bei Jie
DaLian, LiaoNing
P.R.China, 116023

Date: _____

The refinement of techniques for the prompt discovery of errors serves as well as any other as a hallmark of what we mean by science.

–J. Robert Oppenheimer

Most papers in Computer Science describe how their authors learned what someone else already knew.

–Peter Landin, Circa 1967

University of Alberta

FCL: Automatically Detecting Structural Errors in Framework-Based Development

by

Daqing Hou

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Spring 2004

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **FCL: Automatically Detecting Structural Errors in Framework-Based Development** submitted by Daqing Hou in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

H. James Hoover
Supervisor

Piotr Rudnicki
Co-Supervisor

Kenny Wong

James Miller

Andrew J. Malton
External Examiner

Date: _____

To Xiao-Li, Niu-Niu, and George

Abstract

Although they are intended to support and encourage reuse, object-oriented application frameworks are difficult to use. The architecture and implementation details of frameworks, because of their size and complexity, are rarely fully understood. Instead, faced with a framework problem, developers must somehow learn just enough about the parts of the framework required for their task and ask for assistance or muddle through using a trial-and-error approach. In many cases, they misuse the framework by not learning what the framework designer had in mind as the proper solution to their problem.

This thesis investigates both the feasibility and the effectiveness of tools support for the problem: The idea is to formalize the patterns to which the code structure of the application should conform, and thereafter detect violations of such patterns with an automated checker program. To capture the know-how knowledge about frameworks use, we introduce the notion of *framework constraints*: framework constraints are rules that frameworks impose on the *code* of framework-based applications.

The tool consists of a specification language and an associated checker. The specification language, FCL (Framework Constraints Language), is defined to formally specify framework constraints. The semantics of FCL is based on a first-order logic extended with set and sequence operations. Essentially, framework constraints can be regarded as framework-specific typing rules conveyed by FCL specifications and thus can be enforced by techniques analogous to those of conventional type checking.

Several case studies have been conducted to evaluate the approach. These include a part of the MFC (Microsoft Foundation Classes) framework, the law of Demeter, Scott Meyers' C++ guidelines, and the Observer design pattern. Lessons in terms of both the strengths and the limitations of FCL are reported.

Acknowledgements

First, I wish to thank my thesis supervisor, Prof. Dr. H. James Hoover, for many things he has done for me during the five years. From the beginning, Jim gave me the freedom to explore my interests. His frowning on the initial formulation of FCL for being ad hoc and lacking formal foundation has been a major incentive for improvement. I have learned from this process of formalization more than anywhere else. I agree with his mature attitude on research, and hope that I can do something of real value in the future. Last but not least, I thank Jim for loaning me *many* useful books (he has a wonderful bookshelf!) and his generous financial support during the end phase of my program.

My co-supervisor, Prof. Dr. Piotr Rudnicki, although joined my committee later, earned my respect for being a real scholar and teacher. It has been a joy to work with Piotr. He always strives to be precise, concise, and simple. I will remember forever how promptly he worked to improve my thesis; he will be my model if, someday, I have my own students.

Prof. Dr. Kenny Wong has been a good person to talk to. He pointed out much related works. I benefit particularly from some of the Cmput 605 (Winter 2003) readings and discussions. It was also he who introduced me the Datrix tool suite.

I benefited also from interacting with many outside persons, notably the mentors from two doctoral symposiums of ICSE 2001 and OOPSLA 2001 respectively.

Prof. Dr. Eleni Stroulia served as a co-supervisor at the early phase of this thesis.

Profs. Drs. James R. Cordy and Andrew J. Malton taught me TXL (Tree Transformation Language) from October 2000 to Spring 2001. Although TXL turned out not being quite suitable for FCL, it opened a relevant area of research which I should know of. I also thank Dr. Malton for his thorough and insightful commentary on this thesis.

Profs. Duane Szafron and Jonathan Schaeffer taught me object-oriented programming and heuristic search respectively. Professor James Miller served as a committee member for both my candidacy exam and oral defense. English Professor Janice Williamson proofread a draft of this thesis and gave sound suggestions on English writing; of course, all remaining errors are mine. I thank them all.

Bell Canada provided a research-free license for their Datrix tool suite. I also acknowledge the financial support from the University of Alberta, NSERC, and ASERC (Alberta Software Engineering Research Consortium).

For the record, a discussion in February 2000 with the then fellow graduate student, Garry Froehlich, inspired the idea of this thesis. Thank you, Garry!

Finally, I thank my family for allowing me to immerse in my thesis work. It is *really* a luxury to be able to do so.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Statement of Research Problem

An object-oriented framework is a reusable design for all or part of a system that is represented by a set of abstract classes and the way their instances interact [JF88]. From the perspective of application programmers, it is a skeleton application that can be customized to produce specific applications in a certain domain. Framework builders provide mechanisms, the variation points, to enable this [Deu89].

While a deep understanding of general framework-based development [Joh97, BMMB98] remains a research problem, many frameworks are being used for production development. Having chosen a framework, how does the development team address the problem of correct usage of the chosen framework? The essence of software determines that solutions are likely to be multi-faceted and complementary to each other.

Experience with using industrial strength frameworks has shown that in order for frameworks users to understand and properly use a framework, precise, concise, and complete documentation is required [FHLS97, Fro02]. However, textual and diagrammatic documents are informal, and in general, we do not know yet how to test whether a programmer has understood a document.

Other conventional approaches such as framework design review, manual code inspection, and testing can also be helpful. But they are not without problems. Frameworks are supposed to capture commonality in a way that makes reuse easier. But applying most current frameworks requires a nontrivial body of knowledge about the frameworks on the part of users. Lack of understanding makes debugging difficult because it can be hard to follow a thread of execution that is mostly buried in the framework code. Testing is similarly difficult since it often requires a fundamental understanding of the architecture of the framework.

The size and complexity of frameworks and their notorious lack of design and intended-usage documentation make framework-based development a learning-intensive and error-

prone process [HHY03]. Commonly framework users misunderstand the relationships between their application and the framework and are unaware of how the framework designer intends the framework to be used, resulting in overly complex solutions or subtle bugs.

This work was inspired by our initial observation that many errors of using frameworks are due to wrong structures or misunderstanding of structures. This observation has been backed up by two studies on user groups of MFC (Microsoft Foundation Classes) and Java/Swing: In the Java/Swing case, we collected and analyzed about 200 news group questions related to the JTree component and confirmed that structural errors indeed represent an important class of errors [HHY03]; our MFC study supports the observation in a more rigorous manner because the evidences are documented by an expert user who has been using MFC and answering news group questions for several years [New]. We analyzed only a subset of his essays and found many of them are structural errors (see Chapter 6). In addition, our experience of interacting with active users of other frameworks also supports this observation [ML].

For the framework user with shallow knowledge, something more akin to type-checking is desirable. That is, framework developer takes on the burden of describing/specifying how to properly use the framework so that compliance by the framework user can be checked mechanically. Although correct type matching is no guarantee that a function is used properly, it does catch many common mistakes. We would like something similar to apply to framework use.

We use the term *framework constraints* to denote the knowledge that a user needs to know in order to use a framework properly. The idea is to formalize the framework constraints on hot spots and check whether a framework instantiation satisfies these constraints. Our goals are twofold: to create specification languages and tools so that framework builders can specify the intended use of their frameworks and framework users can then use the tools to check their applications.

In the long run, we want to look into the feasibility of two technologies, namely, static analysis and model checking, to the problem [HHS02]. Along that line, framework constraints can be categorized into *structural constraints* and *behavioral constraints*. Structural constraints can be evaluated by parsing and analyzing source code while behavioral constraints could be dealt with by model checking.

This thesis focuses on the structural aspect of framework constraints. A specification language, FCL (Framework Constraint Language), is designed to express the constraints on the structure of source code. The feasibility and effectiveness of FCL are demonstrated through several case studies. We conclude that it is indeed useful to framework users.

1.2 FCL

FCL is a strongly typed first-order logic extended with set and sequence operations. Its term language consists of a special set of total functions, reflecting the entities and relationships of programming languages.

FCL's formalism lends to its automation. Specifiers specify a set of constraints on program structure, and conformance of a specific program to the constraints is then automatically determined by an FCL checker.

Since FCL is typed, run-time errors can be ruled out, but it is up to the specifiers to ensure the consistency of FCL specifications. With FCL, it is the specifiers who carry out the reasoning process and determine to what constraints the code structure should conform. FCL specifications are only the "conclusion" of that process.

The need for FCL stems from the absence of constraint languages rich enough to express the kinds of structural constraints we explored and the possibility to do inference with FCL specifications in formal systems such as the Mizar environment [Org] in the future.

Several case studies have been done to gain knowledge on how FCL can be better designed and used, and whether it can be helpful in practice:

- The Observer Pattern
- MFC's dialog architecture
- Swing's JTree component

In addition to frameworks, FCL has also been used to encode general design rules:

- Law of Demeter
- Scott Meyers' C++ guidelines

False positives and false negatives are always a big concern with any error detection tool. When specifying constraints, one often inclines to add as many constraints as possible, hoping to pick up more errors. But care must be taken not to add too many; otherwise spurious errors might be generated. Ideally, one would expect to have just enough constraints, no more and no less, so that FCL can help pick up all errors but the spurious ones. We do not have any hard principles on how this can be achieved yet, but our experience suggests that the following guidelines tend to be useful:

- In general, the more specific the context is, the more effective FCL tends to be. Many examples show that the components which FCL is used to constrain are highly specialized; they often assume a great deal of context. FCL can be more effective under such circumstances.

- Specifiers should know the design well and avoid immature generalization. However, this often implies that the specifiers should be the designers or somebody who works closely with them.
- Specifying against specific symptoms sometimes can be economical.

The errors FCL detects include both errors of omission and errors of commission. Many errors are design errors caused by misunderstanding and/or by being unaware of properties and interaction at system interfaces. The errors are both system- and domain-specific. They are different from generic implementation errors such as dereferencing null pointers or array bound overflow; independent of the domains of the inspected programs, such implementation errors can occur in any programs.

Our experience so far shows that FCL specifications tend to be short. This is encouraging since practitioners would be more likely to pick up and use it.

The current version of FCL is targeted at C++ for several reasons. C++ has many features that can potentially complicate program analyses: separate compilation, inheritance, overloading, templates, exception handling, and so on. Thus choosing C++ as the target is more likely to expose potential problems with the approach. A real C++ framework, MFC (Microsoft Foundation Classes), is publicly available [SW96]; furthermore, extensive user experience with MFC is also available in the forms of both news group discussion and essays by expert users [New]. We are relatively familiar with MFC. But the idea should be ready to extend to other strongly-typed programming languages such as Java.

The implementation of FCL consists about 20,000 lines of C++ code, and to date about 1,200 lines of FCL specifications have been written.

The FCL approach is different from program verification; there, the specifications are mainly concerned with program behavior. Moreover, conventional methods of specification and verification emphasize abstractions; typically, abstraction functions are used to tie a piece of specification with a specific implementation [Hoa72]. The idea is that by focusing on the abstract specification and ensuring its correctness, one can not only reuse it in many different implementations, but also use it as a surrogate when reasoning about other implementations that use it.

Verification in theory can guarantee the correctness of a program with respect to a specification. But automatic theorem proving is intractable, which burdens the users to provide both lemmas and proof strategies. Typically, the process is labor-intensive and tedious. In contrast, FCL is fully automated and FCL specifications tend to be short and thus easier to write. Its drawback is that even if a program passes all the constraints in an FCL specification, FCL cannot guarantee its full correctness. FCL is partial in its expressive power, modeling ability (only on syntactic structure), degree of analysis, and degree of compositionality. Thus FCL belongs to the family of "lightweight" formal methods [JW96].

Although FCL has no constructs for specifying abstractions, its focus on syntactic structure seems to be desirable in practice, to quote the authors of the Larch book [GH93]:

> Specifications should not just describe mathematical abstractions, but real interfaces supplied by programs. They should be written at the level of abstraction at which clients program. This usually means sinking to the level of a programming language.

In a sense, FCL is a technique that complements conventional formal specifications.

The relation between FCL and model checking can be characterized as follows: The program being checked would be the model, while the FCL specification would be the properties that the model is supposed to uphold. The difference lies in the fact that technically, tools for model checking often assume the responsibility of constructing all the possible models and check them *exhaustively*, whereas in FCL, it is the programmers who provide programs to the tool one by one.

Technically, FCL is more akin to type-checking. The difference can be best illustrated by an example: while the empty program would pass all the type checkers, it can hardly pass any non-trivial FCL specifications.

FCL complements testing. Testing requires executability and execution implies completeness, in the sense of both the completeness of the application and the coverage of the test cases. Testing is also intrusive. In contrast, a completed application is not necessary in order for FCL to work; FCL can check partial implementations. Thus FCL fits well with incremental development.

1.3 Summary of Contributions

This thesis is motivated by the desire to have tools support for detecting errors in framework-based development. More specifically, it proposes to extend the technique of type checking and apply it to framework-based development. A summary of the main result and contributions are as follows:

1. The design and implementation of FCL make the bulk of the work. A model of the static properties of C++ programs is presented, and FCL is defined based on that model. A formal semantics is also defined for FCL.

2. The feasibility and the potential usefulness of the approach are then demonstrated by applying FCL to real frameworks. Specific lessons learned from the experience are reported. These lessons are important for both the use and future development of tools like FCL.

To use a framework is to learn its design; many problems originate from a lack of understanding about its architecture, design pitfalls, and evolution. FCL has been used to perform the following kinds of tasks:

- Detecting omission of programming obligations;
- Detecting violations of programming constraints implied by a design;
- Detecting violations of programming constraints implied by a "negative design;"
- Enforcing programming disciplines;
- Helping with the co-evolution of both the framework and its intended use.

1.4 Outline of the Rest of the Thesis

The rest of this thesis is organized as follows:

- Chapter 2 provides the necessary background for understanding framework-based development. It traces the origin of the related concepts such as program families, frameworks, and software product lines. The state of the art of frameworks is then surveyed. Finally, we show the current status of the research area, presenting one recent result in better modularization support for reusing large scale components, the *mixin layers*.

- Chapters 3 and 4 describe the design and implementation of FCL. Specifically, chapter 4 presents a model of the static properties of C++ in the object-oriented notation; chapter 3 describes how a first-order logic, FCL, is defined and implemented on top of that model. Appendix A presents a formal treatment of the static and dynamic semantics of FCL.

- Chapters 5, 6, and 7 present three case studies on the application of FCL. Chapter 5 is purported to demonstrate the main features of FCL. It was done to explore the design space of FCL. One lesson reported in the chapter is that FCL depends on the specificity of the contexts: The more specific they are, the more effective FCL can be. Chapter 6 describes how FCL is applied to part of the MFC framework and analyzes the kinds of situations where FCL may help. Chapter 7 describes how FCL is applied to general design principles and programming guidelines with the law of Demeter and a subset of Meyers' C++ guidelines as examples.

- Chapter 8 positions FCL in related works. Program analyses, error detection tools, formal specification languages, and work specifically for frameworks are surveyed and analyzed. Closely related works such as CCEL and CoffeeStrainer are compared with FCL.

- Finally, chapter 9 summarizes the main contributions of the thesis and outlines some future work.

Chapter 2

Background

The essence of software construction is the formulation of complex conceptual constructs that are abstract in the sense that they remain the same under different representations. The four inherent properties of this essence are complexity, conformity, changeability, and invisibility [Bro87].

Software does not grow by a mere duplication of the same elements; instead, a scaling-up of a software is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly. This comprises the complexity of software.

Many technical and managerial problems stem from this complexity including: the difficulty of enumerating and understanding all the possible states of the program and the difficulty of communication among team members, the cause of product flaws, cost overruns, and schedule delays. This makes an overview challenging and thus impedes conceptual integrity. The subsequent learning and understanding burden makes personnel turnover a disaster.

Managing complexity therefore has been a central topic in the pioneering work of our field [Dij70, Par01b, Dij68, HFC76]. In particular, these works all center on the hierarchical structures of systems for a number of reasons, among which its linear structure is probably the most attractive one for its lower complexity and thus better intellectual manageability.

The rest of this chapter traces the common origin of such seemingly different concepts as frameworks, program families, and software product lines, demonstrating that they are indeed all derived from early research on programming methodologies whose main concern is how to construct correct and reliable programs systematically.

2.1 Stepwise Refinement and Program Families

With stepwise refinement [Dij70], one composes programs in minute steps, deciding each time as little as possible. Each of the steps contains such design decisions as choosing a

particular data structure or creating a loop statement. As one proceeds with the problem analysis, so does the further refinement of their program. At any given moment, the program can be viewed as being derived from successive *levels*, with the later levels containing refinements to entities in the earlier ones. When a program has to be made, the desired computation has to be composed from actions corresponding to a *well-understood repertoire* of instructions.

Since the steps are small, their correctness, with respect to the entities they refine, should subject to rigorous proofs immediately. This way, one essentially obtains a correct program by construction. This is the so-called *constructive approach to program correctness*, examples of which can be found in, for instance, [Dij70].

One important insight inspired by stepwise refinement is the notion of *program families*. During stepwise program composition, the following may happen: (1) often one encounters situations where more than one refinement exists for an entity of the higher level, and (2) sometimes, two entities at the same level can be refined independently. Therefore, many programs rather than the only one for the task at hand can be conceived using these two kinds of variations. These programs either do the same task in different ways or carry out similar tasks. Thus it makes sense to create and study a *program family* and to consider the programs as members of the family.

Another motivation for program families comes from the evolution of large software systems. Large software are often changed for two reasons: the software is not satisfactory in one way or another and thus needs to be changed; and/or the software is suitable for the task at hand, but people want to adapt it to similar tasks. In either case, the programmer would have to "change some design decisions." On the other hand, at any instant of stepwise refinement, what has been built up is an intermediate program that is a suitable "common ancestor" for all the possible programs produced by further refinements. In this process, one defers design decisions. It is the potential similarity between "the decision to be changed" and "the decision still left open" that inspires the concept of "program families."

It pays off for one to focus on creating a program family instead of individual programs. The benefit of regarding programs as family members derived from a common ancestor can be summarized as follows:

- Members of the family share the correctness proof as far as possible. Regardless what decisions are made at the current level, the coding of the earlier levels remains valid.
- Members share as far as possible the common coding.
- The regions that can be adapted or modified are well localized.

The paper [Par01d] further develops the concept of *program families* in the context of information hiding modules [Par01c]. It considers "a set of programs to constitute a *family*, whenever it is worthwhile to study programs from the set by *first* studying the common

properties of the set and *then* determining the special properties of the individual family members." Although the discussion on program families and the method of developing them in [Dij70] is general and not necessarily tied to any specific computing mechanism, it is demonstrated primarily through algorithmic programs at the level of control statements. Thus the contributions of [Par01d] are to introduce the concept of modules and demonstrate how module specifications work in designing program families. The conclusion is that these two methods complement each other.

The paper [Par01a] studies the design of software both extensible and easily contracted. The most critical step is the design of a software structure called the "use" relation. Having defined the "use" structure, one can then identify subsets of modules that can be used independently of their dependents. The resulting practical benefit means that one can not only reuse a subset in different contexts, but also "failsafe" in case of schedule slippage by delivering only the subset. The identification of *minimal* subsets and *minimal* extensions can lead to such software.

Operating systems have been a good subject for studying design in general [Dij68] and program families in particular [HFC76, C+93]. To describe systems structure, an important concept, variously named as "layered abstraction," "levels of abstraction," or "hierarchical structure," was used in these studies. One well-known observation is that, even in strictly layered domains like operating systems, the notion of "information module" [Par01c] does not necessarily coincide with the notion of "layers of abstraction": modules may encompass different parts of several layers [HFC76].

A note has to be made here about the distinction between the concept of modularization and language constructs such as macros, procedures, and classes. Parnas' early work clearly treated modularization as a design issue, not a language issue. A module was a work assignment, not a sub-routine or any other language element, although language supports could make the job easier [D+03].

2.2 Software Product Lines and Object-Oriented Frameworks

While the idea of program families is old, it has become a popular topic only recently. The latest incarnation of the notion is software product lines [WL99]. A *product-line architecture (PLA)* [BCS00] is a design for a family of related applications. The motivation for PLAs is to simplify the design and maintenance of program families and to address the needs of highly customizable applications in an economical manner.

A *framework* is an abstract design for a family of related problems within a certain domain; the abstract design consists of a set of abstract classes, each of which defines the interface for a major component of the applications [Deu89, JF88]. Certain methods of

these classes are left unspecified (and hence are "abstract") because their implementation is application-specific. Thus a framework is a "code template": key methods and other details still need to be supplied, but all common code is present in abstract classes. A *framework instance* provides the missing details. The pairing of a concrete class with each abstract class of the framework provides a complete implementation.

Object-oriented application frameworks are appropriate for reusing software parts and specializing them in multiple ways for distinct applications, and thus are an enabling technology for producing members of program families. This is due to three features supported by object oriented languages: data abstraction, polymorphism, and inheritance. Data abstractions define interfaces behind which implementation can change. Polymorphism increases the likelihood that a given component will be usable in new contexts. Inheritance promotes the emergence of standard protocols and abstract classes, and allows existing components to be customized. Therefore, frameworks are a simple and general technique-from an implementation standpoint, they are just a coordinated use of inheritance. Since data abstraction, polymorphism, and inheritance are fundamental mechanisms of object-oriented languages, the applicability of the framework approach is wide.

Through the means of customization, frameworks can be categorized into whitebox and blackbox [JF88]. For whitebox frameworks, one mainly relies on inheritance and subclassing. Thus one would have to know the implementation details of the frameworks in order to use them correctly. For blackbox frameworks, one primarily relies on composition of existing components; to customize the frameworks, one only has to know the interfaces of the components. One the surface, it seems that blackbox frameworks are easier to use than whitebox ones; thus they are claimed to be the ideal towards which a framework should evolve.

Frameworks are a kind of reuse technique [Kru92] different from both components and design patterns [Joh97]. Components are code reuse, while design patterns represent design reuse; design reuse reuses *concepts* while code reuse reuses *implementation*. Frameworks are both code and design reuse. One of the problems in reusing design knowledge is the lack of standard notations to capture and express them [BR89]. In frameworks, no special notations represent designs: object-oriented programming language are used as both implementation and design notations. While both design reuse and code reuse are important, in the long run it is probably the design and understanding of the domains that provide the biggest payoff [BR89].

Many frameworks have been built for such diverse areas as graphical user interfaces [Deu89, Inc89, SW96], operating systems [C+93], drawing editors [VL89], distributed software and manufacturing control [FS97], to mention a few. Frameworks that can be reused across many domains are *foundation frameworks*, and those only applicable to a certain domain are

domain or application frameworks. Most domain frameworks are kept proprietary [Joh97].

Frameworks are harder to build than plain applications. Building a framework requires experience, and the process is almost always iterative and incremental [JF88]. The reasons are as follows [Joh97]:

- Domain analysis is needed to understand a domain. Mistakes in domain analysis are discovered when a system is built, which leads to iteration.
- A framework makes explicit the parts of a design likely to change. In general, the only way to learn what change is by experience.
- Frameworks are abstractions, and abstractions are expensive to discover and verify.

Framework developers should always start with concrete examples and generalize from them [Joh97]. For large and complex domains, the number of examples can be huge. This proliferation of examples also explains why frameworks are both hard to build and difficult to build on schedule. They should be built by advanced development or research groups, with close collaboration of application teams.

A common mistake is to start using a framework while its design is still changing. On the other hand, the only way to find out what is wrong with a framework is to use it [Joh97]. However, a framework should only be released to public use when it is stable enough.

2.3 Scaling Stepwise Refinement

It is well-known that in object-oriented design, objects are encapsulated but rarely self-sufficient entities. The semantics of an object is often defined by its relationship with others. Object interdependencies can be expressed as collaborations. A *collaboration* is a set of objects and a protocol that determines how the objects interact [BCS00]. The part of an object that enforces the protocol of a collaboration is called the object's *role* in that collaboration [BCS00].

Two problems with object-oriented frameworks are identified [BCS00, SB02]. One is that frameworks "grow" in a "top down" fashion: subclasses are coded in terms of superclasses; they cannot exist independently. It is desirable, however, for subclasses to exist independently of superclasses; for instance, if the strategies for graph traversal could exist independently of the graphs, regardless whether the graphs are directed or not, then given a graph, it would be possible to compose a traversal collaboration with it, instead of reimplementing the traversal for the graph. The other problem has to do with "optional features." Given an optional feature, a framework would have to either include it in the code base, which would be inappropriate for those applications that do not need it, or exclude it. In the latter case, if the implementation of the feature is nontrivial, then it would be inappropriate for those who need the feature since they would have to reimplement it each time it

is needed. One way to get out of this dilemma is to provide a *simple* way so that users can specify that a certain feature is needed; accordingly, code for the feature is composed into the final program.

Historically, software design and programming languages intimately evolve around the concept of modularity. The language solution to this problem is a new construct that allows one to represent collaborations as modules parameterized by their dependees and supports the composition of such collaborations.

In collaboration based design, however, each collaboration usually involves more than one *program fragment*; to refine a collaboration, one typically would have to change more than one program fragment. Mixin layers extend the notion of *mixins* and are a programming technique that allows one to program all the changes in one place.

In object-oriented languages, a superclass can be defined independently of any subclass. This property, however, is not symmetric; it does not hold for subclasses. *Mixin classes* are *abstract subclasses*. Mixins represent a mechanism for specifying classes that eventually inherit from a superclass, but the superclass is not specified at the site of the mixin's definition. Thus a single mixin can be instantiated with different superclasses, yielding widely varying classes. Note that this is different from C++'s concept of mixin classes. In C++, mixin classes are classes that share a common virtual base class, each of which implements part of the interface of the base class [Str99].

One can implement mixins through C++ template classes:

```
template<class Super>
class Mixin: public Super {
... // mixin body
};
```

Mixin layers can then be implemented with parameterized templates and nested classes in languages such as Java and C++. In C++, a mixin layer is a mixin class that may define multiple nested classes. The mixin layer itself is called the *outer mixins*, while the nested classes are called the *inner mixins*. An outer mixin is implemented as a template class whose type parameter is the layer that it depends on. Some nested classes within an outer mixin may be newly defined while others refine the corresponding nested classes of the layer on which the current layer depends.

```
template<class SuperLayer>
class MixinLayer: public SuperLayer {
        class role1: SuperLayer::role1
        {
         // role1 body
        }
        class role2: SuperLayer::role2
        {
         // role2 body
        }
        ...
```

```
        class newClass
        {
         // newClass body
        }
};
```

Collaboration-based designs in general and mixin layers in particular are not universally applicable but more suitable to mature domains that are well-understood and amenable to detailed decompositions. The domains should be decomposable into largely independent refinements. Clearly, this is different from frameworks, which pose no such requirements on the domains. In fact, some future extensions to frameworks can be completely open at the time the frameworks are constructed. Therefore, collaboration-based design and frameworks are two complementary techniques.

Examples for the applications of mixin layers can be found in [SB02]. A case study that applies both mixin layers and a domain specific language for finite state machines to a software product line appears in [B$^+$02].

Chapter 3

Design and Implementation of FCL

3.1 An Example

Let us introduce FCL with a simple example with two files F.h and F.cpp:

```
// F.h
class F {
         ...
         virtual void m();
         ...
};

// F.cpp
#include "F.h"
         ...
void F::m(){
...
}
         ...
};
```

Note that class F defines a virtual member function m. Assume class F belongs to a framework and a constraint on F is that if a subclass overrides m, then the override must call F::m.

The following code shows how a user might break this constraint; they forget to call the base class version from the subclass:

```
// A.h
#include "F.h"
class A: public F {
         ...
         void m();
         ...
};

// A.cpp
#include "A.h"
void A::m(){
... // does not call F::m
```

}
...

FCL is meant to detect violations of such framework constraints. This constraint may be specified using FCL as follows:

```
1 forall c:  subclass(class("F")) holds
2 [m as {f:  function(c) | sizeof(param(f)) = 0 and name(f)="m" };]
3 (sizeof(m)=0 or
4 exist e:  exp(m) holds
5   function(e) = function("m", class("F"))
6 or print(c) and println(" breaks the constraint!"))
```

In this specification, subclass, class, function, param, and exp are primitive functions on the syntactic structure of C++ source code (see Table 3.2 for their semantics); sizeof is a standard set operation returning the size of a set; print and println are printing functions and always return true. These will be explained in more detail in Section 3.2.2.

3.1.1 Tools Operation

Four steps are involved in checking a program:

1. Preprocess each *.cpp* file with an appropriate preprocessing tool. On Unix, use the *cpp* program; for Microsoft C++, use *cl /P*, where *cl* is the name of the compiler.
2. For each preprocessed file *f.i*, run *dxparscpp f.i –asgf.asg*, where *f.asg* is the file containing the generated Datrix ASG. *dxparscpp* is a parser from Bell Canada.
3. Run *dxlinker* on the set of ASG files, $asg_1, ..., asg_n$, with the following format:
   dxlinker $asg_1, \ldots, asg_n >$ *final.link*,
   where *final.link* is the file that stores the full ASG. *dxlinker* performs type analysis on the outputs of *dxparscpp* to form a single ASG.
4. Given a file *f.fcl* containing the FCL specification, we can check whether the above program conforms to it by issuing the command:
   fcl f.fcl -d final.link.

Although still rudimentary, the output of the *fcl* program is informative enough for debugging purpose. For example, running *fcl* on a small program that forgot to call the member function m of class F yields the following message:

```
Parse FCL file base.fcl...
Check FCL file base.fcl...

$INSTANCE 16 cAggrType
16{
  beg = 10.1
  end = 17.6
  name = "A"
  type = class
}
breaks the constraint.
```

This message corresponds to line 6 of the specification, showing what the output of the printing facility looks like. Note that the current implementation outputs the corresponding ASG node of a program entity; in this example, the ASG node for class A is shown, which does not call the base class version of member function m.

This implementation is only a prototype; depending on the nature of the environment to which FCL is attached, in the future the output can be more interesting, for example, a highlight of the code region that violates the constraint.

3.2 Introduction to FCL

FCL is a little language for talking about the structure of object-oriented programs. Its notations are borrowed from first-order logic extended with set and sequence operations; its term language consists of a set of total functions reflecting the entity-relationships in the domain of object-oriented programs.

Abstractly, the syntactic structure of an object-oriented program forms a graph whose nodes represent syntactic elements such as namespaces, classes, functions, variables, and expressions and whose edges represent the relationships between these elements such as an expression and the function that it is statically bound to and a variable and its type. In this chapter we will appeal to readers' intuitive understanding of such a data model for C++ source code; a detailed account for our C++ source code model appears in Chapter 4.

In the rest of this section we introduce FCL based on the syntax of Table 3.1.

At the topmost level, an FCL specification consists of a sequence of interleaved declarations and formulas. Declarations do not have to be defined all at once before formulas; they can be freely interspersed among formulas as long as variables are defined before they are used. The combination of a top-level formula and all the declarations that it refers to forms *an FCL constraint*.

Each declaration binds a variable to an associated expression, and the variable takes the value of the expression. FCL variables are different from programming language variables because their values do not change over time. In particular, logical formulas are treated as a special kind of expressions; they are expressions that yield values of the boolean type. Therefore, FCL allows one to define boolean variables with formulas as their value expressions.

FCL allows one to introduce local variables for expressions through a syntactic structure called *block*; blocks are a simple grouping mechanism, and each expression is allowed to have at most one block. One variable overrides another variable if the former has the same name as the latter, appears after the latter, and is defined either in the same scope or in any enclosed scope of the latter.

Local variables are most useful when writing formulas; the variables assigned for expres-

FCL_spec	::=	Statement_list
Statement	::=	Declaration | Formula
Declaration	::=	Variable as Expr
Formula	::=	not Formula| Formula and Formula| Formula or Formula| Existential | Universal | Expr
Expr	::=	Variable | Constant | Operation | Formula | ExprWithVars
Existential	::=	exist BVar_Declaration_list hold Formula
Universal	::=	forall BVar_Declaration_list hold Formula
BVar_Declaration	::=	Declaration
ExprWithVars	::=	'['Declaration_list']' Expr
Operation	::=	Set_op | Seq_op | Relational | FCL_fct
Set_op	::=	subset(s1, s2) | belongsTo(ele,s) | s1 + s2 | s1 - s2 | union(setOfsets) | sizeof(s) | Set_comprehension | Set_enumeration
Set_comprehension	::=	{ BVar_Declaration_list | Formula | Expr} | { BVar_Declaration | Formula}
Set_enumeration	::=	'[' Expr_list ']'
Seq_op	::=	member(seq, index) | indexOf(ele, seq)
Relational	::=	> | >= | < | <= | =
Constant	::=	true | false | Str | Int | global
Variable	::=	Str
T_list	::=	T*

Table 3.1: The Syntax of FCL

sions, if properly named, can help reveal intent; it also helps structure the specification by avoiding long or repeated expressions.

Formulas have conventional semantics; the syntax replaces Greek symbols with English words. Formulas include negation, conjunction, disjunction, and universal and existential quantifications.

Elementary formulas include the boolean constants *true* and *false*, relational operations, and such predefined predicates as the subset relation and the membership relation (stating that an element belongs to a set). Syntactically, these predicates are represented as function applications (that is, in the form of $f(e_1, \ldots, e_n)$).

In both universal and existential quantifications one is allowed to define more than one bound variable at once, binding their values to the elements of the set-valued expressions. For the sequence of bound variables, those that appear later can both refer to and override the ones that appear before them.

As mentioned, FCL treats formulas as a special kind of expression. Other kinds of expressions include variable references, literal constants, function applications, and sets.

Variable references and literals are elementary expressions; they can be used to construct more complex expressions. In addition to the usual kinds of literals, *true* and *false*, integral constants, and strings, FCL has a special literal, *global*, which represents the global namespace of a given program.

Function applications of the form $f(e_1, \ldots, e_n)$ are probably the most frequently-used expressions. FCL predefines a variety of functions. Some of them are standard set and sequence operations. Others are functions defined on the data model for source code. For instance, given a variable c of type Cls, function application *var(c)* returns the set of data members defined in the class represented by c. These functions are specified in detail in section 3.2.2.

One can get sets in three ways: through function applications (some functions return sets as their result), set comprehension, and through set enumeration.

FCL supports two forms of set comprehension:

$$\{\ e : s \mid f(e)\ \}$$

and

$$\{\ e_i : s_i, 1 \leq i \leq n \mid f(e_1, \ldots, e_n) \mid E(e_1, \ldots, e_n)\ \}$$

The first form defines a subset of the set s, of those elements e that make the formula f *true*. But it only permits one bound variable e. The second form allows more than one bound variable, $e_1, \ldots, e_n$, over multiple sets $s_1, \ldots, s_n$. If a tuple $e_1, \ldots, e_n$ satisfies f, the function E is then applied to the tuple and the value $E(e_1, \ldots, e_n)$ is taken as an element of the new set. The second form is more general than the first one in that the first is a special case of it, which can be represented as follows: $\{\ e : s \mid f(e) \mid I(e)\ \}$, where I denotes an identity function.

Unlike for sets, FCL does not provide constructor functions to create new sequences from scratch; instead, sequences are return values from function applications. For instance, an execution path consists of an array of expressions; a function can have a sequence of parameters; and a path on an inheritance hierarchy contains all the classes from a source class to a target class. To model these, FCL needs sequences. A sequence can also be used as a set; the range of the sequence will be used.

3.2.1 FCL's Type System

FCL is strongly typed so that when evaluating an FCL specification, an FCL checker will not suffer run-time errors. This is ensured by putting a set of constraints on the structure of FCL specifications. The constraints comprise the static semantics of FCL.

The type system of FCL includes two kinds of types: basic types (Figure 3.1) and compound types. Basic types can be further divided into "facility" types and "domain" types. "Facility" types help form constraints, including Str for string values, Int for integers, and Bool for boolean values. "Domain" types come from the problem domain of programming constructs, including Exp for expressions, Var for variables, NS for namespaces, Fct for functions, Cls for classes, Name for named entities, Unit for program units, Gen for types

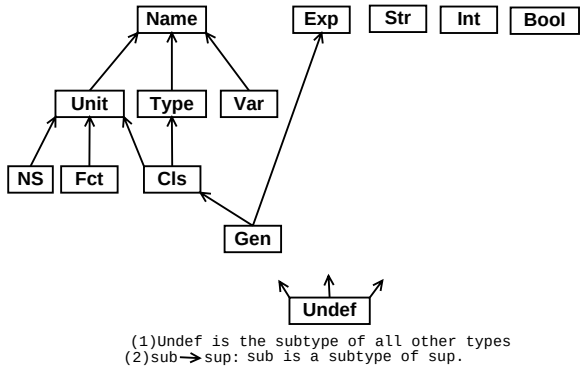

Figure 3.1: Basic types and the subtype relation

generated through template instantiations, and Type for types. The current FCL covers only a subset of the object model; for instance, pointer and reference types, arrays, builtin types, and generated functions are not included.

Undef is the type for "undefined" values. Undef is considered as the subtype of any types.

Figure 3.1 also defines the *subtype* relation between basic types. Besides basic types, subtype relations can also exist between compound types. One set type is the subtype of another if and only if the base type of the former is the subtype of the latter. Similar definition holds for sequence types.

3.2.2 Functions on Source Code Model

Table 3.2 presents the signatures of all FCL functions. The following is a brief explanation for each of them:

- class, var, function:

  These constructors allow one to refer to a known entity of the checked program. Their arguments specifies the name of the entity and its context. For instance, if one wants to refer to a method named m within a class C, one can write it as function("m", class("C", global)). Since global can be omitted, it can also be written as function("m", class("C")).
- isPrivate, isProtected, isPublic:

  Test visibility. Can only be applied to variables and functions.
- isStatic, isConst:

  Test staticness and constness. Can only be applied to variables and functions.
- isBranch:

  Test whether an expression is conditional.
- isReturn:

  Test whether an expression is the argument of a return statement.

Constructors	class: **Str**×**Unit**→**Cls**
	var: **Str**×**Unit**→**Var**
	function: **Str**×**Unit** → **F Fct**
Property Predicates	isPrivate/isProtected/isPublic: **Var**|**Fct** → **Bool**
	isStatic/isConst: **Var**—**Fct** → **Bool**
	isBranch/isReturn: **Exp** → **Bool**
	isVirtual: **Fct** → **Bool**
	name: **Name** → **Str**
Type Query and Type Coersion	isClass/isGenerated/isVar: **Name** → **Bool**
	class: **Type** → **Cls**
	Generated: **Type** → **Gen**
	ptdType: **Type** → **Type**
	ptdTypeStar: **Type** → **Type**
	var: **Name** → **Var**
Unit Operations	unit: **Name** | **Expr** → **Seq Unit**
	class: **Unit** → **F Cls**
	exp: **Unit** → **F Exp**
	var: **Unit** → **F Var**
	function: **NS** | **Cls** → **F Fct**
	expClosure: **Fct** → **F Exp**
	param: **Fct** → **Seq Var**
Expressions	receiver: **Exp** → **Exp**
	arg: **Exp** → **Seq Exp**
	parent: **Exp** → **Exp**
	top: **Exp** → **Exp**
	exp: **Exp** → **F Exp**
	upPath/downPath: **Exp** → **F Seq Exp**
	function: **Exp** → **Fct**
	refd: **Exp** → **Name**
	refSet: **Name** → **F Exp**
	change: **Exp** × **Var** → **Bool**
Inheritance	subclass: **Cls** → **F Cls**
	superclass: **Cls** → **F Cls**
	descendant: **Cls** → **F Cls**
Type	type: **Exp** → **Type**
	type: **Var** → **Type**
	type: **Fct** → **Type**
Misc	isDefined: **Any** → **Bool**
	print: **Any** → **Bool**
	println: **Any** → **Bool**
	regex: **Str** × **Str** → **Bool**
	concat: **Str** × **Str** → **Str**

Table 3.2: FCL functions: **F T**: finite subsets of type **T**; **Seq T**: sequences of type **T**.

- isVirtual:

  Test whether a function is virtual.

- name:

  Returns the name of a named entity.

- isClass, isGenerated, isVar:

  Test whether a named entity is a class, a generated type, and a variable, respectively.

- class:

  Downcast the argument from Type to Cls. If the argument is not a class, then the result is the "undefined" value.

- Generated:

  Downcast the argument from Type to Gen. If the argument is not a generated type, then the result is the "undefined" value.

- ptdType:

  If the argument is a pointer type, then returns the type pointed. Otherwise, return the type itself.

- ptdTypeStar:

  If the argument is a pointer type, then returns the base type. Otherwise, return the type itself.

- var:

  Downcast the argument from Name to Var. If the argument is not a variable, then the result is the "undefined" value.

- unit:

  Returns the context of the argument, a sequence of Unit. Can be applied to both Name and Expr.

- class:

  Returns the set of classes defined within a Unit.

- exp:

  Returns the set of expressions defined within a Unit.

- var:

  Returns the set of variables defined within a Unit.

- function:

  Returns the set of functions defined within either a NS or a Cls.

- expClosure:

  Returns the set of expressions that can be statically reached from the argument Fct.

- param:

  Returns the sequence of parameters of an argument of type Fct.

- receiver:

Returns the receiver expression of an argument expression. If the expression has no receiver expression, e.g., an arithmetic one, then "undefined" is returned.

- arg:

  Returns the sequence of argument expressions of a given expression.

- parent:

  Returns the parent expression of a given expression. If the expression has no parent, then "undefined" is returned.

- top:

  Returns the root expression of a given expression.

- exp:

  Returns the set of all sub-expressions of a given one, including itself.

- upPath, downPath:

  If the argument is an expression within a function, then returns the sets of its up paths and down paths respectively. Otherwise, return the empty set.

- function:

  Returns the function that a given expression invokes.

- refd:

  If the argument is a name reference expression, returns the named entity referred. Otherwise, returns "undefined."

- refSet:

  Returns the set of name references that refer to the given named entity.

- change: change(exp, aVar) tells whether a given expression may change the variable represented by the parameter aVar.

  If aVar is not referenced by the expression, change returns false. Otherwise, for each of the expressions where aVar is referenced:

  - If the expression is the pre-defined assignment and aVar is at its left-hand side, then change returns true.
  - If the expression is a function call and aVar is the receiver: If the function invoked is not const, then change returns true; otherwise, change returns false.
  - If the expression is a function call and aVar is an argument to it: If the corresponding parameter of the invoked function is not const but of reference type, then change returns true; otherwise, change returns false.

- subclass:

  Returns the set of classes that are the subclass of the given class.

- superclass:

  Returns the set of classes that are the super class of the given class.

- descendant:

Returns the set of classes that are the descendant classes of the given class.

- type:

  Returns the type of a given expression.

- type:

  Returns the type of a given variable.

- type:

  Returns the return type of a given function.

- isDefined:

  If a given value is "undefined," returns *false*. Otherwise, returns *true*.

- print:

  Prints the textual representation of the given value, and returns *true*.

- println:

  Prints the textual representation of the given value, followed by a new line, and returns *true*.

- regex:

  Returns *true* if the second argument is an instance of the first one, which should be a pattern of regular expressions.

- concat:

  Returns a new string which is the concatenation of the two string arguments.

3.2.3 FCL's Treatment of "undefined"

In FCL several ways can lead to a value of "undefined." Casting an element of one type to another can generate values of "undefined." For instance, casting a type that is not a class to a class will yield the value "undefined" as the result. Asking for a receiver expression from an arithmetic expression will also result in "undefined."

For any function applications with values of "undefined" as arguments, if the return types of the functions are basic types but not boolean, then the result will be a value of "undefined." But if the return type is boolean, then the function application will return *false*. If the return types are compound types, that is, sequences or sets, then the function application will return empty sequences and empty sets respectively.

Further detail on other treatments of "undefined" can be found in [Jac02] and [Par01e].

3.3 Implementation

FCL requires a FCL parser and interpreter and a program database. The program database is populated with a link program called *dxlinker*. Our prototype implementation consists of about 20,000 lines of C++ code: 11,000 lines for dxlinker and 9,000 lines for FCL parser and interpreter.

3.3.1 Program Database

A program database for FCL should contain sufficient information so that the object model described in Chapter 4 can be built. Our implementation adopts an *abstract semantic graph* representation of source code. The program database is stored as a textual file using the Datrix schema [HHL+00]. At run-time such a representation is mapped to an object-oriented representation where source code entities are strongly-typed objects.

3.3.2 Parser and Linker

A parser is responsible for extracting facts out of C++ source code and storing them in compliance to the Datrix schema. A linker then links the multiple graphs into one single graph but still stores it in the Datrix schema. Conceptually a linker is needed to build a graph where relationships such as all subclasses of a given class can be *conveniently* identified and facts irrelevant to FCL, such as files and redundant declarations, are eliminated.

In our implementation we chose *dxparscpp* [HHL+00] from Bell Canada as our parser and developed our own linker program *dxlinker* [HHR03].

3.3.3 FCL Interpreter

FCL interpreter accepts two files as arguments from command line: one storing FCL specifications and the other a program database. It checks if all constraints are satisfied and reports error messages if any of them is violated. Specifically, the FCL specifications are first parsed using a Bison-generated parser. FCL interpreter then type-checks the FCL specifications to rule out errors such as applying FCL function var to a Variable or using a Variable as the scope of a quantifier.

The interpretation of constraints is done within the context of the given program database. For each constraint, FCL interpreter evaluates its components first and then the constraint itself. Primitive functions are interpreted directly against the data model. A formal treatment of the semantics of FCL can be found in Appendix A, which provides more details on the interpretation of FCL constraints.

3.4 Complexity Analysis

FCL is designed to be tractable at the first place. For instance, although Int appears in the type system of FCL (Figure 3.1), quantifying directly over the *infinite* integer set is not allowed. All sets in FCL specifications are finite.

In theory the complexity of evaluating FCL constraints is exponential. For example, for the constraint presented at the beginning of this chapter, if the number of subclasses is M and the maximal number of expressions within all member function m is N, then the

complexity of that constraint is MN.

However, in practice we have not found performance to be a real problem: the deepest level of nesting quantification formulas for the examples we have tried is 4, and the performance of our simple interpreter is acceptable. On a laptop running Red Hat Linux 2.4.18 with a Pentium II Celeron 300 MHz CPU, all MFC constraints presented in Chapter 6 are done in less than 20 seconds. Considering the size of the ASGs for the MFC examples (10-15 MBs), we do not think that performance will be a big problem for the acceptance of FCL.

Chapter 4

A Model of Static Properties of C++ Programs

This chapter presents an object-oriented model of the structure of C++ programs, which provides an anatomical view of program structures [1]. For instance, in our model, a function application f(p) will be decomposed into two parts: the function name, f, and the sequence of parameters, p. Therefore, the model is a meta one concerning with the elements of programs. We will use the UML (Unified Modeling Language) class diagram to depict the model.

The class diagram of Fig. 4.1 depicts the overall model. It forms the foundation of the FCL language, whose syntax and semantics will be presented formally in chapter A. Due to the limitation on visual space, some details, e.g., the kinds of expressions, have been left out of the diagram; they will be further explained in later sections.

The goal of this chapter is to "digest" the diagram. We will proceed in the following order: section 4.1 explains the top three levels, which abstract out the common properties of all program elements; section 4.2 program units, the constructs for structuring programs; section 4.3 types; section 4.4 operators, the constructs for defining computation; sections 4.5 and 4.6 expressions, the applications of operators.

4.1 Program Elements, Contexts, Names and Types

A program consists of a variety of *program elements*, which can be a class, a function, a variable, an expression, or a class or function template, and so on. Program elements are associated with each other through various relationships, for instance, that between a class and its members, a function and its parameters, a loop or conditional and its condition, variable declaration and references, inheritance and friendship, and so forth.

Program units are a special kind of program element that help to organize programs;

[1] In the literature, there seem to exist at least two notions of structures, the semantics one and the syntactic one. Our structure concerns with the abstract syntax of program languages.

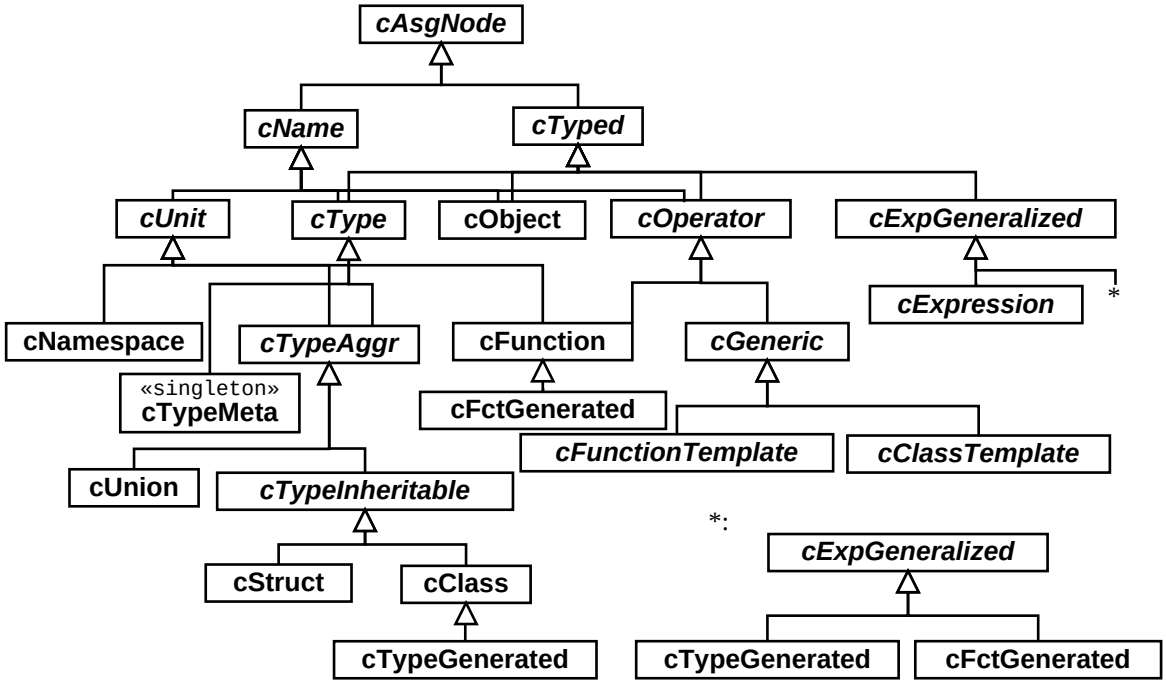

Figure 4.1: Object model for C++ programs

namespaces, classes, functions, and blocks are program units. In particular, the global namespace is the topmost level program unit.

Each program element exists in a certain *context* that is composed of the sequence of program units that enclose the element. For example, a class defined within the global namespace takes the global namespace as its context; an expression within a member function will have the function, the class, and the context of the class, in this order, as its context.

Each program element can also have a textual representation for display purpose, which can be useful for diagnosis. For example, the textual representation for an expression `a+b` can be "a+b"; a variable `x` of type `int` can have "int x;" as its representation.

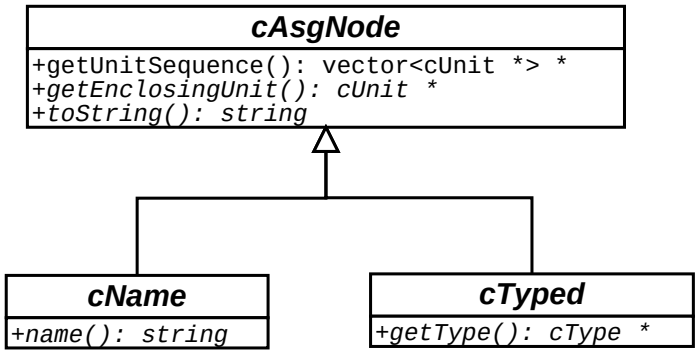

Figure 4.2: The class `cAsgNode` and subclasses

The class cAsgNode [2] [3] models the most common features of all program elements. For a given program element:

- getUnitSequence returns the vector of its program units, which are modeled by the class cUnit.
- getEnclosingUnit returns the innermost program unit that encloses it.
- toString returns the textual representation.

getUnitSequence and getEnclosingUnit together form a simple instance of the so-called template design pattern [GHJV94]. The former is implemented on the basis of the latter. And since getEnclosingUnit is virtual, derived classes of cAsgNode can override it to implement different ways of obtaining their corresponding innermost enclosing units.

Some program elements like classes and variables have names while others like expressions do not. The abstract class cName is effectively the interface for any named element.

Similarly, some elements are typed while others are not. If the element is an operator such as a function or class template, then the type is its algebraic type; for instance, the arithmetic addition on integer set has the type $Z \times Z \rightarrow Z$. If the element is a variable or an expression, then the type is the type of its value.

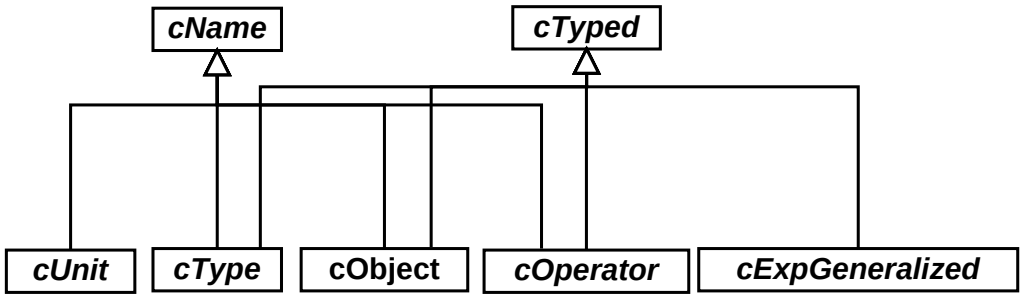

Figure 4.3: Abstract classes cName and cTyped

A number of classes inherit the classes cName and/or cTyped. cUnit is the abstract class for all the program units, cType for types, cOperator for operators, and cExpGeneralized for expressions, respectively. These will be described in the following sections.

The class cObject models all kinds of variables. A variable can be either a global one or a local one, it can be a data member of aggregate types, or it can be a parameter to a function. The kind which a variable can be (being *local* or *global*, for example) is determined by examining its context information.

[2] As a naming convention, all classes of the FCL object model start with letter *c*. The name *cAsgNode* is due to the Datrix tool that we use: Datrix models programs as graphs called ASGs (Abstract Semantics Graph) [RW91], whose nodes are program elements.

[3] In this chapter, the font sans serif is used for names such as method names and class names in the FCL model.

4.2 Program Units

Program units are mechanisms that help organize programs. C++ supports four kinds of program units: namespaces, aggregates, functions, and blocks. FCL supports a limited amount of control flow analysis (section 4.6), where no block information is needed. Thus the current object model does not treat blocks as the first class entity.

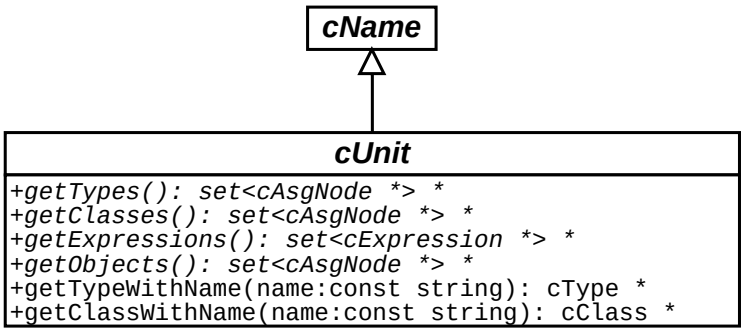

Figure 4.4: Program units

Essentially, the class cUnit provides a "container" view to program units. That is, a program unit can contain new types, expressions, and/or variables. Thus it should support queries on its internals, both collectively and individually.

A program unit may contain new types; the new types can be, for instance, an aggregate type, an enumeration, or a template-generated type. In particular, the global namespace contains all the built-in types. getTypes returns the set of all types that the current unit contains, while getClasses returns only the subset of aggregate types. Given a name, getTypeWithName returns the type that has the name, while getClassWithName returns the class. If there is no such a type or class, the respective method will return null.

A program unit may contain expressions. For example, the set of expressions for a function includes both its default arguments and the expressions within its body (the current version of FCL does not handle exceptions yet). The set of expressions for an aggregate type or a namespace consists of all the initialization expressions. getExpressions returns the set of expressions of a unit.

A program unit may also define a set of variables. For a function, the set of variables includes both its parameters and local variables; for a class, the set consists of all the data members. getObjects returns the set of variables defined by a program unit.

Namespaces, aggregates, and functions are the concrete program units that FCL currently supports. In addition to the properties captured by the class cUnit, each of them also has special ones:

- A namespace may have sub-namespaces defined within it. getNamespaces returns the set of immediate sub-namespaces.

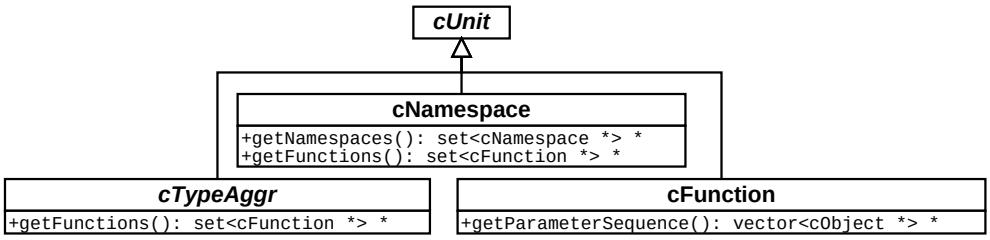

Figure 4.5: Namespaces, aggregates, and functions

- Both namespaces and aggregates can have functions. getFunctions returns the set of functions.
- Each function has a sequence of parameters. getParameterSequence returns the sequence for the function.

4.3 Types

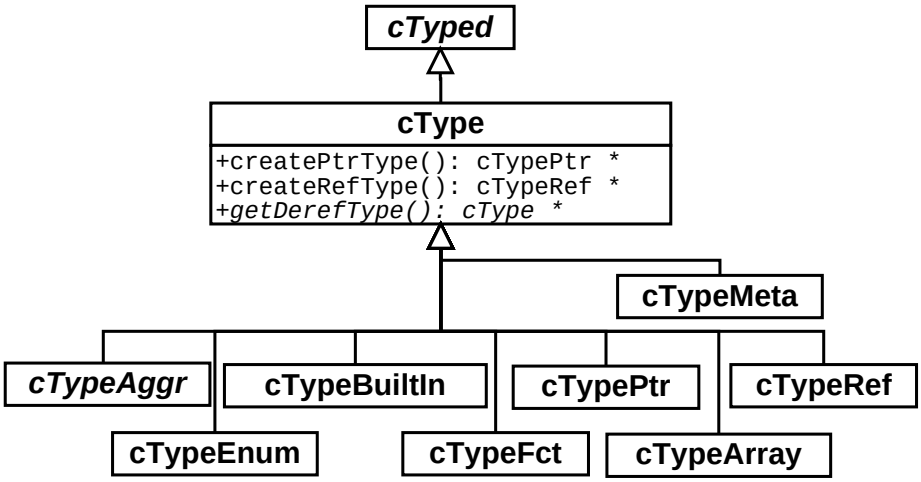

Figure 4.6: Types

C++ supports seven different types: built-in types, pointer types, reference types, enumeration types, function types, array types, and aggregate types. In the FCL object model, each of these types has a corresponding class (for instance, an aggregate type is an instance of the class cTypeAggr). The class cType is the abstract interface for all of the seven classes.

The class cType inherits cTyped, thus types themselves are also considered being "typed." That is, each of them has a type. The singleton class cTypeMeta is used to denote the type of types. The reason for assigning a type to types is that in C++, types can be referenced as expressions, and all expressions have types.

The class cType defines the following operations:

- createPtrType: Given a type, createPtrType returns the pointer type to it. If the type

is a meta type, that is, the instance of cTypeMeta, createPtrType returns the type itself.

- createRefType: Given a type, createRefType returns the reference type to it. If the type is a meta type, that is, the instance of cTypeMeta, createRefType returns the type itself.
- getDerefType: If the given type is an indirect type, getDerefType returns the type that it points to; otherwise, getDerefType returns the type itself. For example, if the type is of the form "**T", where T is not indirect, applying getDerefType to it will yield the type "*T". If, however, the type is "T", then the result of applying getDerefType to it will be "T" itself.

4.3.1 Aggregate Types

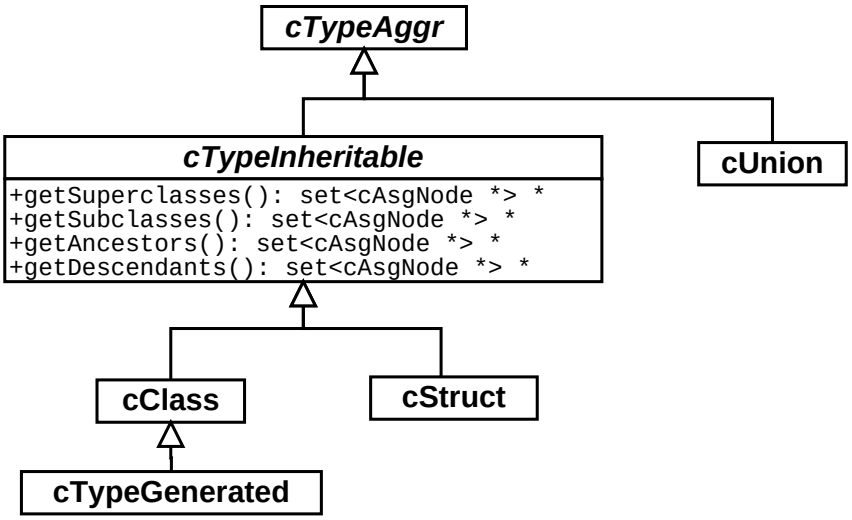

Figure 4.7: Aggregate types

C++ supports three kinds of aggregate types, class, struct, and union (Fig. 4.7). Only class and struct can be part of an inheritance hierarchy. Given such a type, one can query its direct base classes and derived classes, ancestor classes, and descendant classes, from the inheritance graph. These commonalities are captured by the class cTypeInheritable. Both class cClass and class cStruct are its subclasses. In particular, the class cTypeGenerated, for types generated from class templates, is a subclass of cClass.

In contrast to class and struct, a union can be neither a base class nor a derived class. The class cUnion is for unions.

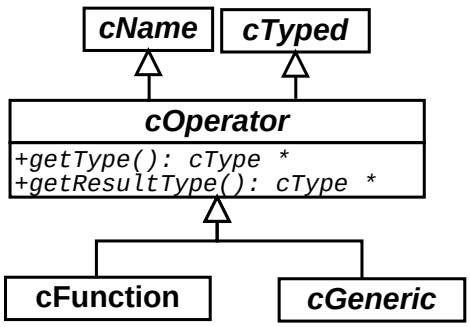

Figure 4.8: Operators

4.4 Operators

Informally, an operator [4] is something that can be applied to some operands. Examples of operators are functions and class and function templates (class templates and function templates are modeled as subclasses of cGeneric).

The types of operators are their algebraic types, and getType returns the types. But for a function, in addition to its type, normally one would also be interested in its return type. The operation getResultType can be used to get the return type [5].

4.5 Expressions

An expression is a sequence of operators and operands that specifies a computation. An expression can result in a value and can cause side effects. FCL recognizes not only "normal," first-order expressions like arithmetic operations and function calls, but also second-order expressions that either take types as operands or generate types as values. In Fig 4.9, cExpression models first-order expressions. cTypeGenerated and cFunctionGenerated model instantiations of class templates and function templates respectively.

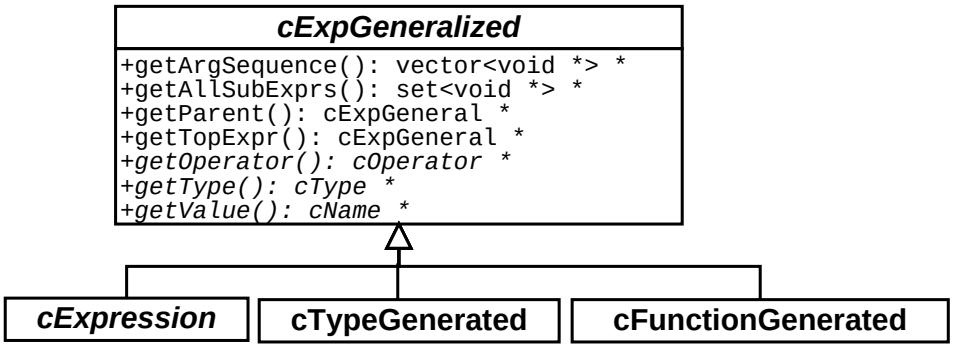

Figure 4.9: Expressions

[4] The term operator is much overloaded; here it does not refer specifically to C++ operators, and its meaning is more general than that in C++.

[5] The current version of FCL has not been used to check any property of a function type or a template type. getResultType is only a shortcut for obtaining the return types of functions and templates.

Regardless of being first- or second-order, all expressions share some common properties. Any expression can have sub-expressions as operands, which in turn can have their own sub-expressions as operands, and so on. Thus, structurally, an expression can be viewed as a tree, whose nodes represent expressions.

Consequently, cExpGeneralized supports the following "tree" operations:

- getArgSequence:
  Each expression can have a sequence of arguments, which are also expressions. The sequence can be obtained through getArgSequence. For instance, for y++ and *y, the argument would be y, and for y=x+z(j), the arguments would be y and x+z(j).
- getAllSubExprs:
  It returns the set of all sub-expressions of an expression.
- getParent:
  If this expression is a top level expression, then getParent returns this expression itself; otherwise, it will return the expression in which this expression is an argument.
- getTopExpr:
  Returns the expression corresponding to the root node of the tree where this expression "lives".

The operator and result of an expression can be obtained through the following operations:

- getOperator:
  Each expression is the result of invoking an operator on some operands (arguments). For instance, a function call is the result of invoking a corresponding function. The arithmetic expression a+b is the result of applying the pre-defined operator, $+ : Z \times Z \rightarrow Z$, to two integers, a and b. Particularly, a generated type is the result of applying a class template to operands of types and/or expressions of constant values. getOperator returns the operator that the expression uses.
- getType:
  The result of evaluating an expression has a type, and getType returns the type. For example, for a function call, getType will return the return type of the function being invoked; for a generated type, getType will return cTypeMeta.
- getValue:
  For first-order expressions [6], most of the time static analysis does not know their values. In that case, getValue returns null. For second-order expressions, getValue returns the types or functions generated.

[6] Expressions whose operands are not types are considered first-order, otherwise, second-order.

4.5.1 Generated Types

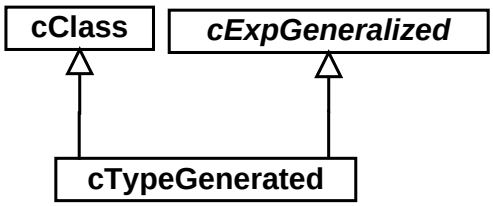

Figure 4.10: Generated types

Like expressions, a generated type also has a sequence of arguments. Unlike expressions, the arguments of a generated type can only be constant values and types. In addition, the value of a generated type is the result of instantiating a class template with concrete arguments, and it is viewed as a class.

4.6 "Normal" Expressions

The class cExpression models expressions other than generated types, hence the name "normal." Such an expression can be a function call; a pre-defined operation such as arithmetic, relational, and logical expressions; a cast call or a sizeof expression, both of which involve a type as their operand; and name references (to variables and to types) and literals.

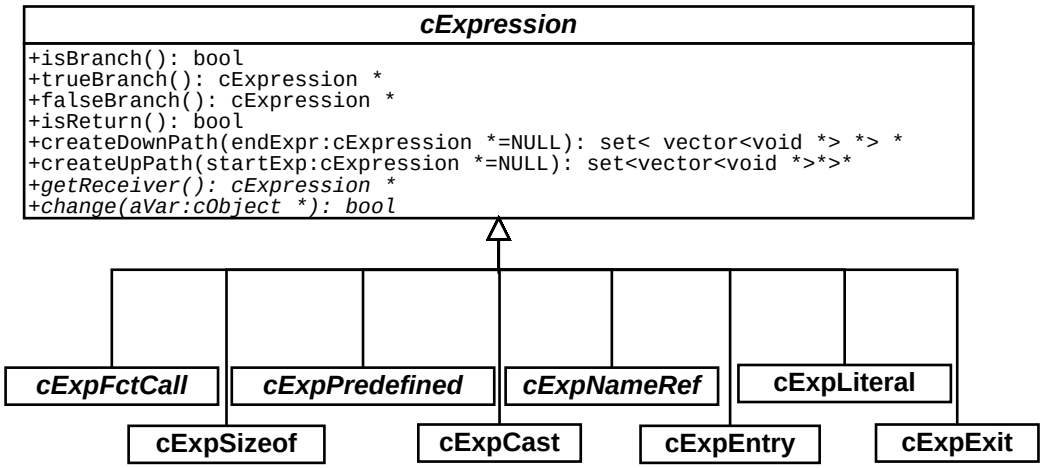

Figure 4.11: Normal expressions

An expression can appear at several places. It can be an intializer to a static data member, a constructor, or to a variable. It can also be a default argument to a function. Of course, most often an expression will be a computational step within a function.

FCL builds a CFG (Control Flow Graph) for each function. The nodes of CFGs are expressions. Particularly, two artificial expressions, of types cExpEntry and cExpExit, respectively, are added to each CFG: cExpEntry represents the starting point of control, and

cExpExit represents the end of control. Edges are also added, respectively, from the cExpEntry node to the entry expressions of a function, and from the exit expressions to the cExpExit node.

Six operations are available for querying control flow related information about an expression:

- isBranch:

  Given an expression, if it is the condition of a conditional or loop statement, isBranch returns true; otherwise, isBranch returns false.
- trueBranch:

  Given an expression, if it is the condition of a conditional or loop statement, trueBranch returns the first expression on its true branch; otherwise, trueBranch returns null.
- falseBranch:

  Given an expression, if it is the condition of a conditional or loop statement, falseBranch returns the first expression on its false branch; otherwise, falseBranch returns null.
- isReturn:

  Given an expression, if it is the expression of a return statement, isReturn returns true; otherwise, isReturn returns false.
- createDownPath:

  If this expression is within a function, createDownPath(endExp) returns the set of paths starting from this expression and ending at endExp. By default, endExp has the value of cExpExit. If this expression is not within a function, then createDownPath returns the empty set.
- createUpPath:

  If this expression is within a function, createUpPath(startExp) returns the set of paths starting from startExp and ending at this expression. By default, startExp has the value of cExpEntry. If this expression is not within a function, then createUpPath returns the empty set.

The following example may help illustrate control paths:

```
(*)  starting point
(1)  if (a+b<100)
(2)     doSomething();
(3)  if (a+b>50)
(4)     doTheRest();
(*)  exiting point
```

The set of down paths from the function call at (2) to the exiting point is { <a, b, a+b, 50, a+b>50, doTheRest>, <a, b, a+b>50> } . The set of up paths to (2) from the starting point is { < a, b, a+b, 100, a+b<100 > }.

The evaluation order of the constituent expressions is implementation-dependent [Int98]. The example assumes a left-to-right order.

The operation getReceiver returns the receiver expression r for an expression of the form "r.m(...)." Otherwise, that is, if the expression is not sending a message to an object, getReceiver returns null.

bool change(cObject * aVar) tells whether a given expression may change the variable represented by the parameter aVar.

If aVar is not referenced by the expression, change returns false. Otherwise, for each of the expressions where aVar is referenced:

- If the expression is a pre-defined one, the pre-defined operator is an assignment, and aVar is the left-hand side of the assignment, then change returns true.
- If the expression is a function call and aVar is the receiver: If the function invoked is not const, then change returns true; otherwise, change returns false.
- If the expression is a function call and aVar is an argument to it: If the corresponding parameter of the invoked function is not const but of reference type, then change returns true; otherwise, change returns false.

If aVar is of pointer types, one may want to query whether an expression changes the value that aVar points to. Currently, FCL has yet no constructs to express this.

4.6.1 Function Calls

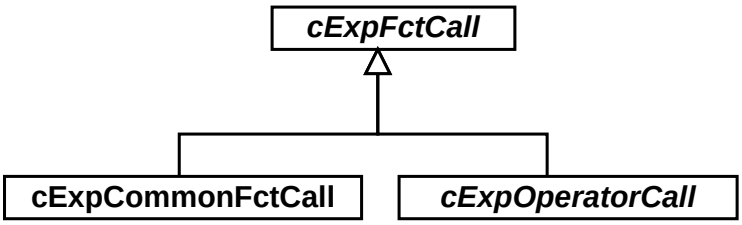

Figure 4.12: Function calls

Function calls are further divided into two categories: common function calls and operator calls. Invocations of both free functions and member functions are common function calls. Operator calls correspond to functions that use operators as names; in C++ operators can also be overloaded.

4.6.2 Pre-defined Expressions

Pre-defined expressions invoke the pre-defined operators. FCL puts pre-defined operators into the global namespace and treats the operators and their expressions in the same way as user-defined functions.

Assignments	= &= |= ^= >>= <<=
Bit manipulation:	& | ^ >> << ~
Memory	new delete & * . -> .* ->*
Arithmetic	+ - * / % ++ (pre) -- (pre) ++ (post) -- (post)
Logical	&& || !
Relational	> >= == < <= !=
Special	, ?: []

Table 4.1: Pre-defined Expressions

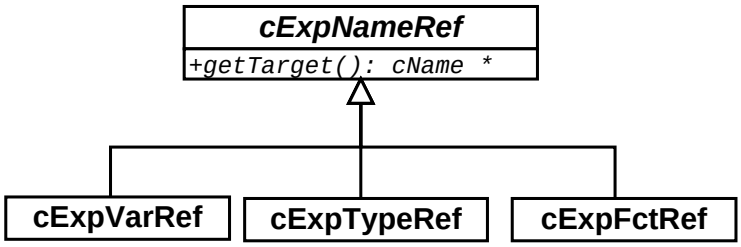

Figure 4.13: Name reference expressions

4.6.3 Name Reference Expressions

An identifier can be bound to a variable, a type, or a function. For a given name reference expression, getTarget returns the corresponding elements that it is bound to.

4.6.4 Control Statements

Control statements are not treated as an essential part of the FCL object model. The reason is that by building control flow in terms of expressions, many important constraints can be handled without the need of information about control statements. They may be needed when, say, one wants to enforce certain coding conventions. But that is not the focal point of FCL at the moment.

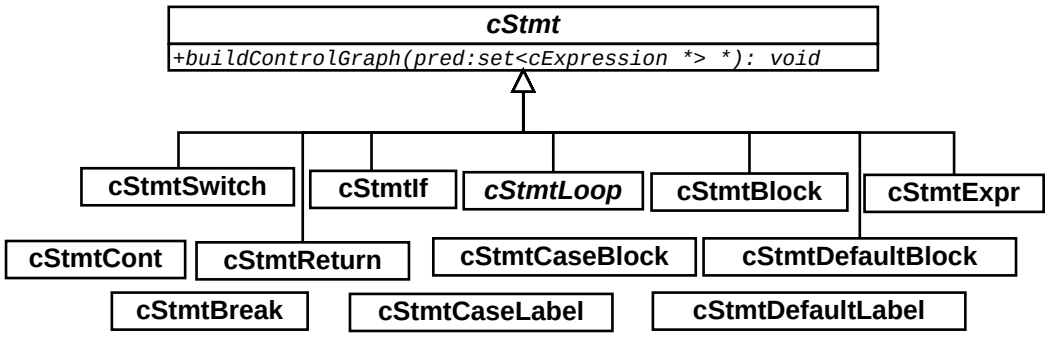

Figure 4.14: Control statements

Control statements are used to generate control flow graphs for functions. The class diagram for C++ statements is depicted in Fig. 4.14. The control statements of a function form a tree structure. The control flow graph is built by a syntax-directed traversal of the

tree, connecting the corresponding expressions at each node.

The following example may help illustrate the idea:

```
void f()
{
 E1;
 if (E2)
    B1;
 else B2;
 E3;
}
```

whose CFG is shown in Fig. 4.15.

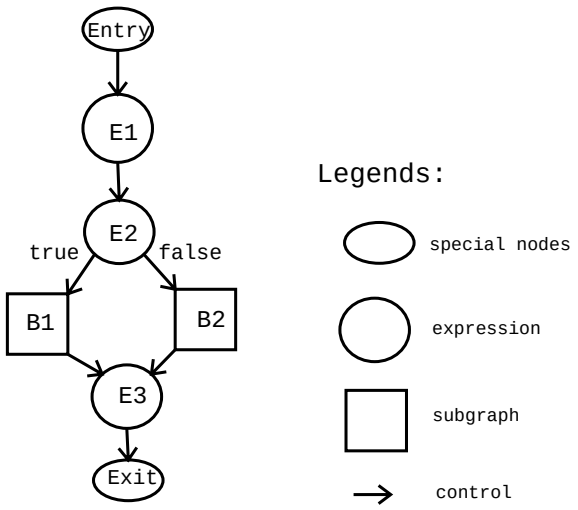

Figure 4.15: A CFG example

More details on constructing CFGs from statements can be found in [AU78].

Chapter 5

Case Study 1: Enforcing Framework Constraints–the Observer Design Pattern

The purpose of the Observer example is to introduce the main features of FCL. It also illustrates how to use the checker program, *fcl*, both the format of its command line and the format of its output message.

5.1 The Observer Design Pattern

Design patterns [GHJV94] can be seen as small frameworks made of a few classes. We introduce FCL by specifying constraints for an implementation of the Observer design pattern (Figure 5.1).

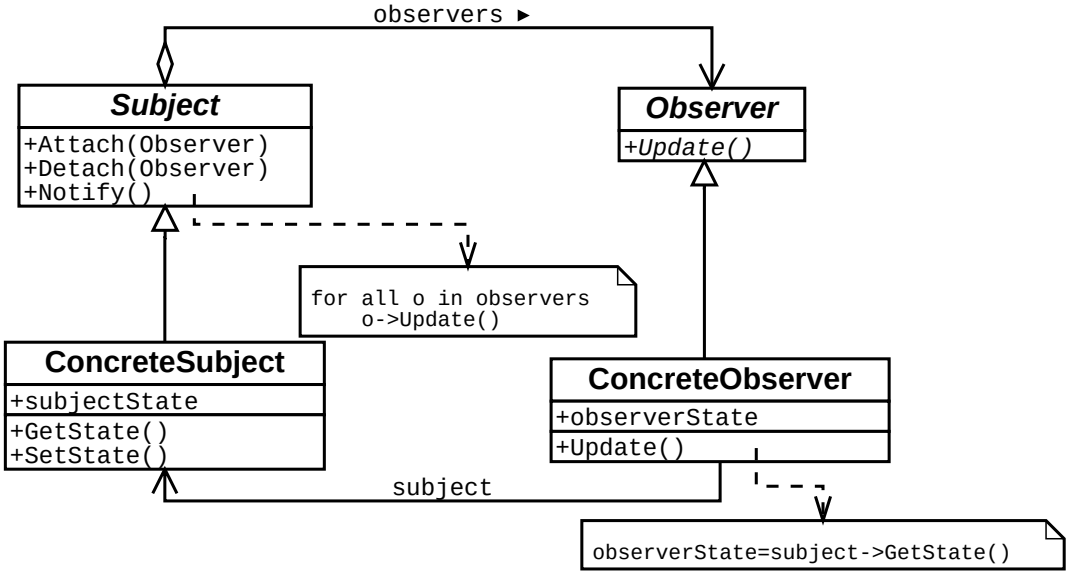

Figure 5.1: The Observer Pattern

Together, the abstract classes Subject and Observer implement an abstraction for the notification of state changes, as depicted in the top of Figure 5.1. The class Subject provides an interface for attaching and detaching Observer objects; a subject knows its observers. Any number of observers may observe a subject. The Observer class defines an updating interface for objects that should be notified of any state changes in a subject. A subject should notify all of its observers whenever its state changes. This can be done by calling the notify method of the class Subject, which in turn, calls the update method of each observer object, with the subject itself as the actual argument. As a response to the notification, each observer object should query the subject object to synchronize their states.

Let's assume that any extensions to the framework are required to have at least one subclass of the class Subject. Then, as designers, we can stipulate that each subclass of the class Subject must obey the following constraints:

1. The Subject class has to define some state, thus it must define at least one instance variable to represent the state.
2. All the instance variables must be private for sake of information hiding.
3. The class must define at least two methods. Because all of its instance variables are required to be private, at least one method is needed to change the state and another to query it.
4. In the class, there must exist at least one method that satisfies the following conditions:
   - First, it changes some new variables defined by the class.
   - Second, it calls the inherited notify method.
   - And third, the change must happen before the notify method is called.

   We use the name *modifier* for the set that contains all such methods.

Similarly, we require that:

1. Any extensions to the framework must have at least one subclass of the class Observer.
2. Furthermore, all the subclasses of Observer must override the update method.
3. In addition, the override must call some method defined in a subclass of the class Subject. But it is prohibited to call any methods that belong to the set *modifier*.

5.2 Observer Pattern in FCL

The following is an FCL specification for the Observer pattern:

```
1     subject as class("Subject");
2     observer as class("Observer");
3     model as subclass(subject);
4     view as subclass(observer);
5
6     forall mdl :  model holds
7     [
8     modifiers as { m :  function(mdl) |
```

```
9          exist e :  exp(m), v :  var(mdl) holds
10          (
11               change(e, v)
12               and
13               exist p :  downPath(e), notify :  p holds
14                    name(notify)="notify"
15          )
16          }
17     ]
18     (
19       for v being var(mdl) holds
20            isPrivate(v)
21       and
22       sizeof(function(mdl)) > 1
23       and
24       sizeof(modifiers) > 0
25       and
26       forall v :  view holds
27       [
28       update as { m :  function(v) |
29            [
30              p as param(m);
31              firstParam as member(p,0);
32            ]
33            (name(m) = "update" and sizeof(p) =1 and
34            type(firstParam) = ptr(subject))
35            and
36            forall e :  exp(m) holds
37                 not belongsTo(function(e), modifiers)
38            and
39            exist e :exp(m) holds
40            [
41              invokedFunction as function(e);
42              programUnits as unit(invokedFunction);
43            ]
44            (
45                 belongsTo(first(programUnits), model)
46                 and
47                 last(programUnits)=global
48            )
49            }
50       ]
51       sizeof(update) = 1
52     )
```

Overall, this specification maps relatively straightforward with the natural language description of the last section. For example, lines 7 through 24 correspond to the items 1 to 4 for the Subject class; the constraints of item 4 are reflected in the set comprehension defined at lines 8 through 16. In the following, we remark on some peculiarities of the specification.

The overall structure of the specification consists of two nested forall formulas, starting at lines 6 and 26, respectively. This structure is used because constraints for each subclass of the Observer class need to refer to the set *modifier* defined for each subclass of the Subject class. Should there be no such a dependence between subjects and observers, we would have had two independent forall formulas instead.

Note that both *subject* and *observer* are FCL variables that represent the Subject class and the Observer class, respectively; they are constructed through the constructor function *class*. Moreover, the FCL variables *model* and *view* represent the respective sets of

subclasses.

The set comprehension that defines the set *modifier* deserves some comments. One question many may ask about the *change* predicate at line 11 goes along the lines of "what if a subclass of the Subject class defines an instance variable of integer type, which does not represent state, but some methods happen to change it, and thus make the code satisfy the constraint?" In that case, indeed, FCL will miss the error, and there will be no error messages reported. This example shows that although FCL can detect errors of omission, it is not complete (that is, it generates false negatives); in particular, sometimes an error might be camouflaged by code that happens to expose the same structure as expected, and thus satisfies the constraint.

This specification also examplifies the use of sequences. FCL needs sequences to model three things, the execution paths within a method, the parameter list of a method, and the syntactical context of a program element. In the above specification, there are three examples of sequences, at lines 13, 30, and 42, respectively.

The function *downPath* at line 13 returns the set of all the execution paths that start from the argument *exp* and end at the end of the method; each element of the set is a sequence of expressions.

The function *param* at line 30 returns the parameter list of the function *m*. At the next line, *member(p, 0)* returns the first element of the sequence; in fact, it is the first and the only parameter of the function, as indicated by the constraint at lines 33 and 34. At line 34, *ptr* is a constructor function that constructs a pointer type to the subclass *subject*.

The last example is at line 42: the *unit* function returns the sequence of program units that enclose the argument, which is a function in this case. Furthermore, the specification goes on to check whether the class of this function belongs to the set *model*; the function *first* at line 45 effectively obtains the enclosing scope of the function. If the return value is a class that belongs to the set *model*, then the *belongsTo* test will yield *true*. Finally, the formula at line 47 is a tautology since the last element of a context is always the global namespace; the function *last* returns the last element of a sequence, and *global* is a constant of FCL that denotes the global namespace.

5.3 Discussion

5.3.1 Expressions Are Important for FCL

In addition to classes and methods, it is also important to be able to specify constraints for expressions. Without support for expressions, the content of the above specification would drop rapidly; all that would remain are the requirements that the subject subclasses should have both some methods and variables defined and that the observer subclasses should override the update method. These are still useful, but their likelihood of detecting subtle

errors is reduced.

Some of the constraints are implementation oriented, for example, the one that requires all the instance variables of a subject class be private and the one on the number of methods. Although they may not be as important as design constraints, it is worth enforcing them.

Certain parts of a specification can get very detailed; for example, to accurately specify a method, one needs to provide not only its name, but also the number of parameters and their respective types.

Context knowledge can be used to shorten FCL specifications. At lines 13 and 14 of the above specification, we only check whether there is a function call expression that invokes a function named *notify*. This may be insufficient under certain circumstances; for example, the function being called may be a global one that has the same name. On the other hand, within the specialized context of a project, it may be the case that this will never happen. Therefore, the specifier may take advantage of the knowledge, and the above specification becomes acceptable. For another example, notice that line 34 specifies the *update* in full detail. If one knows that the *update* method will not be overloaded, then one can eliminate the code that specifies the parameters.

5.3.2 FCL Depends on Specialized Context

Conventional methods of program specification emphasize abstractions and generality [GH93]. This is achieved through using concepts and constructs such as specification variables and abstraction functions in the specifications. By concentrating on only properties of the abstractions, one obtains the benefits of easier maintenance and a higher chance of reusing the specifications. With abstractions, it becomes clear what properties must be preserved when one changes the implementations. By programming to abstractions, an abstraction can be used in various contexts without the programmer having to worry about the underlying implementations. Moreover, one implementation can be replaced by another without affecting correctness.

In contrast, FCL depends on specialized context; it specifies constraints in terms of implementation details such as variable names, expressions, and specific classes and methods. However, FCL complements rather than contradicts the established wisdom. Traditional methods focus on identifying and specifying the core abstractions within a problem domain. This is of course a right strategy. But frameworks by definition already contain the key abstractions of the corresponding problem domains. Therefore, FCL is not concerned with the correctness of the abstractions themselves. In contrast, FCL is addressing a different issue; it is meant to detect the potential errors that may occur at the boundary between the core and the extension. This requires that FCL specifications explicitly mention implementation details.

Specific implementations of a generic design may specialize in certain aspects. For example, there are several dimensions along which a particular implementation of the generic observer design pattern can vary:

States Both primitive types and user–defined types can be used to represent states. Furthermore, if the state representation is of a collection type, the state may be either the collection, the elements that it contains, or both. Clearly, at the general level, without further information, what FCL can speak about is limited. The best that one can say about the state is probably that "there must be a change to the state, and it is followed by a call to the notify method." Although quite conservative, it captures the key requirement that the user of the pattern has to fulfill. When specifying constraints, one has to balance between picking up more errors and avoiding spurious ones.

There are also examples of the pattern in which the state is divided into substates and, subsequently, a different notification method is provided for each individual substate. The JTree component of the Swing framework provides a default implementation for the TreeModel interface, which corresponds to the *Subject* class of the Observer pattern. For performance reason, the default implementation distinguishes four kinds of state changes: change to the whole tree, addition of nodes to or removal from a particular internal node, and change to a whole sub–tree.

Registration Where can an observer be attached to a subject? The registration can be done by the client of the pattern, that is, outside of both the subjects and observers. It can be done by the observers; the JTree class actually registers itself to the tree model. It is also conceivable to do it in the subject classes.

Cardinalities Not only can each subject have multiple observers, but each observer can watch multiple subjects. Moreover, these subjects can belong to different classes.

Who calls notify? Notification does not have to always be issued by the methods of subject classes. It can also be done by the client of the pattern. The related advantages and disadvantages are discussed in [GHJV94].

Specializations can influence the content of the specifications. In the following, a special implementation of the observer pattern will be used to illustrate this point.

The implementation of Figure 5.2 makes several changes to the generic one:

1. In this example, the state is a name of the string type, and a modifier *setName* is implemented to change it. As a result, the class *Subject* is not abstract any more.
2. The *Observer* class also becomes concrete. In addition, it adds a private data member to remember the name. It also implements a method *printName*.

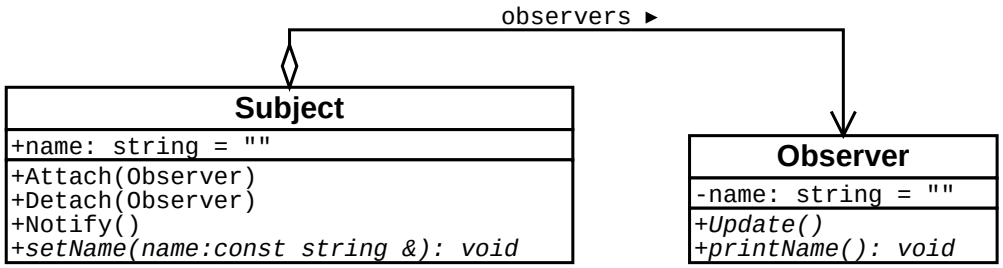

Figure 5.2: A Specific Implementation of the Observer Pattern

3. Both *setName* and *printName* are virtual.
4. The data member *name* of the *Subject* class is deliberately made public to illustrate how FCL can be used to restrict the access to data.

The FCL specification for this implementation is as follows:

```
1     // Constraint 1:  not call attach
2     setOfExpInFreeFunction as { fct:  function(global) | true | exp(fct) };
3     expInFreeFunction as union(setOfExpInFreeFunction);
4     setOfExpInClass as { cls:  class(global) | true | expClosure(cls) } ;
5     expInClass as union(setOfExpInClass);
6     attach as function("attach", class("Subject"));
7
8     exist exp being expInFreeFunction+expInClass holds
9       function(exp) = attach
10
11     // Constraint 2:  directly assign to Subject::name
12     Subject as class("Subject");
13     Observer as class("Observer");
14     classesOfSubjectAndObserver as [Subject, Observer ]+descendant(Subject)+
15          descendant(Observer);
16     expOfSubjectAndObserver as union( { cls:  classesOfSubjectAndObserver |
17                         true | expClosure(cls) } )
18
19     // version 1:
20     not exist exp :  expInFreeFunction+expInClass-expOfSubjectAndObserver holds
21     (
22       name(exp)="="
23       and
24       [arg as arg(exp); leftHand as member(arg,0);
25       arg1 as arg(leftHand);
26       receiverType as ptdTypeStar(type(member(arg1, 0)));
27       varRef as member(arg1, 1);]
28       (
29         name(varRef)="name"
30         and
31         belongsTo(receiverType, classesOfSubjectAndObserver)
32         and println(name(varRef))
33       )
34     )
35
36     // version 2:
37     forall exp being expInFreeFunction+expInClass-expOfSubjectAndObserver holds
38     (
39     not (
40       name(exp)="="
41       and
42       [arg as arg(exp); leftHand as member(arg,0);
43       arg1 as arg(leftHand);
44       receiverType as ptdTypeStar(type(member(arg1, 0)));
45       varRef as member(arg1, 1);]
46       (
47         name(varRef)="name" and
48         belongsTo(receiverType, classesOfSubjectAndObserver)
```

```
49        )
50        )
51        or not (println("the node for the expression is") and println(exp))
52      )
53
54      // Constraint 3:  NewObserver::printName does not call Observer::printName
55      NewObserver as subclass(class("Observer"));
56      printNameOfObserver as function("printName", class("Observer"));
57
58      forall observer being NewObserver holds
59      [
60        printName as { fct:function(observer) | name(fct)="printName" and
61                  sizeof(param(fct))=0 } ;
62        expOfPrintName as union( { fct:printName | true | exp(fct) } )
63      ]
64      exist exp being expOfPrintName holds
65        function(exp)=printNameOfObserver
66
67      // Constraint 4:  NewObserver::update calls Subject::setName
68      setNameOfSubject as function("setName", class("Subject"));
69
70      forall observer being NewObserver holds
71      [
72        update as{ fct:function(observer) | name(fct)="update" and
73                  sizeof(param(fct))=1 } ;
74        expOfUpdate as union( { fct:update | true | exp(fct) } )
75      ]
76      forall exp being expOfUpdate holds
77        not function(exp)=setNameOfSubject
```

There are four constraints in the specification. A few comments about them fall in place:

Constraint 1 requires that the *attach* method be called. To enforce this, the specification considers not only all the expressions of classes, but also those of free functions. The constructor *function* on *attach* returns a set. The equality at line 9 is actually a shorthand supported by FCL; it requires both that the set *attach* be a singleton and that its only element be equal to the left hand side.

Constraint 2 demonstrates how access to data can be restricted. Note that how verbose it can be to detect even such a simple assignment expression. The two versions of the constraint also show how the printing facility of FCL can be used. Currently, the *print* and *println* expressions are treated as predicates that alway return the value of *true*. It is up to the specifier to make sure that the use of them does not change the meaning of the original specification.

Constraint 3 is representative; it requires that the overriding methods call their corresponding superclass versions.

Constraint 4 requires that the overriding *update* method in any subclass of the *Observer* class do not call the *setName* method of the *Subject* class. This is a specialized version of the corresponding constraint for the generic observer pattern. It is considered "specialized" in that now we know what the modifier method is, and thus can directly use it to specify constraints.

Chapter 6

Case Study 2: Enforcing Framework Constraints-MFC

This chapter provides a few examples of framework constraints taken from the MFC framework. The purposes of these examples are twofold: first, they demonstrate that framework constraints are nontrivial in the practice of framework-based development; second, they show how FCL can be applied to a real framework and what lessons we can learn.

These examples are adopted from a third party, which brings us certain benefits in terms of the quality of our validation. First, it increases our confidence in the authenticity of the examples since contrived examples can be more easily and often accused of being not real. The quality of the examples also depends on the calibre of the author. In our case, the author, Dr. Joseph M. Newcomer, has a strong background in both the theory and practice of software construction, which can be evidenced by the following quotation from his web site [New]:

> Dr. Joseph M. Newcomer is a Microsoft MVP [(Microsoft Valued Professional)], an award presented to a fairly small group of people who help out on a volunteer basis on the Microsoft newsgroups. A collection of his articles, expanded beyond the postings on the newsgroups, is now available on his MVP essays page, as well as useful code samples.
>
> Joseph M. Newcomer is the author of several articles on Windows programming as well as co-author of two books on Windows programming and a course in writing Windows NT Device Drivers and a course in Windows System Programming. He also is co-author of three U.S. patents on distributed information technology.

Section 6.1 presents some preliminaries for MFC. Sections 6.2 through 6.10 present examples of MFC related problems and how FCL is used to detect them. Section 6.11 summarizes lessons learned.

6.1 Preliminaries of MFC

6.1.1 The Windows World and the MFC World

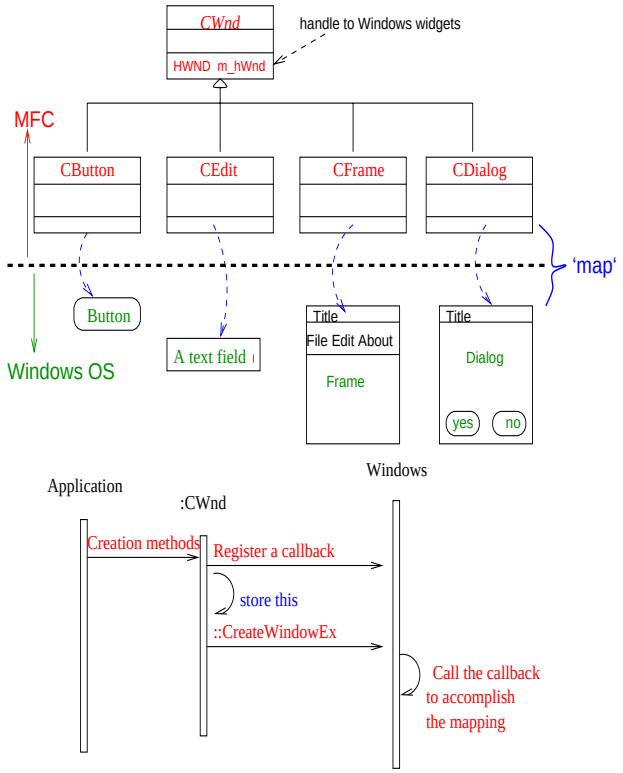

Figure 6.1: The relation between Windows and MFC

As an operating system, Microsoft Windows provides a set of APIs that can be used to create, manipulate, modify, and delete objects of both graphical interfaces and other resources. Graphical objects are often called controls. As usual in systems design, each control has a corresponding piece of information called 'handle' to identify itself; the type of handles is HWND. Similarly, resources are also managed through handles; for example, fonts have handles of type HFONT, and brushes have handles of type HBRUSH, and so on.

To support an object-oriented style of programming, MFC provides a set of "wrapper classes" that encapsulate the Windows controls and resources (top of Figure 6.1). For example, a CWnd object wraps an HWND, a CFont wraps an HFONT, a CBrush wraps an HBRUSH, and so on. Thus given a wrapper object, one can access its wrapped control through the instance variable m_hWnd. By default, MFC designates a default class for each type of controls and resources, for example, CButton for buttons, CListBox for lists, CEdit for input fields, and so on. But programmers can customize the default behavior of these classes through subclassing.

The wrapper classes provide various message handlers to handle the messages that the

underlying controls need to respond to. Moreover, operations are also designed so that one can move freely between the Windows object domain, where objects are represented by handles, and the MFC object domain, where objects are represented by instances of the C++ classes. In practice, it is important to understand the relationship between these two representations in order to use them in a safe and non-leaking fashion.

It is also known that when an event happens for a Windows control, the event loop mechanism can get the handle of the control from the operating system. Because the message handlers are defined as methods of the wrapper object, in order to find the right handler for the event, however, the event loop of MFC has to find the object first. It turns out that MFC uses a map data structure to maintain the relation between the handles and their wrapper objects; given a handle as a key, MFC can retrieve the corresponding wrapper object from the map.

The bottom of Figure 6.1 illustrates how the mapping relation is established. It starts by calling one of the creation methods for the wrapper object. After the method is invoked, it first registers a callback function for a certain pre-defined event of the operating system; the pre-defined event has the following property: when a control is created, Windows will generate an instance of the event, which will then trigger the registered callback function. Then, the creation method stores a pointer to the wrapper object into a global variable. Eventually, the method will call the windows API CreateWindowEx to *actually* create a control. As a response to the invocation of the API, Windows automatically calls the previously registered callback with the handle of the new control as an argument. The callback then retrieves the previously stored pointer to the wrapper object from the global variable and adds the pair of pointer and handle to the map.

6.1.2 Dialog, Control ID, GetDlgItem, and Control Variables

A dialog is a container that can contain other controls that have visual appearance. In MFC programming, a dialog is represented by a class that inherits the framework class CDialog. Each control of the dialog can have a constant integer called "control ID" associated with it. A control ID can uniquely identify the control and is only valid relative to the dialog that contains it.

The method CWnd::GetDlgItem is defined to retrieve a child control from a dialog. Note that CDialog inherits CWnd. Figure 6.2 depicts artifacts related to the implementation of CWnd::GetDlgItem. CWnd::GetDlgItem is implemented using ::GetDlgItem and the FromHandle method of the CWnd class.

The free function ::GetDlgItem is a Windows API function. It takes a handle to a dialog and a control ID and returns the handle to the control. Also note that the handle to a Windows control, HWND, is implemented as a pointer to a structure. In particular, a

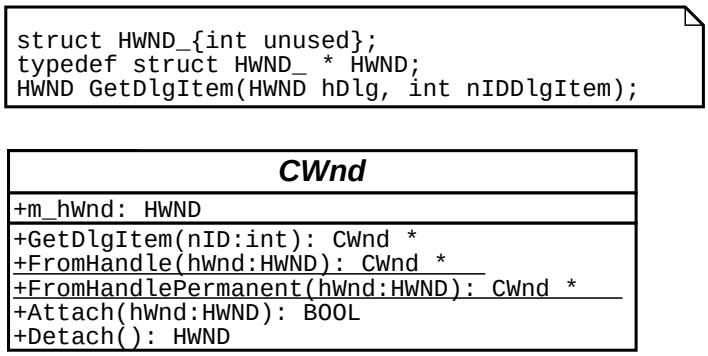

Figure 6.2: CWnd::GetDlgItem

dialog can be represented by a handle of type HWND.

The diagram for the class CWnd shows only a small portion of its implementation. Each CWnd object can wrap a Windows control, represented by the instance variable m_hWnd. Both FromHandle and FromHandlePermanent are static methods, which, given a handle, can look up the map for a corresponding CWnd object; the difference is that FromHandlePermanent will return NULL if there does not exist a CWnd object whereas FromHandle will return a temporary one in that case (and MFC can automatically manage the memory). Attach and Detach, as their names suggest, effectively add and remove a pair of handle and CWnd object to and from the map.

Sometimes one may want to customize the behavior of a control; this can be done by creating a subclass of the default MFC class for the control. For example, one may want to create a subclass, CHorzListBox, for the CListBox class; CHorzListBox overrides the AddString method to recompute the horizontal extent of the box and call SetHorizontalExtent, and consequently also overrides ResetContent to set the horizontal extent to 0.

Each control of the dialog may also be represented by an instance variable of the dialog class whose type is the corresponding class of the control. The instance variable is then called a "control variable." With control variables, one can operate on a control in the "normal" C++ way:

```
CButton c_Button; // c_Button is an instance variable of the dialog
...
if(c_Button.GetCheck() == BST_CHECKED) ...
```

6.1.3 Dialog Control Management: Continuous Validation

A user interacts with a computer not only by reading data from it, but also by writing data to it. Typically, when users input data to a program, the program has to provide certain capabilities of checking whether the data are valid according to some criteria. The process of checking is called "validation."

There exist two ways of validation: one is "continuous validation," and the other can be called "validation-on-ok," which does not validate the data until the user presses the ok button. They are distinguished by the immediacy of the provided feedback; coded in the style of continuous validation, the program can provide immediate feedback to the user whenever the value of the current input field becomes invalid, instead of deferring it until the user presses the ok button. The benefit of "validation-on-ok" is that it is easier to program; in fact, many introductory textbooks use this style of validation as examples.

To provide better interaction experience to the end user, "continuous validation" is preferred to the simple, "validating-on-ok" style. In fact, product quality software should all be programmed with "continuous validation." Section 6.3 will discuss issues on how to properly program with the style.

So much for the MFC preliminaries. More information on the architectural design of MFC can be found from the article [DiL95]. For information on the detailed design, one can consult the book [SW96].

6.2 Avoiding CWnd::GetDlgItem

MFC supports a style of dialog programming without creating control variables. Instead, one obtains an MFC object for the underlying control through the method GetDlgItem, with a control ID as the argument:

```
CButton * aButton = (CButton *)GetDlgItem(IDC_BUTTON);
if(aButton->GetCheck( ) == BST_CHECKED) ...
```

IDC_BUTTON is the control ID for a button in the dialog, and BST_CHECKED is a constant that represents the state that a button is checked. CWnd::GetDlgItem returns a pointer to CWnd. A downcast to "CButton *" is then applied to the return value. After getting the pointer to the MFC object in the variable aButton, one can send it button-specific messages. For example, the code above sends the message GetCheck to determine whether the button has been checked.

Programming in the GetDlgItem style is all right as long as one programs only simple dialogs, which may, for example, manage only a few controls. In fact, many introductory MFC programming books use GetDlgItem to illustrate how to program (simple) dialogs. Unfortunately, real world dialogs are usually much more complex than that; it is normal for one dialog to have tens of controls in it. Furthermore, sometimes using GetDlgItem can pose a severe maintenance headache.

The problem happens when your program has many GetDlgItem casts and you want to subclass a control. You then will have to find all uses of the control obtained by GetDlgItem and change the casts to the new class. Say, for example, you have created a subclass, CMyButton, of the CButton class, then you will have to find all the GetDlgItem and change

the target type of the casts from "CButton *" to "CMyButton *." It can be time-consuming to make such changes. However, if you use control variables, all you have to do is to change the variable type in the declaration, and all the overloading and inheritance work correctly. Much better. This is how C++ is supposed to be used.

Avoiding CWnd::GetDlgItem can be enforced by FCL as follows:

```
1  CWnd as class("CWnd");
2  CDialog as class("CDialog");
3  DerivedDialog as descendant(CDialog);
4  // GetDlgItem as function("GetDlgItem", CWnd);
5  // CWnd * CWnd::GetDlgItem(int); CWnd * CWnd::GetDlgItem(int, HWND);
6  GetDlgItem as {fct:function(CWnd) | name(fct)="GetDlgItem" and
7            sizeof(param(fct))=1};
8  forall derivedDialog being DerivedDialog holds
9    forall fctCallExp being expClosure(derivedDialog) holds
10       not function(fctCallExp) = GetDlgItem
```

This specification requires that no derived classes of CDialog invoke the method "CWnd * CWnd::GetDlgItem(int)." As shown at line 5, the class CWnd has two overloaded GetDlgItem methods. Thus, had we defined the set GetDlgItem as line 4, the set would have contained two elements, and the negation at line 10 will always yield true regardless whether there are invocations of the method GetDlgItem. The above specification fixes this problem by defining the set GetDlgItem through set comprehension; since we know that the class CWnd defines two GetDlgItem methods with different number of parameters, it suffices to define the set as containing methods with the name "GetDlgItem" and only one parameter. Effectively, this will yield a singleton set with "CWnd * CWnd::GetDlgItem(int)" as its only element.

Note that the equality test at line 10 is a shortcut for:

```
1  forall getDlgItem being GetDlgItem holds
2     function(fctCallExp) = getDlgItem
```

This effectively requires GetDlgItem to be a singleton set.

6.3 Continuous Validation in Dialog

When programming in the style of continuous validation, based on the current values of some other controls, one often wants to enable or disable certain controls or turn them into visible or invisible states. There are two ways to do this: One is to implement the logic *in the code that responds to events* such as button presses, ListBox selections, and so on. That is, put the logic in the event handlers. The other is to put *the control manipulation code in precisely one place in the program.* The "event handler" approach can create code scattered all over the place and thus hard to change and maintain. In the following, we first use an example illustrating why the former approach is a bad idea and then show how to improve it by localizing the code. Finally, we analyze the main characteristics of the localized code and capture them with FCL.

Now, assume we have a dialog with 4 controls: one button c_OK, two edits, c_Text and c_Count, and one check box c_Option, and we want to handle the updating of c_OK and c_Count based on the following requirements:

- c_OK is disabled if c_Text is empty.
- c_OK is disabled if c_Option is selected and c_Count is 0.
- c_Count is enabled if and only if c_Option is checked.

The event handler solution is as follows:

```
void CMyDialog::OnChangeText()
{
     CString s;
     c_Text.GetWindowText(s);
     s.TrimLeft(); // ignore leading space
     if (s.GetLength()==0)
        c_OK.EnableWindow(FALSE);
}

void CMyDialog::OnChangeCount()
{
     CString s;
     c_Count.GetWindowText(s);
     s.TrimLeft(); // ignore leading space
     if (s=="0" && c_Option.GetCheck()==BST_CHECKED)
        c_OK.EnableWindow(FALSE);
}

void CMyDialog::OnChangeOption()
{
     BOOL enable = (c_Option.GetCheck()==BST_CHECKED);
     c_Count.EnableWindow(enable);
     CString s;
     c_Count.GetWindowText(s);
     s.TrimLeft(); // ignore leading space
     if (s=="0" && enable)
        c_OK.EnableWindow(FALSE);
}
```

This code is hard to write correctly, not to mention maintaining it. The disabling of c_OK depends on the state of all other three controls; thus all of the three event handlers above have to check whether c_OK should be disabled. If we want to add another condition to c_OK, then we shall have to revisit all three places. This is error-prone: one may miss updating some of the places or mess up with the computation of the enabling condition. We need a better strategy to deal with this kind of code.

One alternative is to encode each enabling condition into a single compound boolean expression and allow each control to have only a minimum number of "EnableWindow" and "ShowWindow." Normally, at most one instance of "EnableWindow" and at most one instance of "ShowWindow" should suffice. But there can be exceptions: for example, a simple if-statement may have two instances, one in each branch.

Furthermore, all invocations of "EnableWindow" and "ShowWindow" should be localized in one single place. To localize the control updating logic, a dialog can have a method, say, "updateControls," in which all state changes on all controls are computed. The shortcoming of this scheme is that we may end up with some control handlers (for example, those for checkboxes) doing nothing but calling "updateControls."

When the state of one control depends on one or more other controls, the state of the controls affecting it should be directly accessed only at the time the computation is done. Always compute from the first principles, every time; every variable that can affect the state is computed when needed, not at any instant before it is needed, and no references to any boolean variables set magically from some other functions are allowed.

The alternative implementation is as follows:

```
void MyDialog::OnChangeText()
{
     updateControls();
}

void CMyDialog::updateControls()
{
     BOOL enable;

     // c_OK ========================================
     CString s;
     c_Text.GetWindowText(s);
     s.TrimLeft(); // ignore leading spaces

     enable = s.GetLength() != 0 &&
                 (c_Option.GetCheck() == BST_UNCHECKED ||
                  c_Count != "0");

     c_OK.EnableWindow(enable);

     // c_Count =====================================
     enable = c_Option.GetCheck()==BST_CHECKED;
     c_Count.EnableWindow(enable);
     x_Count.EnableWindow(enable); // x_Count is the caption of c_Count
     //==============================================
}
```

Now if we want to add or change a condition to c_OK, all we have to do is to concentrate on the first assignment to the variable enable. That everything is physically centered together makes changes easier. This is a much better approach.

The dialog should conform to three constraints: (1) minimizing the number of invocations of "EnableWindow" and "ShowWindow," (2) always computing the condition directly from the state of the controls, and (3) localizing the calls to one single place. Ideally, we should enforce all three of them. In the following, we shall show a specification of (3) in FCL and then explain why we leave the first two not enforced.

```
1  // Based on the essay ''Dialog Box Control Management'' by Joseph Newcomer
2  // Revision:  all "EnableWindow" and "ShowWindow" of each control must be
3  // in the same routine.
```

```
4
5  windowClasses as descendant(class("CWnd"));
6  Dialogs as descendant(class("CDialog"));
7
8  forall subclassOfDialog being Dialogs holds
9  [
10  EnableWindowOrShowWindow as
11       {exp:  expClosure(subclassOfDialog) |
12       [receiverType as type(receiver(exp));]
13       ((name(exp) = "EnableWindow" or name(exp) = "ShowWindow")
14       and
15       belongsTo(receiverType, windowClasses))
16       }
17  ]
18    forall e1, e2 being EnableWindowOrShowWindow holds
19    (
20      (e1 = e2) or
21      (not refd(receiver(e1)) = refd(receiver(e2)) or
22      unit(e1) = unit(e2) or
23      not (println(e1) and println(e2))
24      )
25    )
```

One special case of constraint (1) is to require that for each control, there be at most one instance for each of the two methods; this effectively forces everybody to program in the idiom embodied in the above improved implementation, that is, do not call the methods until a final enabling condition is completely computed. But sometimes this may be too restrictive; it would be reasonable for one to write code as follows:

```
c_OK.EnableWindow(TRUE);
if (c_Option.GetCheck()==BST_CHECKED &&
    s_Count == "0")
   c_OK.EnableWindow(FALSE);
if (s_Text.GetLength()==0)
   c_OK.EnableWindow(FALSE);
```

Constraint (2) is an example that we would like to enforce but cannot with FCL. It might become feasible if we add to FCL more sophisticated analyses of data dependences such as Aspects [Jac95]. But in the context of this example, we feel that enforcing (3) alone yields the best benefit versus cost ratio. Following it, one is getting on the right track for dealing with the distributed update problem. Once one gets the structure right, chances are much lower for them to mess up with (1) and (2).

6.4 Avoiding CWnd::UpdateData(BOOL) in Dialogs

A dialog is a control that can contain a set of other controls, allows a user to edit the data associated with the controls, and provides validation to the user during the interaction. Designing dialogs involves two aspects: a way to transfer data items back and forth between data members and the controls and a mechanism to validate the data. Figure 6.3 depicts MFC's design for dialogs.

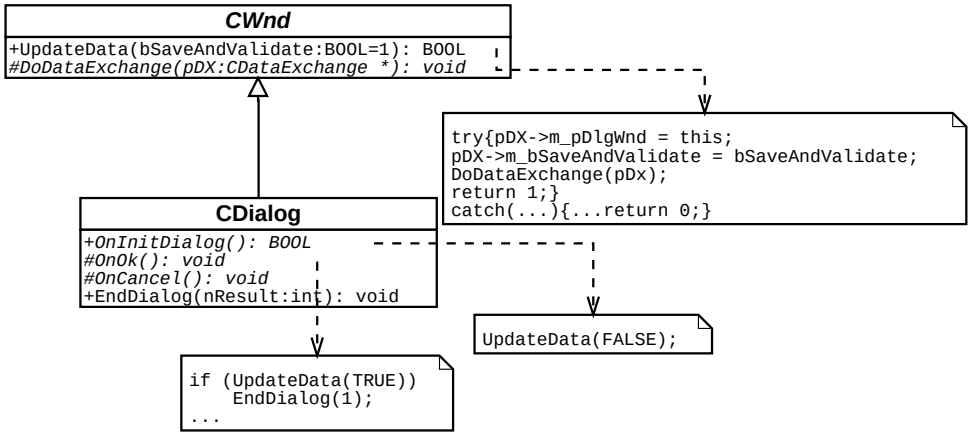

Figure 6.3: DoDataExchange

6.4.1 CWnd::DoDataExchange

The core of the design is a virtual method CWnd::DoDataExchange that is based on a set of so-called "data exchange and data validation routines." Data exchange routines are prefixed with "DDX_," and data validation routines "DDV_." For example, "void DDX_Text(CDataExchange* pDX, int nIDC, CString& value)" is a data exchange routine that transfers a string between the parameter, value, and a control with a control ID nIDC. The parameter pDX is a pointer to an object that carries the context information for the control. The class CDataExchange will be explained later. "void DDV_MinMaxInt(CDataExchange* pDX, int value, int minVal, int maxVal)" is an example of data validation routines: this routine checks that the parameter, value, falls between minVal and maxVal inclusively; otherwise it will pop up a window to inform the user.

A specific subclass of the class CDialog that has only one edit as its child control, say, CModalDialog, can then override DoDataExchange. Its code may look like as follows:

```
void CModalDlg::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);

        DDX_Control(pDX, 1001, c_MyEdit);
        DDX_Text(pDX, 1001, m_MyEdit);
        DDV_MaxChars(pDX, m_MyEdit, 4);
}
```

In this code, DDX_Control is a data exchange routine mapping the control variable c_MyEdit to a Windows control with a control ID 1001. Next, DDX_Text transfers the string between the edit and the member variable m_MyEdit, which is of the CString type. At last, DDV_MaxChars, when pDX->m_bSaveAndValidate is true, validates if the length of the string is less than or equal to 4.

It is up to the programmer to decide which DDX and DDV routines they want to use

in the DoDataExchange method. Particularly, since the capability of both DDV routines and the error handling is quite primitive and inflexible, one may elect not to use the DDV mechanism at all.

6.4.2 DDX and DDV Routines

The implementation of all DDX and DDV routines exploits the helper class CDataExchange shown in Figure 6.4:

- m_bSaveAndValidate: if !m_bSaveAndValidate, then loads the data state into the controls; if m_bSaveAndValidate, then validates and sets the data members from the controls. Validation occurs only when m_bSaveAndValidate is true.
- m_pDlgWnd: The dialog that contains controls. The dialog is needed to get the control from a given control ID.
- PrepareCtrl and PrepareEditCtrl: Remember the current control that is exchanging data. The handle will be used to set the focus if a validation fails. PrepareCtrl is used for nonedit controls and PrepareEditCtrl is used for edit controls.
- Fail: If there is an input error, this routine will be called, bringing up a message box to alert the user. This routine will restore the focus to the last control (the one referenced by PrepareCtrl/PrepareEditCtrl) and throw an exception. This member function may be called from both DDX_ and DDV_ routines.

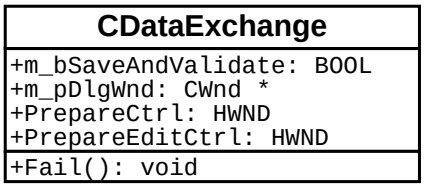

Figure 6.4: Class CDataExchange

6.4.3 CWnd::UpdateData

CWnd::UpdateData, as shown in Figure 6.3, does the initialization and exception handling around the call to CWnd::DoDataExchange. It takes a boolean as its parameter. If the parameter is true, then UpdateData will transfer data from the controls to the member variables and validate them; otherwise, it will transfer the data from member variables to the controls. UpdateData returns 0 if there is an exception thrown by a data validation routine; otherwise, it returns 1.

The class CDialog implements the standard behavior for dialogs. On the initialization of a dialog, an initialization event is generated, and the corresponding event handler OnInitDialog will be invoked. OnInitDialog calls UpdateData with false as the argument to initialize

all the controls with the values of their corresponding member variables. A standard dialog has both an ok button and a cancel button with OnOK and OnCancel as their respective event handlers. Particularly, OnOk calls UpdateData to transfer data from the controls to the member variables. If the transfer succeeds, that is, no exceptions are thrown by any DDV routines, then the dialog will be closed off by calling the EndDialog method.

So much for dialog design.

6.4.4 Why One Should Avoid CWnd::UpdateData

In dialog programming, data validation can be done in two ways: the simple, "validating-on-ok" style and the sophisticated, continuous validation style. MFC fully supports the former with the design presented before. Customizing the design to implement continuous validation is also possible, but requires more work on the part of the programmer.

In practice, many programmers have trouble with implementing continuous validation for several reasons. For one thing, Microsoft does not adequately document the correct way to work with controls; thus it is left up to the programmers to somehow magically infer how to do it correctly. For another, many MFC programmers do not fully understand the design of the dialog, and thus cannot correctly extend it. Even worse, the "Microsoft model" (simple validation) is designed in such a way that one does not have to fully understand the above design before using it. One gets so used to the simple model that they are not prepared well enough to transit to the sophisticated one. For example, one mistake that many make is to call UpdateData(TRUE) to get the current values of the controls. In fact, assigning control variables to controls can eliminate calling UpdateData since one can access the values through the control variables.

Of course, the Microsoft model has its use in certain simple scenarios, provided that one is satisfied with the error handling provided by the default implementation. Sometimes one may need a dialog only to obtain some data from the end user even without validation, then the default model is certainly adequate for the purpose. In either case, however, no UpdateData(TRUE) is needed to be called in any subclasses of the CDialog class.

Therefore, we conclude that one should avoid calling UpdateData(TRUE), and that calling it is a sign of trouble.

6.4.5 "Avoiding CWnd::UpdateData" in FCL

```
1  // Based on Dr. Joseph Newcomer's MFC essay 'Avoiding UpdateData'
2
3  DerivedDialog as descendant(class("CDialog"));
4
5  forall derivedDialog being DerivedDialog holds
6  [
7    doDataExchange as function("DoDataExchange", derivedDialog);
8    callMFCDdv as sizeof(doDataExchange)=1 and
9         exist ddv being exp(doDataExchange) holds
10              regex("DDV_.+", name(ddv));
11    allMeth as function(derivedDialog);
```

```
12   onOk as function("OnOK", derivedDialog);
13 ]
14 (
15   (not callMFCDdv or sizeof(onOk)=0) // callMFCDdv => sizeof(onOk)=0
16   and
17   not exist e being exp(allMeth-onOk) holds
18   (
19        name(e)="UpdateData"
20        and
21        [argument as member(arg(e),0)]
22        (
23          name(argument)="1" or name(argument)="true"
24        )
25   )
26 )
```

A few comments are in order:

- This specification exemplifies how to use the FCL function "function." This function is overloaded, with one version being a constructor that returns a set of functions sharing the given name and the other returning a set of functions contained by a given program unit. Lines 7 and 12 define two singleton sets, 'doDataExchange' and 'onOk,' respectively. They apply the constructor version whereas the one at line 11 applies the non-constructor version.
- The FCL variable 'callMFCDdv,' defined at lines 8 through 10, is of type boolean. This is an example where one can define a variable for a formula, because FCL allows for treating formulas as expressions. The definition also makes use of 'regex,' the regular expression operator of FCL. In this case, it matches any function call whose name starts with "DDV_."
- Note that the 'exp' operator at line 17 is applied to a set of methods instead of an individual method.
- 'argument' at line 21 represents the first argument of expression 'e'; 'arg(e)' returns the sequence of arguments of 'e', and 'member' then returns the one at index 0.
- TRUE is a macro representing '1.' The formula at line 23 checks whether 'argument' is '1' or 'true.'

6.5 Use the Combo Box Controls Correctly

A combo box control can display a list of strings, each of which has an associated data item and an index identifying its position in the list. A user can select a certain item from a combo box by clicking on its corresponding string.

Figure 6.5 depicts a relevant part of the design. Some brief specifications for the methods follow:

- GetCount: Returns the number of items in the combo box.
- GetCurSel: Returns the index of the current selected item.

CComboBox
+GetCount(): int +GetCurSel(): int +SetCurSel(): int +AddString(str:CString &): int +GetLBText(nIndex:int,str:CString &): void +GetItemData(nIndex:int): DWORD +SetItemData(nIndex:int,dwItemData:DWORD): int

CString
+LoadString(nID:UINT): BOOL

Figure 6.5: Class CComboBox

- SetCurSel: Sets the current selected item.
- AddString: Adds a string into the list and returns its index.
- GetLBText: Retrieves the string of the *nIndex*th item and returns it with *str*.
- SetItemData: Sets the data associated with the *nIndex*th item.
- GetItemData: Gets the data associated with the *nIndex*th item.

The IDE for MFC, Microsoft Visual Studio, supports a notion of "resources;" any compilation time constants can be treated as "resources." For instance, both icon images and constant strings can be resources. Each resource belongs to a specific category and is assigned a constant ID. Each type of resource has a special set of APIs to load the real resource. For example, given the ID of a string resource, CString::LoadString effectively initializes a CString object with the string corresponding to the ID.

A tool of the IDE called "resource editor" can help set up the list of strings for a combo box. For example, suppose that one wants to program a combo box with the following color names: *Black*, *Blue*, *Red*, and *Green*. With the resource editor, one can simply type in these strings; at runtime MFC will automatically load the strings into the combo box. The tool saves the programmer from explicitly writing code for loading the strings.

With the support of the resource editor, many programmers have developed some wrong ways of programming combo boxes, ways that can cause problems for future maintenance. Their programs depend on either the item index or the item string, both of which can subject to future change. When such change happens, one is then forced to inspect and adjust existing source code to make sure that the software will still work. Such adjustment can be both tedious and error-prone.

The following subsections will use the color example to show first two typical wrong ways of programming combo boxes and then the correct way. FCL specifications are then written to help reveal the errors.

6.5.1 "Order Sensitive"

One kind of solution is exemplified by the following code:

```
switch(c_ComboBox.GetCurSel())
     {
       case 0: // black
              color = RGB(0, 0, 0);
              break;

       case 1: // blue
              color = RGB(0, 0, 255);
              break;
       ...
     }
```

This solution implicitly associates each index with a certain color, that is, 0 with *Black*, 1 with *Blue*, and so on.

The problem is that it is "order sensitive:" if later on one decides to sort the strings or insert some new colors into the list, then it is highly possible that the index-to-name map will have to change. The only way to make the change is to examine the source code line by line. Since examining source code can be tedious and error prone, a better approach is needed.

6.5.2 "Language Sensitive"

Another kind of solution is as follows:

```
CString s;
int index = c_CComboBox.GetCurSel();

c_CComboBox.GetLBText(index,s);

if(s == CString("Black"))
    {
      color = RGB(0, 0, 0);
    }
else if(s == CString("Blue"))
    {
      color = RGB(0, 0, 255);
    }
```

This solution has the benefit that one does not have to read the source code again if they add or re-order the strings, since the code does not depend on the index for the meaning of the item.

But this code has a "language sensitive" problem. If, for example, one is asked to port this software to German, then one will have to find from the source all the strings for colors and change them to the corresponding German words. This change may be easier to make than the last one, but another solution can make all these changes much easier.

6.5.3 The Right Solution

The key to the right solution is to associate data with each item and use the data, rather than the index or string, to determine which item the user wants to select from the list. In the following, a sample implementation is provided.

First, the string-to-data pairs can be established by the following data structure:

```
typedef struct IDData {
             UINT id;
             DWORD value;
           };

IDData colors [] = {
          IDS_BLACK, RGB(0, 0, 0),
          IDS_BLUE,  RGB(0, 0, 255),
            ...
          0, 0 // end of table

        };
```

Note that the IDSs are for string resources defined by the IDE. This solution has a single point of definition for all values in the combo box. Thus, to add or delete items, one needs to look at and change only one place.

The next step is to initialize the combo box with the above data. This can be done through a method of the subclass of CComboBox, defined as follows:

```
void CIDCombo::load(IDData * data)
   {
    for(int i = 0; data[i].id != 0; i++)
         {
           CString s;
           s.LoadString(data[i].id);
           int index = AddString(s);
           SetItemData(index, data[i].value);
          }
    }
```

where CIDCombo is a subclass of CComboBox.

And make sure to call CIDCombo::load from within the event handler OnInitDialog:

```
BOOL CMyDialog::OnInitDialog ( )
   {
    ...
    c_Colors.load(colors);
    ...
   }
```

One can create the method CIDCombo::GetColor to get the color value of the current selected item:

```
COLORREF CIDCombo::GetColor()
    {
     int sel = GetCurSel();
     if(sel == CB_ERR)
        return RGB(0, 0, 0); // or other suitable default value
     return GetItemData(sel);
    }
```

Another method CIDCombo::Select is needed to select an item:

```
int CIDCombo::Select(DWORD value)
    {
     for(int i = 0; i < CComboBox::GetCount(); i++)
         { /* compare */
          DWORD v = CComboBox::GetItemData(i);
          if(value == v)
             { /* found it */
              CComboBox::SetCurSel(i);
              return i;
             } /* found it */

     CComboBox::SetCurSel(-1);
     return CB_ERR;
    }
```

This ends our presentation of the right solution.

6.5.4 Combo Box Programming in FCL

One has to be selective when writing FCL specifications. Usually, many details can be specified and it is up to the specifier to choose the "appropriate" constraints on the structure. The specifier has to weigh all the candidates and choose only those that are most likely to reveal errors. Sometimes one specifies against specific symptoms with the expectation to catch recurring, common mistakes. Other times one specifies the necessary features that a correct solution must possess, expecting to detect errors of omission.

The following two specifications are targeted at specific symptoms. The first one requires no invocations of the GetCurSel method on any combo box whose return value is compared with something else to make a decision. If otherwise such a code pattern does appear, then the programmer must have assigned some meaning to the offset, which is not desirable. The specification follows:

```
1  // Based on Dr.  Joseph Newcomer's MFC essay 'Combo Box Initialization'
2  // In any derived classes of CDialog, there must not be such patterns:
3  // comparing return value of CComboBox::GetCurSel with constant integers.
4
5  DerivedDialog as descendant(class("CDialog"));
6  defaultCombo as class("CComboBox");
7  DerivedComboBox as descendant(defaultCombo);
8
9  forall derivedDialog being DerivedDialog holds
10   not exists exp being expClosure(derivedDialog) holds
11       (name(exp)="GetCurSel" and
12       belongsTo(type(receiver(exp)), DerivedComboBox) and
13       exist path being downPath(exp) holds
14       exist compExp being path holds
15         (isBranch(compExp) and dep(compExp, exp)))
```

Note the two predicates isBranch and dep: isBranch determines whether the parameter expression is a branch condition, and dep determines whether the value of the first parameter depends on the value of the second.

Similarly, the following specification requires the control flow of the program to not depend on the display strings:

```
1  // In any derived classes of CDialog, there must not be such patterns:
2  // comparing text returned by CComboBox::GetLBText with constant strings.
3
4  DerivedDialog as descendant(class("CDialog"));
5  defaultCombo as class("CComboBox");
6  DerivedComboBox as descendant(defaultCombo);
7
8  forall derivedDialog being DerivedDialog holds
9    not exists exp being expClosure(derivedDialog) holds
10       (name(exp)="GetLBText" and
11       belongsTo(type(receiver(exp)), DerivedComboBox) and
12       exist path being downPath(exp) holds
13       exist compExp being path holds
14         (isBranch(compExp) and dep(compExp, exp)))
```

In contrast to the above two, the following specification constrains the structure of the right solution. It requires all the classes of the combo boxes contained in a dialog to call both the GetItemData method and the SetItemData method, and the dialog to initialize the combo box by calling one of its methods in the OnInitDialog method.

```
1  // In any derived classes of CComboBox, there must be expressions of both
2  // GetItemData and SetItemData
3
4  DerivedDialog as descendant(class("CDialog"));
5  defaultCombo as class("CComboBox");
6  DerivedComboBox as descendant(defaultCombo);
7
8  forall derivedDialog being DerivedDialog holds
9  forall aVar being var(derivedDialog) holds
10  [ varType as type(aVar)]
11  (
12    not belongsTo(varType, DerivedComboBox) or
13    exist onInitDialog being function(derivedDialog) holds
14    (
15      name(onInitDialog) = "OnInitDialog" and
16      exist fctCall being exp(onInitDialog) holds
17      (
18        refd(receiver(fctCall)) = aVar and
19        exist setItemData being exp(function(fctCall)) holds
20          name(setItemData)="SetItemData"
21      )
22    )
23    and
24    exist getItemData being expClosure(varType) holds
25      name(getItemData)="GetItemData"
26  )
```

Even if the implementation of a combo box class satisfies this specification, it is not guaranteed to be correct: it may contain other kinds of errors. What the specification is interested in is to detect the opposite: when it is not satisfied, chances are high that the implementation has an error; FCL helps us catch such errors.

6.6 Constraints on Control Flow

Suppose you have a dialog that contains a text control and want to resize the text control whenever the size of the dialog changes. In MFC, this can be done by subclassing the class CDialog and in the subclass, implementing the message handler OnSize for message WM_SIZE:

```
CMyDialog::OnSize(...)
{
       CDialog::OnSize(...);
       ...
       // c_text is a control in the dialog
       c_text.SetSize(...);
}
```

This implementation is problematic. When running it, you will find that the program crashes in the middle of an ASSERT statement, which requires that the control object c_text must have a control associated with it. If you trace the program, you will find that the ASSERT statement comes from the SetSize call.

The problem is because of a mismatch between MFC and the underlying Windows. A dialog can receive the WM_SIZE message in two states: (1) where the dialog is created but some of its child windows have not been initialized yet, and (2) where both the dialog and all of its child windows are properly initialized. What has happened is that at state (1), a WM_SIZE is generated and dispatched to the dialog, and the corresponding event handler, OnSize, is called. But since the c_text control has not been properly initialized, the call to SetSize will cause the assertion failure.

The solution is to distinguish the two states of the dialog. One can set up a condition that indicates whether the controls are ready and guard all the control operations by that condition. Usually a member of int or boolean in the dialog class would suffice. It should be initialized as false in the constructors and set to true at the end of the OnInitDialog method, since by then all the controls of the dialog should have been properly initialized.

Thus we can write a specification to detect the key features of this solution:

```
1  // Assertion failure caused by the mismatch between Windows and MFC
2
3  PredefinedDialog as [class("CCommonDialog"), class("CPropertyPage")];
4  DerivedDialog as subclass(class("CDialog"))-PredefinedDialog;
5  cWndSet as descendant(class("CWnd"));
6
7  forall derivedDialog being DerivedDialog holds
8  [
9    OnInitDialog as { fct:function(derivedDialog)| name(fct)="OnInitDialog"};
10
11   Cstor as { fct:function(derivedDialog)| name(fct)=name(derivedDialog) };
12
13   OnSize as { fct:function(derivedDialog) | name(fct)="OnSize" };
14
15   expOfOnSize as union( { fct:OnSize | true | exp(fct) } );
16
17   ctrlExp as { exp :expOfOnSize |
18             [receiver as receiver(exp); type as type(receiver);]
19             (belongsTo(type, cWndSet))
20             };
21
22   booleanVar as { v:  var(derivedDialog) |
23        [type as type(v)]
24        (name(type)="boolean" or name(type)="int")
25     };
26 ]
27   (
28     sizeof(OnSize)=0
29     or
```

```
30     exist aBool being booleanVar holds
31     (
32       // assigned in both OnInitDialog and Constructors
33       forall fct being OnInitDialog+Cstor holds
34         exist expr being exp(fct) holds
35         (var(expr) = aBool and name(top(expr))="=")
36       and
37       forall fct being OnSize holds // control operations guarded
38       [controlledExp as union({ expr:  exp(fct) |
39            var(expr) = aBool and isBranch(parent(expr)) |
40            union(downPath(parent(expr)))
41         });
42       ]
43       subset(ctrlExp, controlledExp)
44     )
45   )
```

A brief explanation for the specification follows. The main part runs from line 28 to line 44. Line 28 says that if the checked program does not define the OnSize handler, then the specification is satisfied; otherwise, proceed. The rest of the specification, lines 30 through 44, says that both OnInitDialog and the constructors must have at least one assignment statement that involves a member of boolean or int and that all the control operations in the OnSize method must be guarded by a predicate expression that involves that data member.

Note that the specification requires not only that there be member variables of appropriate types, but also that there exist at least one such variable being changed in both the constructors and the OnInitDialog method. We should know some subtleties with this specification. Let us explore two alternative specifications.

The simplest one would be to check only that the class has the required condition variable. In general, any primitive types can be used as the type of the variable. In practice, however, normally only int and boolean would be considered; even it is acceptable by the compiler, it would be rather unusual to use a char to represent a boolean. Even such a simple specification can be useful: when using dialogs, often one does not define member variables at all, and a miss of such a variable will indeed be an error. Of course, if one does define other member variables in the dialog class, this specification risks treating member variables for other purposes as the condition variable and thus generates false negatives. Clearly, the situation calls for engineering judgment for the right decision.

The other alternative is to check only that the member variable is changed in both the constructors and the OnInitDialog method. In fact, compared with the last one, it provides a better chance of detecting the error of missing the definition of the condition variable. Since the OnInitDialog method is only for initializing the dialog, application logic that has nothing to do with GUI normally would not get into this method. But the condition variable should be used here. Therefore, it is less likely to mistakenly treat variables for other purposes as the condition variable; thus a violation of the specification has a high chance of revealing a genuine error.

One concern for this alternative goes like this: "what if, instead of changing the variable directly by itself, OnInitDialog calls another method that indirectly changes the variable?" This is certainly a valid concern; if that does happen, one can use the expClosure operator of FCL, which returns the set of expressions that are statically reachable from a certain method. On the other hand, for this example this is unlikely; thus the specification should remain as it is. Clearly, again, engineering judgement is needed here.

However, the two alternatives fail to detect an error where one does define a condition variable but does not guard all the control operations with it. Our specification considers this. So much for the two alternatives.

FCL specifications must be sufficiently strong to detect as many errors as possible, but must not be so strong as to report false positives (spurious errors). One example is the definition of the set controlledExp, from line 38 to line 41, representing the set of guarded control operations (to be more precise: all the control operations that execute after the first evaluation of the condition expression). It would be too restrictive to define a guarded control operation as one guarded by a condition expression consisting of only the boolean itself, since it is possible for one to define the meaning of the boolean as "not initialized" rather than "initialized." In that case, the guard would then have to be the negation of the boolean. The specification has been relaxed to cover both cases. On the other hand, it would be unconceivable to implement the logic in idioms other than these two. Hence the current specification.

Strictly speaking, the current specification is not sufficient either. It assumes that code for all control operations appears in the OnSize handler. It may be so for most cases, but it is also possible for one to create another method to separate the code for "updating controls," then this specification would fail to check that the control operations there are also properly guarded. The aforementioned call graph operator 'expClosure' might come to the rescue in this case.

6.7 Optional Features

Optional features are logical functionalities provided by the framework but not mandatory for all applications. In general, programmers may have two types of problems with optional features: how to find them and how to correctly use them. FCL can help with the latter type of problem.

To apply FCL to a certain optional feature, one needs first to know whether the feature is indeed used. With that information, FCL can then check whether the feature has been correctly used. Logically, this is a pattern of the form $F_1 => F_2$, where F_1 is a formula indicating that the feature is used, and F_2 the constraints to which the solution has to conform.

There are two possible ways of getting F_1: One is to ask the programmer to provide the list of features that he thinks his application is using, and the other is to find some code patterns whose appearance can be used to infer safely that the feature is indeed used. Since the latter method is less demanding of information, normally it should be preferred; however, not all features possess the necessary code patterns for FCL to leverage; thus sometimes user input may be needed as well. The following two examples illustrate the respective types of optional features.

6.7.1 Enabling/Disabling ESC and RETURN Keys

It is common in GUI programming to provide multiple ways of issuing a command, for instance, through a button, a menu item, a hot key, and so forth. In particular, a dialog can be terminated in three ways: through the ok and cancel buttons, through the close icon provided as part of the Microsoft user interface standard, and through the ESC and ENTER keys.

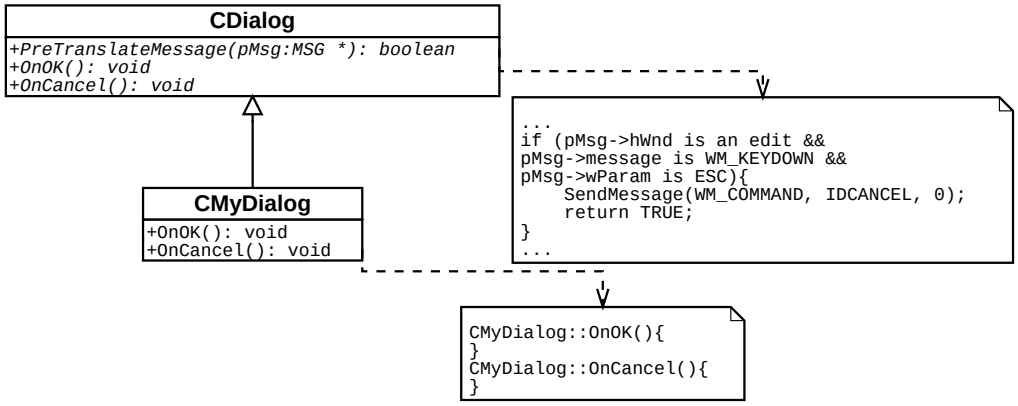

Figure 6.6: Disabling ESC and RETURN Keys in a Dialog

Figure 6.6 depicts the relevant design. The OnOK and OnCancel methods are the corresponding event handlers for pressing the ok and cancel buttons. PreTranslateMessage is a virtual method that can be overridden to pre-process a message; only when the method returns true will the message be further passed on to the message dispatch mechanism. By default, CDialog::PreTranslateMessage intercepts key presses such as ESC and ENTER, interprets them as requests to terminate the dialog, and invokes the corresponding event handlers.

At least two solutions can prevent the ESC key and the ENTER key from terminating a dialog, as shown in Figure 6.6. One solution is to (1) define two empty event handlers for the ok and cancel buttons, which are provided as default by the GUI editor, and (2) remove the two buttons from the dialog. This works because of polymorphism: both OnOk and

OnCancel are virtual methods; thus when ESC and ENTER are pressed, their corresponding empty versions of handlers will be executed, which effectively disables them.

The other solution is through overriding PreTranslateMessage in the CMyDialog subclass and checking the message represented by the pMsg parameter. If the message is an ESC key or an ENTER key, then the method returns false, and thus effectively consumes the message without any effect. The shortcoming of this solution would be that it distributes the termination logic into two places, which might cause a potential maintenance problem in the future. Thus the first solution is preferred.

It is easy to write an FCL specification to detect the presence of the two empty handlers; thus we omit it.

6.7.2 Enabling Tooltips

Tooltips is an example with a code pattern indicating that the tooltips feature is being used. MFC Internally keeps track of when the mouse pointer enters or leaves the boundary of a control. If the pointer stays within the control for over a certain amount of time, a notification message will be generated and sent to the dialog that contains the control. To respond to the notification, the dialog has to define its corresponding message handler named "OnToolTipNotify." Inside that method, one can prepare the tooltip text to be displayed.

But to enable the tooltips feature for the dialog, one also has to call the EnableToolTips method in the OnInitDialog method. This is a programming obligation that many often forget. The following FCL specification can be used to detect this error of omission:

```
1  DerivedDialog as subclass(class("CDialog"));
2
3  forall derivedDialog being DerivedDialog holds
4  [
5    onToolTipNotify as { fct:function(derivedDialog) |
6         name(fct)="OnToolTipNotify"
7         };
8
9    onInitDialog as { fct:function(derivedDialog) |
10         name(fct)="OnInitDialog"
11         };
12 ]
13 (
14   sizeof(onToolTipNotify)=0
15   or
16   sizeof(onInitDialog) = 1
17   and
18   forall onInitDialog being onInitDialog holds
19     exist enableToolTips being exp(onInitDialog) holds
20     (
21       name(enableToolTips)="EnableToolTips"
22       and sizeof(arg(enableToolTips))=1
23       and
24       [theOnlyArg as member(arg(enableToolTips),0)]
25       (name(theOnlyArg)="true" or name(theOnlyArg)="1")
26     )
27 )
```

In this example, the appearance of the OnToolTipNotify method is used to indicate that the tooltip feature has been used.

6.8 The Windows Creation Methods

The class CWnd is the base class of all the MFC classes 'wrapping' Windows controls. It defines three public methods for window creation, as shown in Figure 6.7. Among the three methods, CreateEx_1[1] is the core. It calls the Windows API CreateWindowEx to create the control object. It also implements the crucial mapping between the operating system controls and C++ wrapper objects so that once the mapping is established, all messages for the controls can be routed to the message handlers of the MFC subclasses rather than the default message processing routines provided by the Windows operating system. Both CWnd::Create and Cwnd::CreateEx_2 eventually call CWnd::CreateEx_1.

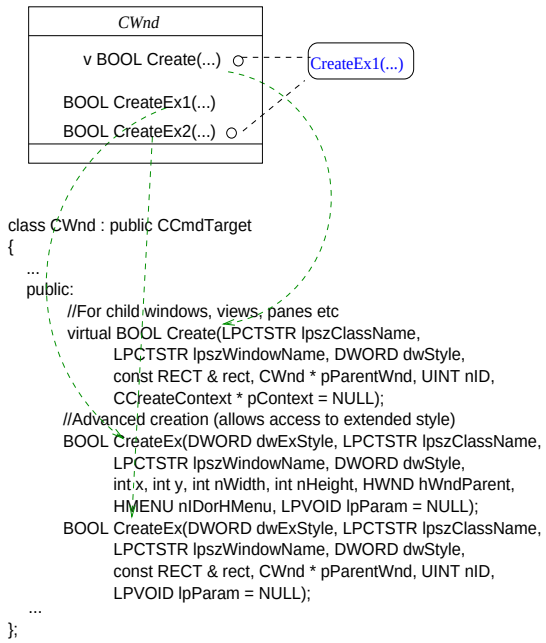

Figure 6.7: CWnd::Create and the two overloaded CWnd::CreateEx

If a programmer subclasses the class CWnd, then each subclass must define at least one public method *directly or indirectly* calling one of the three methods: CWnd::Create, CWnd::CreateEx_1, or CWnd::CreateEx_2. Furthermore, outside of the subclasses there must exist invocations of at least one of the methods.

These constraints can be specified using FCL as follows:

```
1 // for child windows, views, panes etc
2 def create = CWnd::Create(LPCTSTR,
3 LPCTSTR, DWORD,
```

[1] The first *CreateEx* in Figure 6.7. We use subscripts to distinguish between the two *CreateEx*.

```
4  const RECT&,
5  CWnd*, UINT,
6  CCreateContext*);
7
8  // advanced creation (allows access to extended styles)
9  def create1 = CWnd::CreateEx(DWORD, LPCTSTR,
10  LPCTSTR, DWORD,
11  int, int, int, int,
12  HWND, HMENU, LPVOID);
13
14  def create2 = CWnd::CreateEx(DWORD, LPCTSTR,
15  LPCTSTR, DWORD,
16  const RECT&,
17  CWnd*, UINT,
18  LPVOID);
19
20  allClasses as class(global);
21  newWindows as subclass(class("CWnd"));
22
23  not sizeof(newWindows) > 0 or
24  forall class being newWindows holds
25  [
26  creationMethods as {m:  function(class) |
27       exist exp:  expClosure(m) holds
28       [fct as function(exp);]
29       (fct=create1 or fct=create2 or fct=create)
30       };
31  ]
32  (
33  sizeof(creationMethods) > 0 and
34  exist anotherClass being allClasses - newWindows holds
35    exist exp being expClosure(anotherClass) holds
36         [fct as function(exp);]
37         belongsTo(fct, creationMethods)
38  )
```

The first three statements of the specification are shorthands for the long method prototypes.

6.9 Overriding CDocManager::DoPromptFileName

Figure 6.8 depicts the most important classes of MFC's MDI (Multiple Document Interface) architecture (the MFC variant of the MVC architecture [KP88]). The singleton class CWinApp hooks up all aspects of an MFC-based application. The class runs the event looping logic; GUI messages due to user interaction first arrive at and are dispatched from this class. The class CDocManager manages and coordinates the classes that implement documents. Particularly, each application contains a document manager to manage both the types of documents that it supports and the documents currently opened by the application.

MDI standardizes the looks and feel of applications. In particular, each application can have two menu items: "File/Open ..." and "File/Save as ...". If either item is selected, the standard behavior is to pop up a file dialog, allowing users to choose from a list of files. It turns out that the virtual method CDocManager::DoPromptFileName(..., int lFlags, ...) is responsible for popping up the dialog and displaying files according to certain criteria, which are set up through the parameter lFlags. lFlags specifies the file filtering patterns using a

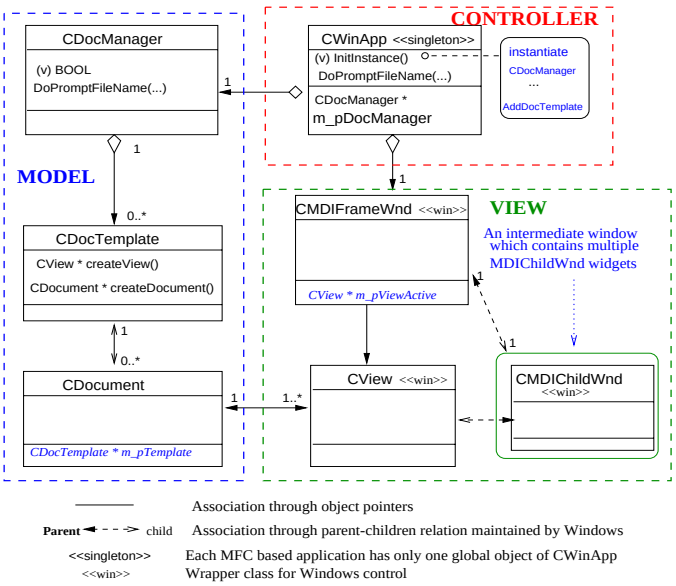

Figure 6.8: Class diagram for the MDI architecture

combination of bit patterns. The sequence diagrams of Figure 6.9 depict how the method is invoked in the two scenarios respectively.

Now, suppose that a framework user is asked to change the default file filtering pattern for his specific application. In order to do so, he must:

1. Subclass the class CDocManager.
2. Override the DoPromptFileName method. The override should first change lFlags and then call CDocManager::DoPromptFileName.
3. Subclass the class CWinApp.
4. Override the CMyWinApp::InitInstance() method. The override should create an object of CMyDocManager on the heap and assign it to the instance variable m_pDocManager before call the AddDocTemplate method.

Figure 6.10 illustrates these constraints, where the class CMyDocManger is the subclass of CDocManager, and CMyWinApp is the subclass of CWinApp.

The constraints can be specified with FCL as follows:

```
1  CDocManager as class("CDocManager");
2  CMyDocManager as subclass(CDocManager);
3  DoPromptFileNameOfDocManager as function("DoPromptFileName",CDocManager);
4  sizeof(CMyDocManager)=1 or sizeof(CMyDocManager)=0;
5  forall docManager being CMyDocManager holds
6  [ DoPromptFileName as function("DoPromptFileName",docManager); ]
7  (
8  not sizeof(DoPromptFileName)=1 or
9  exist e being exp(DoPromptFileName) holds
10  (
11       DoPromptFileNameOfDocManager=function(e) and
12       [
13       lFlags as member(param(DoPromptFileName),3);
```

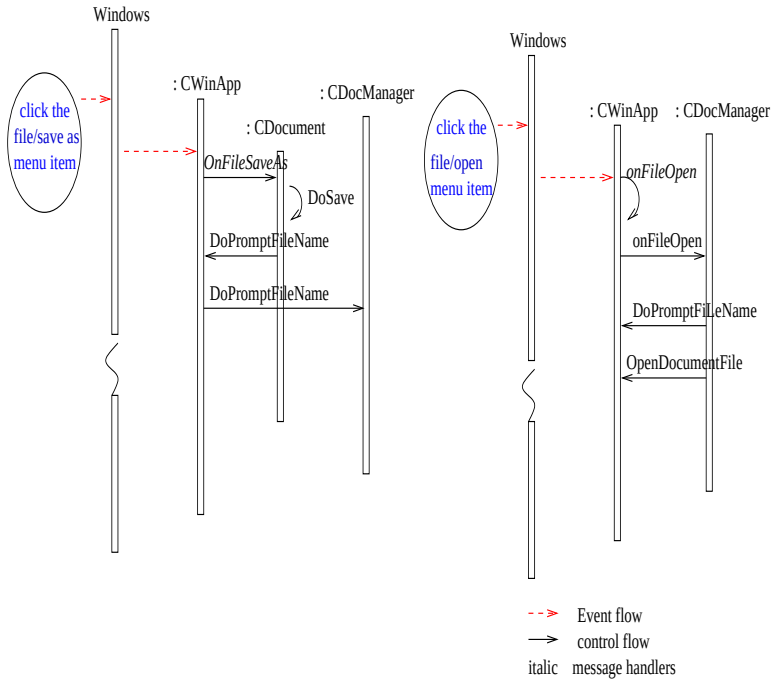

Figure 6.9: How is DoPromptFileName invoked after choosing the "File/Open ..." or "File/Save as ..." menu items

```
14        argFlags as member(arg(e),3);
15        ]
16        dep(argFlags, lFlags);
17  )
18  )
19
20  CWinApp as class("CWinApp");
21  CMyWinApp as subclass(CWinApp);
22  sizeof(CMyWindApp)=0 or sizeof(CMyWinApp) = 1;
23  sizeof(CMyDocManager)=0 or
24  sizeof(CMyWinApp)=1 and
25  forall myWinApp being CMyWinApp holds
26  [ InitInstance as function("InitInstance", myWinApp); ]
27  (
28        sizeof(InitInstance)=1 and
29        exist e being exp(InitInstance) holds
30        [type as refd(member(arg(e),0));]
31          (name(e)="new" and type = CMyDocManager)
32  )
```

This example shows that sometimes a variation point can involve more than one point of a framework. In particular, the variation point may be part of the behavior of two methods. To elaborate it, one has to understand the surrounding context, that is, how these two methods work in general, how the variation point works within them, and how to hook the variant back into the framework. This is not a trivial task.

Furthermore, the current design of DoPromptFileName seems inadequate to support all the customization that users may want to perform. It could be redesigned as a template method design pattern [GHJV94] so that one can do either one of the following:

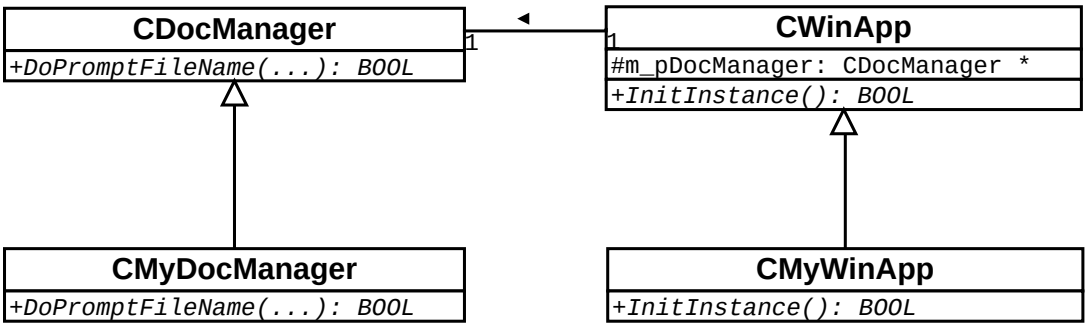

Figure 6.10: Overriding the method DoPromptFileName

- replacing the default dialog used to display files;
- changing lFlags;
- deciding whether the file filter pattern "*.*" is needed in a file dialog.

This example illustrates that frameworks may evolve even after they are used in production.

6.10 Enforcing Naming Conventions

The function regex has been added to FCL to express regular expression patterns on character strings; thus FCL can also be used to enforce certain naming conventions. For example, in MFC programming, one may require that the names of control variables be prefixed with "c_" instead of "m_" and use "m_" only for value variables. Clearly this constraint can be expressed with regular expression patterns.

6.11 Discussion

This section discusses the nature of the previous problems and put them into perspectives:

1. Lack of knowledge on the design of the framework

   Learning to use a framework is not an easy task, and the devil is in the details. The challenge stems both from the complexity of software in general and from the difficulty of communication. This challenge is for both the framework developers and the framework users. As exemplified by the examples in sections 6.4, 6.5, 6.6, 6.7.1, and 6.9, documentation of the detailed designs can get fairly verbose. Framework developers face the task of effectively documenting and communicating the design to the users. On the other hand, to avoid the problems, users must know enough about the details. This is a ramification of the essential difficulty of software.

   The difficulty can be informally appreciated by counting the number of classes and methods each design involves. Although this is not an objective means of measuring design complexity, it seems sufficient to allow us to conclude that most of the designs

are not as trivial as creating an object or calling an API and that the complexity of the design is one of the main contributors to the problem of using frameworks.

Early reuse efforts focus mostly on libraries and generally do not pay off as much as expected. The recent trend is to reuse architectures as a whole, which essentially provide the contexts for reuse. As Johnson [Joh97] points out, "although a good component library is a crucial companion to a framework, the essence of a framework is not the component library, but the model of interaction and control flow among its objects." Programmers have to learn how to live within the provided architectural contexts. That is, to learn the model of interaction and control flow instead of individual functions or classes. Experience has indicated that this is not a trivial matter. Automatically detecting errors is needed to support this kind of reuse.

Another observation is that some variation points are "open." It seems challenging for a user to elaborate on an "open" variation point since to do that, they would have to not only understand the design of the relevant part of the framework, but also make design decisions to fill in the open parts.

Also note that some constraints originate not from exploiting the design to achieve certain functionalities, but from avoiding problems caused by the design. Handling the OnSize message is such an example. We call such a design a "negative design." This does not necessarily mean that overall the design is poor, although they can be triggers for revising the design.

Can we design better to solve the problem? The answer is both yes and no. In general, it is always possible to improve a particular design for concerns like usability. But we suspect that due to the nature of reusing frameworks, the essential difficulty cannot be removed simply by a better design. We just have to live with it.

2. Enforcing programming disciplines

   Sometimes one may want to enforce important programming disciplines for both modifiability and maintainability. Sections 6.5 and 6.3 provide two relevant examples.

3. Forgetting programming obligations

   At least two types of programming obligations are easy to forget. One is for the override to call an appropriate version from the base class, the other is what we call "distributed obligations," where to fulfill a logical function, one has to change more than one spot of the source code. Section 6.7.2 is an example of "distributed obligations."

4. Evolving Frameworks

   It is rarely possible to deliver a high quality framework once and for all; instead, frameworks are more likely to evolve as they are used and feedbacks are gathered from the users. Likewise, the intended ways of using a framework may evolve as well.

Some constraints are related to software evolution; both frameworks and applications can evolve. One example for framework evolution is the DoPromptFileName method. Another is the GetDlgItem method; initially, it is designed as the main way of retrieving the object for a control. But this turns out not to be the right way. FCL can be used to prevent any further use of the method.

Factors that a specifier needs to consider when applying FCL include:

1. Choosing between symptom-oriented and feature-oriented strategies;
2. Avoiding false positives and false negatives;
3. Applying engineering judgments when writing FCL specifications;
4. Knowing the limitation of FCL: some semantics are useful but cannot be captured in FCL.

Chapter 7

Case Study 3: Enforcing General Design Rules

This chapter presents how to specify the law of Demeter [LH89] and Meyers' C++ guidelines [Mey92b, Mey96] in FCL. Our purpose is to illustrate that the applicable scope of FCL is not confined to frameworks; it can also be used to enforce existing generic design principles.

FCL specification for a version of the "Law of Demeter" is written. It is then tested on two small examples. One is taken from the paper: "Assuring Good Style for Object-Oriented Programs" [LH89]. The other is taken from Martin Fowler's refactoring book [F+99]. The specification is also tested against the corrected versions of the examples. In all cases, FCL can respond correctly, either pointing out the place of the violations or reporting that there are no errors.

This chapter also analyzes a subset of C++ programming rules. The goal is to justify our belief that expressions are indispensable for any such tools as FCL.

7.1 Introduction to the "Law of Demeter"

The "Law of Demeter" expresses the general principles of software design, such as information hiding and low coupling and high cohesion, in an easy-to-follow form. The basic idea is to reduce as much as possible the amount of dependency that a method puts on its environment. Simply put it, if one finds oneself making use of more than one level of indirection in code, it is mostly likely that one is violating the "Law of Demeter." It is because of this that some people define the law by simply saying: "Don't use more than one dot." This simplification is not a hundred percent accurate, but it captures the main idea. Figure 7.1 depicts an example that violates the law; the implementation of the statements method of the Customer class uses the class Movie through the Rental class, which should have been avoided.

Even though its name suggests otherwise, it is more accurate to consider the "Law of

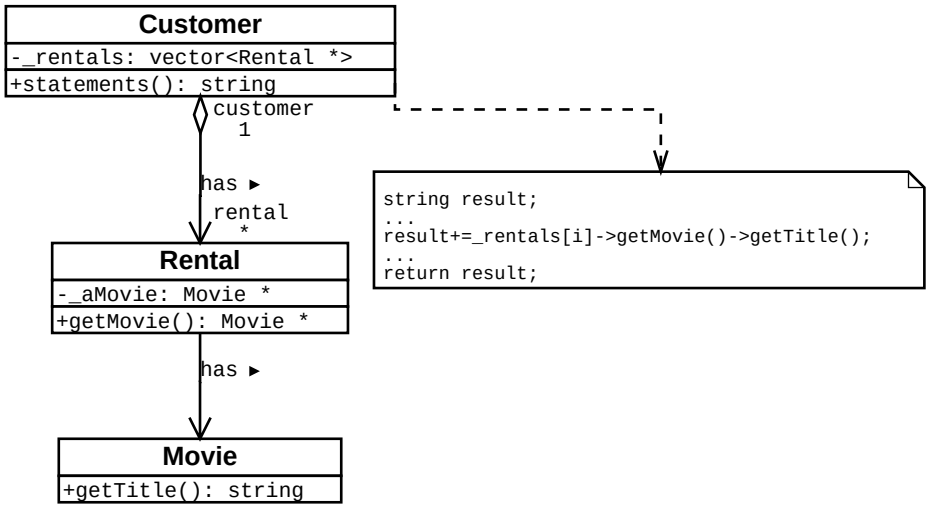

Figure 7.1: Video Store

Demeter" a heuristic or guideline; in practice, certain violations of the law are considered acceptable.

Therefore, the general design principles are still the ultimate standards for judging the quality of a specific design. In a sense, the "Law of Demeter" is only a "surface" version of the general principles. Yet, because the law is phrased in terms of programming constructs, it is possible to enforce it at compile-time with automated tools.

7.2 The Class Version of the "Law of Demeter"

Several versions of the law exist for different purposes. For example, the object version states the law in terms of run-time objects, and thus provides us the best conceptual model to follow. But it is difficult to enforce it with tools. In contrast, the class version phrases the law in terms of compile-time artifacts, and thus makes it possible to programmatically check any violation of the law. Since our goal here is to demonstrate how FCL can be used to encode the law, we choose the class version.

To present the class version of the law, we need to define several terms first:

Client Method: A method can be either an instance method or a static method. If the implementation of a method M has at least one expression either of the form "o.m(...)", where the static type of o is the class C, or "C.m(...)", where m is a static method of the class C, then M is the client of the method m of the class C; M is also the client of the class C itself.

Supplier Class: A supplier class to a method M is the class whose methods are called by M. Thus if the method M is the client of the class C, then C is M's supplier class.

Preferred Supplier Class: A class C is a preferred supplier class to a method M if C is a supplier class to M and one of the following conditions holds:

- C is used as a part of the type of an instance variable of C_M, where C_M is the class to which M belongs.
- C is used as a part of the type of a formal parameter of M.
- C is used as a part of the type of a local variable of M.
- C is used as a part of the type of a global variable referenced by M.

Languages like C++ allows indirect types, i.e., pointer or reference types, array types, and container types generated by instantiating templates of STL. In the case of indirect types, we consider the types that are referenced as suppliers. In the case of array and containers, we consider the element types as suppliers. That is why in the above conditions, C is required to be "a part of the type of" the respective variables.

With the definitions of these terms, the "Law of Demeter" can be phrased as follows:

Every supplier class to a method must be a preferred supplier class.

7.3 The "Law of Demeter" in FCL

```
AllClasses as [class("Movie"), class("Rental"), class("Customer")];
forall C: AllClasses holds
[
  // classes thru instance variables
  classesByInstance as { v:  var(C) |
       [varType as type(v);]
       isClass(ptdTypeStar(varType)) |
       Class(ptdTypeStar(type(v)))
  } ;
   // types thru templates instantiations
   generated as { v:  var(C) |
        [varType as type(v);]
        isGenerated(ptdTypeStar(varType)) |
        Generated(ptdTypeStar(type(v)))
   } ;
   templArgs as union({ t:  generated | true |exp(t)} )
   classesReferredByCollection as { expr:  templArgs |
        isClass(ptdTypeStar(refd(expr))) |
        Class(ptdTypeStar(refd(expr)))
   }
 ]
 forall M:function(C) holds
 [
   classesByParam as { v:  param(M) |
        [varType as type(v);]
        isClass(ptdTypeStar(varType))
        |
        Class(ptdTypeStar(type(v)))
   } ;
   preferredSuppliers as classesByInstance + generated + classesByParam +
        classesReferredByCollection + [C ];

 ]
 forall E: exp(M) holds
 [
        Receiver as receiver(E);
```

```
37            referredVar as Var(refd(Receiver));
38            typeOfVar as type(referredVar);
39      ]
40      ( // The receiver is a var:  a global variable or a local variable
41        (isClass(typeOfVar) or isGenerated(typeOfVar))
42        and
43        (
44          first(unit(referredVar))=global
45          or
46          unit(referredVar)=unit(E)
47        )
48        // The receiver is an expression
49        or
50        ( [
51          typeOfReceiver as type(Receiver);
52          basicType as ptdTypeStar(typeOfReceiver);
53          ]
54          (
55          not isDefined(basicType) or belongsTo(basicType, preferredSuppliers)
56          )
57        )
58        or not println(E)
59      )
```

The overall structure of the specification consists of three nested universal quantifications, starting at lines 2, 22, and 34, respectively. Rephrased in English, they mean:

> For each class C, of the set of classes that need to be checked, for each method M, of C's methods, for each expression E, of the expressions that belong to the method M, E must obey the Law of Demeter.

The block from line 3 to 21 defines local variables for the sub-formula of line 2; that is, these variables will be visible with the formula starting at line 22. The local variables define the sets of supplier classes that are generated due to the class C; specifically, classesByInstance defines the set of supplier classes that are introduced through the instance variables of the class C. generated represents the sets of template instantiations. And classesByCollection is the sets of supplier classes that are the argument to the template instantiations.

Together, lines 5 to 9 form a set comprehension that defines classesByInstance. var is a function that returns the set of instance variables of the class C. type is a function that returns the type of a variable. ptdTypeStar is a function that returns the "basic" type of a type; if the type is a pointer to another non-pointer type, the function will return the latter as its basic type. isClass return true if its argument is a class. Class is a cast function that casts a type to class.

Lines 11 to 15 is another set comprehension that defines generated. isGenerated is a predicate that returns true if and only if its argument type is a generated type. Generated is a casting function that can cast a type to a generated type.

templArgs is a temporary variable that represents the set of types that are used as arguments to the generated types. A generated type is an expression. The function exp returns the set of the sub-expressions of its argument. For example, if the argument to exp is a generated type of the form "L1<L2<L3 *>, L4>," then the result of applying exp to

it should be a set that contains "L2<L3 *>", "L3 *", and "L4". The operation union then flattens all the sets to form a new one.

With templArgs, classByCollection is then defined as the set of supplier classes that are introduced through template instantiations. Note that refd is a function that returns the object that a name reference refers to.

Lines 23 to 33 defines local variables for the second forall formula. It first defines classesByParam, which represents the set of supplier classes introduced through the parameters of the method M. Then it defines the preferred classes of the method M as the union of relative sets, as shown in lines 30 and 31.

With all these variables defined, we can proceed to check each expression E of the method M (lines 34 through 59). For each expression E, Receiver is defined as its receiver expression. If, however, E is not a message sending expression, Receiver will be "undefined." In this case, the whole formula eventually will be true. This is because both typeOfReceiver and basicType will be "undefined" (line 51 and line 52),and thus isDefined at line 55 will yield false, which in turn, makes the whole sub-formula at line 55 true. This reflects our intention; In this version of the law, we are only interested in checking nested message sending.

There can be three cases for the receiver expression. It can be a reference to a variable, which can be either a local one or a global one. It can also be a compound expression. If Receiver is a reference to a variable, then referredVar will be the variable that Receiver refers to and typeOfVar the type of the variable. According to the law, in order for the message sending expression to be valid, referredVar has to be either a global variable or a local variable. The locality of referredVar is checked using the unit function, as show in lines 44 and 46, respectively.

If, however, the Receiver is a compound expression, then we have to check whether its type is one of the preferred supplier classes. This is reflected in the FCL specification at lines 50 through 57.

Finally, if E does not satisfy any of the above conditions, then it is not a valid expression in terms of the law. And its content will be printed out (line 58).

7.3.1 Discussion

Note that the calculation of the set of classes involved through parameters is a bit simplified; parameter types can be generated types, and thus a complete specification would need some extra specification similar to lines 4 through 20. This is all right for our examples since they do not use generated types in the declaration of parameters, but to use this specification in a more general context, we would have to fix this problem.

This example also makes extensive use of the type-related features of FCL. In particular, note that FCL treats generated types as expressions, as shown by line 16.

Note that there is a major flaw in the above treatment of the law: if, instead of accessing the movie through array indexing as the example does, one defines a local variable that is a pointer to the class Movie, and accesses the movie through the local variable, then FCL would not be able to detect this violation of the law. Thus a false negative is generated. A remedy to this problem would be to add to FCL some sort of pointer analyses, reporting errors when a local variable obtains some value that lives beyond the current methods. Pointer analyses, however, are known to be conservative by nature [Ryd03], therefore cannot completely eliminate all such false negatives. This example illustrates the kind of tradeoff which one has to make when using static analyses like FCL.

7.4 Specifying a Subset of Meyers' C++ Rules in FCL

In an article of Dr. Dobb's journal [MK97], Scott Meyers and Martin Klaus reports an evaluation of 5 C++ compilers and 8 static analyzers, including CCEL [MDR93], on their ability of detecting anomalies from C++ programs. The benchmark consists of 36 rules, 34 of which are taken from Meyers' two books: "Effective C++" [Mey92b] and "More Effective C++" [Mey96].

These tools take different approaches on expressing the constraints; some, like the compilers, hard-code the rules, some define special languages for specifying constraints, and still some pre-define a library of common rules while provide the ability for users to define their own custom rules. One finding reported by the article is that the specification languages must be designed with care, and that declarative ones are easier to write and understand than imperative ones.

CCEL fails 17 of the 36 rules. Upon closer scrutiny, we find that most of them involve expressions in one way or another. This is not surprising since CCEL stops at the level of functions and methods: it does not support expressions. This reinforces our belief that expressions are an indispensable part of such tools, and we are glad that FCL has taken them into account right from the beginning.

In the following, we study the 17 rules which CCEL fails to handle. Each rule starts with a number, a letter, and another number, following by a short description. The first number is the number of the rule in the original list [MK97]; letter M represents the book "More Effective C++" and E "Effective C++." The number that follows the letter is the number of the corresponding item in the books. Being lazy, we only rephrase the rules in English; it should be evident that they can be specified in FCL.

1. 2 M 2 Use new-style casts instead of C-style casts

   This can be rephrased as "there are no C-style cast expressions in C++ programs," and thus is checkable in FCL.

2. 3 M 3 Don't treat [a pointer to] Derived[] as [a pointer to] Base[].

   For any argument to any function call, if its type is an array, and the element type of the array is a subtype of the element type of the corresponding parameter of the function, then it violates this rule.

3. 4 E 5 Use the same form for calls to new and delete. (In general, this calls for dynamic analysis, but static analysis can catch some special cases, e.g., calls to new in ctors and to delete in dtors.)

   If there is a new expression in a ctor, then there must exist a delete in a dtor; if there is a new[] expression in a ctor, then there must exist a delete[] in a dtor.

4. 5 E 6 When the result of a new expression in a ctor is stored in a class member, make sure delete is called on the member in the dtors.

   For any pointer data member, if it is initialized by a new expression in any ctor, then there must be a delete expression in a dtor.

5. 8 E 12 Initialize each class data member via the member initialization list.

   Within each ctor, for each class data member, there is at most one of ctors and assignment operators invoked.

6. 9 E 13 List members in a member initialization list in an order consistent with the order in which they are actually initialized.

   This example needs lexical information: in this case, the line and column numbers of each data member would be needed to compare their positions.

7. 11 E 15 Have the definition of operator= return a reference to *this. (Note: this says nothing about declarations.)

   This boils down to the requirement that there be at least one expression of the form "*this."

8. 12a E 16 Assign to every local data member inside operator=.

   This seems to be too strong; a member of "char *" may very well be first initialized in a ctor and then obtain its content through strcpy in operator=. Instead, we require that inside operator= there be an assignment expression for each non-pointer data member.

9. 12b E 16 Call a base class operator= from a derived class operator=.

   Of course, the prerequisite for this is that the base class defines operator=.

10. 12c E 16 Use the member initialization list to ensure that a base class copy ctor is called from a derived class copy ctor.

11. 13 Don't call virtual functions in constructors or destructors.

    Being virtual is a property of functions, and FCL can tell constructors and destructors through their special naming conventions. This is checkable in FCL. In fact, I did make such an error when implementing FCL.

12. 21 E 29 and E 30 Don't return pointers/references to internal data structures unless they are pointers/references-to-const.

    I am not sure this rule is absolutely correct; it seems highly depend on the semantics of the specific application. For example, to return a reference to char from a string seems appropriate to me. Instead, I would like to require that any expression returned be neither a pointer nor an array.

13. 22 M 26 Never define a static variable inside a non-member inline function unless the function is declared extern.

    Any non-extern and inline function should not define static variables.

14. 26 M 5 Avoid use of user-defined conversion operators (i.e., non-explicit single argument ctors and implicit type conversion operators).

    No expressions invoke conversion operators.

15. 29 M 6 Use prefix ++ and -- when the result of the increment or decrement expression is unused.

    In a program, there exist no such expressions that are either postfix ++ or --, but have no parent expressions.

16. 31 M 11 Prevent exceptions from leaving destructors.

    Currently FCL does not handle exceptions.

17. 32 M 13 Catch exceptions by reference.

Chapter 8

Related Works

This chapter surveys works related to FCL:

- Program analyses are technically relevant since they all require parsing and type analysis. Section 8.1 surveys and compares a set of such tools, analyzing their similarities and differences.
- Program specification languages are relevant since FCL is a specification language too. Section 8.2 summarizes the main features of two recent specification languages: Alloy [Jac02] and OCL [WK99], the interaction contract [HHG90], and the Wright architecture specification language [AG97].
- Tools specifically for error detection are surveyed in section 8.3. In particular, two closely related works, CCEL [LM93, MDR93] and CoffeeStrainer [Bok99], are compared with FCL.
- Other related works include documentation, tools for framework instantiation, and semantics for object-oriented languages. Section 8.4 surveys works in these areas.

8.1 Program Analyses

Many source code analysis tools [CNR90, PP96, PP94a, MN96, Dev99, DRW96, LR95, C$^+$00b, MS95, Cre97] have been developed to perform a variety of software engineering tasks, ranging from program understanding, design recovery, and reverse engineering, through software testing, to software transformation and restructuring. These tools are technically relevant to our work; knowing them can help us answer such questions: "since many tools have been created before, why bother developing a new one?" Moreover, by putting the tools together and comparing them, one can gain a better understanding of the whole problem space, the position of our tool in that space, and its strengths and limitations.

In general, these tools can be divided into two categories according to whether they need to change the source: *query tools* that only read source code and *restructuring tools* that both read and write source code.

Query tools parse existing source code and extract facts. The process can be summarized into three steps:

- Extract: Parse the source and extract elementary facts from it.
- Abstract: Perform queries on the extracted facts to derive information relevant to one's analysis tasks.
- View: Visualize the derived information.

Restructuring tools also parse and extract facts from source, but with a different goal: to either transform the source to another language or to "improve" it according to certain criteria. Consequently, the process now becomes *extract-abstract-transformations*, where the *transformation* step restructures code using the semantic information obtained in the abstraction step.

At the high level, all these tools consist of three key elements:

- *program databases* (also called *fact bases*, *program repository*, and so on),
- *schemas* (also called *conceptual model* [CNR90] or *data model* [CGK98]) for the program databases, and
- a *query* or *manipulation* language of some sort.

8.1.1 A summary of common tasks

A list of common tasks may help us better understand the needs for such tools. Query tasks found in the literature include:

- Architecture reconstruction and visualization
- Program organization
  Examples include the set of files in a program, header file inclusion, file interfaces (the set of functions defined in one file and called by another file), and so on.
- Reachability analysis and dead code detection [CNR90, CGK98]
- Computing metrics
  Examples include McCabe's cyclomatic complexity [Cre97, DRW96, PP96], average function and file sizes [PP96], FANIN and FANOUT [PP96], and so forth.
- Fine-grained query of syntactic code patterns, e.g. [PP94a, Cre97, Dev99]:
  - Finding all instances of simple assignment expressions appearing in any conditional expressions.
  - Finding all instances of equality-tests and dereference expressions whose results are discarded.
  - Finding all instances of conditional or iteration statements that have empty bodies.
  - Finding all *case* statement fallthroughs.

- Finding the set of *case* conditions of a *switch* statement.
- Finding all *switch* statements without the *default* branches.
- Finding gaps (unused space due to memory alignment) in structure definition.
- Finding instances of code patterns, such as three levels of nested loops or long *if-then-else*, as a candidate of rewriting to *switch* statements.
- Finding all compound statements whose bodies are not enclosed within curly braces.
- Finding all instances of variable redeclaration in nested scopes.

- Resource flow analyses such as data binding [PP96, CNR90]
- Control flow analyses such as Call Graph Extraction [PP96, CNR90]

Examples of transformation tasks include:

- Pretty printing,
- Semantics preserving transformation (refactorings) such as renaming variables and extracting methods,
- Code rewriting.

8.1.2 Some criteria for classifying tools

Lexical vs. syntactic approaches Many software tasks involve extracting facts from software artifacts, which may include not only source code but also documentation. These tasks are sufficiently broad so that it is hard to have one tool to satisfy all the needs. Instead, a spectrum of tools would have to be built, which can be categorized into *lexical* ones and *syntactic* ones.

Historically, programmers have been using utility tools such as *grep* and *awk* and scripts written in languages like *Perl* to perform such tasks. Since the underlying model that these tools have for the analyzed text is either character streams or record streams, they are *lexical* tools. *Lexical* tools provide certain formalisms of regular expressions to support pattern matching.

As described in the work of *LSME* (Lexical Source Model Extractor) [MN96], the lexical approach has the advantages of being:

- *lightweight*, in terms of writing specifications,
- *flexible*, in terms of the diversity of text that it can handle, and
- *tolerant* to certain syntactic errors in the text.

On the other hand, the simplicity of regular expressions often does not allow one to express queries that involve deep knowledge of the structure of the source code, or even if possible, the answers may be highly *approximate* in nature. Consider tasks

such as "querying the calls to an overloaded operator" or"querying all the variables whose names are x" for C++. For some task, this loss of precision is not tolerable; thus *syntactic* approaches are also needed.

Granularity and precision of program facts Program databases may contain three types of information with different granularities, *global structural information*, *statement structural information*, and *expressions and flow information*. *Global information* includes files, functions, global variables and types, for C, and namespaces, classes, templates, and so forth, for C++. *Statement structural information* includes details such as statement types and blocks; these are useful to identify *code patterns* that match a programming plan or cliche [RW88]. Last, *expressions and flow information* concern with the data and control flows among the expressions within procedures.

Choices of what information a program database may contain are driven by both the consideration of efficiency and the nature of the tasks at hand. In general, the more information, the more space occupied, and the slower the tools would be. Some tools, for example, CIA [CNR90], CIA++ [CGK98], and Hy+/GraphLog [CEH+94, MS95], trade the completeness of information for speed and space.

Completeness of program information has different impacts on different tasks. Works such as reconstructing architectural views from source code [Hol98] focus on the overall program structures and thus do not need detailed information about expressions and data and control flows. In contrast, FCL needs the full detail about expressions and some flow information to detect errors more effectively.

File based vs. whole-program based repositories Some tools [Dev99] work on a file-to-file basis whereas others such as [CNR90, CGK98, PP96] assume that a linkage has been performed over the compilation units and work on the whole-program repositories. While program based repository may cause some performance problem, it allows one to express certain types of queries more naturally and succinctly. Since the specifiers have no way to know the file names a framework user may use, file-based approach does not work for FCL. A whole program repository is necessary.

Concrete syntax vs. abstract syntax Some tools base their query languages on the concrete syntax of the subject languages while others on the abstract syntax. One advantage of using concrete syntax would be the ability to query syntactic patterns such as "all if statements that do not use curly braces." The disadvantage of using it is that if an abstract pattern has more than one corresponding concrete syntax pattern, the query can get quite verbose. Therefore, the advantage of using abstract syntax is the conciseness of queries.

Supported subject languages CIA [CNR90], SCA [PP96], SCRUPLE [PP94a], and TAWK [GAM96] support only C. Hy+/GraphLog [CEH+94, MS95], ASTLOG [Cre97], and CIA++ [CGK98] also support C++.

Query vs. transformation This has been discussed in the beginning of this section.

Supported formalisms A* [LR95] and TAWK [GAM96] support the awk style of programming, where the programs consist of pattern-action pairs and traversal strategies.

SCRUPLE [PP94a] is a query language expressing code patterns as regular expressions.

SCA [PP96] is based on a multi-sorted, sort-ordered algebra. Hy+/GraphLog [CEH+94, MS95] is based on a relational calculus extended with the closure operator. And ASTLOG [Cre97] is based on an extended Prolog.

Tool generators In this context, tool generators are tools that can generate program analysis tools. This definition can be confusing since it is not clear whether a tool or system that provides a language, such as a database management system with a SQL, is a tool or a generator. Our executive definition requires a tool to satisfy at least one of the following conditions to be a generator: either some of its components can be customized or replaced to produce a new tool, or the tools programmed with the provided language are important enough that it is justified to call them tools. GENOA [Dev99] is a generator in the former sense whereas A* [LR95], Aria [DRW96], and TXL [C+00b] the latter.

8.1.3 Tool generators

GENOA [Dev99] is motivated by two observations: first it is expensive to build reliable parsers for complex languages like C and C++; and moreover, in production environments it is a significant task to manage the build procedure for a product. Thus it is important for tools to preserve the efforts in creating the build procedures. GENOA emphasizes both the retargetability and the language-independence of the back end. The retargetability is achieved by defining a common denominator representation for most languages, ASG. A query interface is defined over this data model, and in turn a reusable scripting language is defined in terms of the interface. A translator generator, GENII, is also provided to facilitate the specification and implementation of the transition to a new language.

A summary of the important features of GENOA follows:

- It is front-end retargetable.
- It grafts itself onto a parser by replacing its back end with GENII generated code, which manipulates the parser produced AST and provides an ASG interface to the

scripting language. Since a parser works with compilation units, current instantiations of GENOA are all based on ASGs for individual files.

- The scripting language is procedural, and it allows users to determine traversal strategies.
- The scripting language cannot change the underlying ASGs.

Aria [DRW96] is a tool generator produced by retargeting GENOA to Reprise [RW91], a C++ ASG representation. Aria is used to generate a number of tools that would require long development time otherwise: a metrics tool, a path expression tool, and a CDG (Control Dependence Graph) generator. In essence, Aria is a validation of GENOA.

The TXL system [C+00b] supports program transformation. It was motivated by the desire to experiment with different dialects of the same base language. Using TXL, one can translate programs of a dialect to programs written in the base language. This is accomplished by transforming the parse tree built using the dialect grammar into a tree of the base grammar. A recursive tree matching algorithm is used to achieve this. TXL also finds many other applications [DCMS02].

8.1.4 Pattern based Tools

A* [LR95] is motivated by the desire to create language processing tools with extremely low overhead. The goal is to allow inexperienced users to build simple tools in a matter of minutes with a few lines of code. Therefore, the usual combination of YACC and C is considered too expensive for this. Moreover, the authors observe the similarity between language processing and file processing provided by tools such as Awk. A* comes combining the language definition facility of YACC with the pattern matching capability of Awk[1], which is described as follows:

- retaining the Awk action language and its interpreter;
- providing a mechanism for replacing Awk's parser with an arbitrary LALR(1) parser;
- providing a new data structure and notation for parse trees;
- providing a way to describe parse tree traversal;
- augmenting the action language to ease the construction of larger programs.

One important change that A* makes to Awk is its pattern language; instead of boolean expressions, patterns are specified using the concrete syntax of the processed language.

The following is an example of an infix-to-prefix translation with a yacc-like pattern:

```
_expression: _expression '+' _expression
```

[1] Awk is a language for file processing; the essence of its convenience is a control structure in all Awk programs. Awk breaks each file into records, and each record into fields. A typical Awk program consists of an implicit loop over all all records, and within this loop the user's code is executed. Usr's code consists of "pattern-action" pairs; a pattern is a boolean expression based on the value of the current record. An action is the statements to be executed once the pattern evaluates to "true."

```
{
   printf "Add(";
   traverse($$kids[expression#1])
   printf ",";
   traverse($$kids[expression#2])
   printf ")";
}
```

Where underscore prefixed symbols represent non-terminals in the concrete grammar.

A summary of the key features of A* is as follows:

- It provides mechanism to support new languages; therefore, it is a tool generator.
- Its pattern language is based on concrete rather than abstract syntax of the processed language. Therefore, its pattern-matching facilities are limited.
- The syntax must be LALR(1); this would make it impossible to process "irregular" but popular languages such as C and C++.
- It allows tool builders to specify arbitrary traversals of the ASTs.
- Its action language allows one to change the underlying ASTs, such as performing *tree surgery*.

Therefore, A* is suitable for tasks satisfying at least one of the following conditions:

- the semantics of the language is low (the language is simple).
- the desired tool implements little of the semantics.
- or the tool needs to examine only a proper subset of grammatical constructs.

SCRUPLE [PP94a] is a pure query system that can locate code fragments matching certain patterns. It exceeds in its support for a rich set of patterns.

- The pattern language is based on the abstract syntax of the language; at the same time, it also supports most code fragments of the subject language. This means that in certain cases one can directly type in a fragment of code and let the tool search for it.
- The tool appears to work on program-based ASTs.
- The pattern language provides a certain level of abstraction that allows users to control the precision of patterns. For example, *Query: Find sequences of statements that contain three* if *statements* can be specified as follows:

  ```
  if # @;
  @*;
  if # @;
  @*;
  if # @;
  ```

  In SCRUPPLE, # represents *expressions*, @ statements, and @* zero or more statements. @* provides the abstraction to represent arbitrary sequences of statements.

- Patterns are compiled into finite state machines, and search is efficient.
- Traversal is implicit and always starts globally.

Two more SCRUPPLE queries are included to give a taste of SCRUPPLE works:

1. *Query: Find all functions that have references to the identifier* xmax:

   ```
   $t $f<xmax>($v*) { @* };
   ```

   Where *$t* represents types, *$f* functions, and *$v** a sequence of variables.

2. *Query: Find all instances of three consecutive* if *statements*:

   ```
   if # @;
   if # @;
   if # @;
   ```

TAWK [GAM96] is a tool similar to A* in terms of its overall structure, with differences summarized as follows:

- TAWK is a tool, not a generator; it does not assume the responsibility of parsing source code.
- TAWK's patterns are specified in terms of abstract syntax.
- TAWK's action language is C.
- TAWK recognizes the importance of abstract patterns, and provides a simple, macro based mechanism.
- TAWK provides implicit traversal control.
- TAWK recognizes the importance of dealing with macros and compilation directives in languages like C, and can recognize function-like macro invocation as function calls.

8.1.5 Algebra based Tools

Informally, algebras are mathematical structures that consist of data types (*sorts*) and operations defined on the data types (*operators*). One familiar example is the classical relational algebra [UW97]; Operators of the algebra such as *union*, *set difference*, *select*, *project*, *cartesian product*, and *join*, take relations as arguments and produce new relations. It is, however, a *one sort algebra*, that is, it deals with only one data type, i.e., *relations*.

SCA (Source Code Algebra) [PP96, PP94b] is an algebra-based formalism that can express queries on C programs. Its main features are summarized as follows:

- SCA is both *many-sorted* and *order-sorted*.
- SCA recognizes the importance of *information completeness*. Its data model contains not only *global structural information*, but also *statement-level structural information* and *control and data flow information*. It also supports queries that use both structural information and program flow information.

- The operators of SCA can be categorized into:
  - operators on object attributes

    SCA defines four types of attributes for objects, *components* to reflect the structural relations between a conditional and its condition and body, *references* to capture name references, *annotations* for information such as line numbers that an object appear at, and *methods* for dynamically computation of new information from the model. It is less clear how useful *methods* would be.

    To get an attribute, one can apply the name of the attribute to the objects. For example, applying the operator *name* to a file *f*, *name(f)*, will return the file name.

    An important feature of SCA is *closure*; for example, it allows one to specify all the functions that a certain function can be reach.
  - operators on collections

    SCA allows one to *select* a subset or *pick* one element, which satisfy a given condition, from a collection of objects. It also supports columns projection through the *project* operator, and cartesian products through the *product* operator.

    One important feature of SCA is the *extend* operator, which allows one to extend the existing type definitions with new attributes. Another is that it supports logical quantifiers *forall* and *exists* by, however, treating them as operators. SCA also defines two higher order operators, *apply* and *reduce*; *apply* takes both a collection and a unary operator, and apply the operator to each element of the collection to get a new collection; *reduce* takes a collection and a binary operator, and apply the operator to each pair made of elements of the collection to get a new collection.
  - set operators
  - sequence operators

Two questions are left unanswered by the SCA work: First, it seems that SCA is more driven by the desire of having a formalism than by the kinds of queries that would be needed in practice; it is not clear under what circumstances operators like the *methods* attributes on objects and the *reduce* operator on collections would be useful. Second, SCA is only demonstrated with C as its subject language.

8.1.6 Logic based Tools

Hy+ [CEH+94, MS95] is a generic visualization tool that supports a visual query language called GraphLog. Hy+ supports a graphical formalism that allows comprehensible representations of databases, GraphLog queries, and query answers to be interactively manipulated. Hy+ [MS95] has been applied to tasks such as software metrics, verifying constraints, and

identifying design patterns from code.

GraphLog supports two types of queries, *define* and *filter*. Both queries represent a pattern in the hygraph; the query interpreter searches the hygraph designated as the database for all occurrences of that pattern. A *define* query can be translated to stratified Datalog [UW97]. A *filter* query can be viewed as composed of multiple *define* queries, and its result is the combination of evaluating all the *define* queries.

GraphLog also supports aggregate functions on multisets of tuples, such as *MAX*, *MIN*, *COUNT*, *SUM*, and *AVG*.

Since GraphLog is based on Datalog [UW97] and has the same expressiveness as a *relational algebra extended with generalized transitive closure*, which can be used to, for example, compute the inheritance relation through inheritance chains, it is more expressive than *relational algebra*. But adopting a relational model of source code prevents it from *conveniently* expressing queries that pertain to abstract syntax trees. Moreover, when the complexity of GraphLog queries increase, to what extent its visualization facility can scale up is questionable.

ASTLOG [Cre97] exploits the unification and backtracking facility provided by Prolog to examine ASTs. The key changes ASTLOG makes to Prolog are:

- ASTLOG extends Prolog with primitives suitable for examining tree structures.
- For performance, instead of importing program databases into Prolog as its fact bases, ASTLOG interprets its predicates and queries on top of external objects, in this case, nodes of ASTs.
- ASTLOG adopts the so-called Prolog-with-an-ambient-current-object approach, that is, terms are always evaluated against a current object. For example, a predicate *assignment* can be defined as follows:

  ```
  assignment(target, value)
            <-  op(#=),
                kid(#LEFT, target),
                kid(#RIGHT, value);
  ```

  Note that in the example, there is an implicit current object.
- ASTLOG implements some second-order features such as lamdas and function applications, and Prolog set-predicates through manipulating the current object.
- Compared with the AWK approach, ASTLOG provides additional expressive power; for example, with ASTLOG, one can compare two trees.
- The author feels that ASTLOG needs a support for type checking.

The SOUL (Smalltalk Open Unification Language) work [Wuy98, MMW02] advocates a logic programming approach to manipulate structural information in source code. SOUL is a Prolog variant that currently works with Smalltalk as the subject language. Implementation-wise, SOUL [Wuy98] adopts the same strategy as ASTLOG, that is, instead of importing the

program databases as Prolog fact bases, it interprets logic primitives against the Smalltalk image, through a well-defined access interface.

SOUL is targeted at general programming, instead of specific domains like frameworks. As a result, its rules might be less accurate in terms of finding code patterns, and it is undesirable for a tool to generate too many false positives and false negatives. There is no work seen towards addressing these issues. Therefore, it is not clear how effective it would be in practice.

8.1.7 Schemas for Program Databases

One early work in storing program facts into a relational database is OMEGA [Lin84], which uses the INGRES database management system to store information for a programming language called *Model*. One of the goals of OMEGA is to reconstruct software objects from the program database. Therefore, detailed information about variables, expressions, statements, and relationships among them are stored in the database. A total of 58 relations were used in the database schema. According to [Lin84], the prototype implementation of OMEGA had poor response time in retrieving the body of a procedure. Different objects within the procedure had to be retrieved, and each retrieval required a separate database query.

Over the years, many schemas have been proposed and tried, e.g., Reprise [RW91], CIA++ [CGK98], RSF [MOTU93], and Datrix [HHL+00]. More recently, there have also been works towards a common format for exchanging software information [Hol97, FSH01, WKR02].

When choosing a schema, one should consider many factors, such as the level of semantics, the amount of information, the granularity of the information, and the precision of the information. The level of semantics refers to lexical, syntactic, and semantic kinds of information. The amount of information a schema can provide may also vary, some may provide only lexical and syntactic information while others all three kinds. Granularity refers to the level of detail which the information can reach, for example, Rigi [MOTU93] and CIA++ [CGK98] only keep global structural information such as function and class declarations whereas Datrix tools also provide information about statements and expressions. Finally, precision refers to the extent to which a schema can recognize things such as implicit conversions and operator calls.

It is not clear whether it is possible to devise a *universal* schema that fits all languages and all needs. But it is clear that the current focus should be in understanding the needs of different tasks and the intricacy of individual languages, and correspondingly, experimenting with different combinations.

8.1.8 Query Languages for Program Databases

In general, the nature of a query language closely depends on the underlying schema that organizes the data. Therefore, it is critical to having the right schemas that capture the essence of the data, otherwise, unnecessary details will creep into the query language. For example, the goal of the relational model is to achieve better *data independence*, as shown in [Cod70], by formalizing the data model with the relation theory to eliminate several unwarranted dependencies.

SCA is a many-sorts, sort-ordered algebra whereas both GraphLog and ASTLOG are logic based. In addition, GraphLog is a visual formalism purported to facilitate the query and display of program facts. One of the advantages of having a many-sorts algebra is the possibility of type checking, of which several authors [LR95, Cre97] have felt the need.

Grok [Hol02] is based on a binary relation algebra extended with some procedural constructs such as assignments and loops. A comparison between Grok and another query language, GReQL (Graph Repository Query Language), can be found in [HWW02].

8.1.9 Potential Research Problems

We have identified five problems for future research:

- It is a challenging task to engineer reliable parsers for complicated languages such as C++ [SEH03].

  Sim *et al* proposed to build and share a common set of correct parsers within the research community. Free, open-source parsers for C and C++, for example, CPPX [MDH01], are now available.
- Dealing with macros and compilation directives in languages like C is a problem.
- Query languages need to support abstract patterns for better expressiveness.
- Is it possible to have one query language that serves all the purposes?

  Rather than serving as a generic query language on program databases, FCL was designed specially for "framework constraints"; this is evident from, for example, the inclusion of the existential operator and the exclusion of a potentially useful "ancestor" operator for obtaining all base classes.

  FCL is certainly not complete if one views it as a generic query language for program databases. It remains an interesting question whether one can extend it to a generic one.
- Given the modern computing power, is it feasible to build full ASGs?

  FCL is based on the assumption that a full ASG is available. Real projects often require fast turn-around time. One concern is the feasibility of building ASGs in such a short time.

8.2 Specification Languages

The Alloy [Jac02] specification language is for formalizing and analyzing 'object models'. The OCL (Object Constraint Language) [WK99] of UML was developed with many the same goals as Alloy. They both have the Z specification language [Spi92] as their common root.

Many specification languages can be found in the literature. We choose these two to show that they cannot be used in place of FCL.

8.2.1 Alloy

In Alloy [Jac02], an object model captures the basic structure of a certain problem domain, as well as constraints and operations describing how the structure changes dynamically. Alloy is meant to give an entirely abstract, implementation-free semantics to object models. The insight is that such a semantic model is a better starting point for object-oriented development than a model in which objects have methods and fields. One outstanding feature of Alloy is its fully automated analyses.

Alloy's starting point is Z [Spi92]. Like Z, Alloy is also based on sets and relations, but it contains a few novelties:

- Alloy treats scalars as singleton sets.
- Alloy's navigation syntax for relational image allows one to form expressions by following relations. This feature, combined with the scalars as singleton sets feature, both simplifies and unifies navigation expressions.
- Alloy allows one to put both type information and multiplicity into relation definitions. Moreover, instead of the range notation, Alloy uses the regular expression style for multiplicity, that is, * (zero or more), + (more than zero), ! (exactly one), and ? (zero or one). These can help shorten specifications considerably.

Unlike Z, however, Alloy is designed with automatic analysis in mind at the first place. This requirement has a number of important implications on the design of Alloy:

- Alloy specifications are explicitly structured into *paragraphs*, which include domain declarations, state declarations, conditions, invariants, operations, and assertions. Z exploits conventions to distinguish the roles of the various schemas.
- Alloy exploits two forms of analyses on its specifications, *simulation* and *checking*. Alloy's structuring makes the two analyses possible. The goal of simulation is to find a model[2] for a specification; if a model is found, then the specification is considered *consistent*. The goal of checking is to find *counter examples* that fail the assertions,

[2] *model* is an overloaded term; here it means a configuration of state that satisfies the specification.

which is done by finding models for the conjunction of the negation of the assertions and the rest of the specification.

- Since the language is undecidable, compromise is inevitable to make the analyses feasible. Alloy's analysis works by limiting the scopes of the carrier sets of the primitive types to a given size. The analysis is done by first translating each relation into a boolean formula, and an off-the-shelf SAT solver is then used to find solutions to the formula.
- Alloy is implicitly typed.

FCL can be viewed as an Alloy whose domains are the syntactic elements of object-oriented programming languages. It would be interesting to encode FCL specifications with Alloy and detect inconsistency of FCL specifications. Some difficulties would be the constructor functions of FCL, string, and integers, which Alloy does not support yet.

8.2.2 OCL

The Object Constraint Language [WK99] of UML is meant to bring rigors into the UML notation. However, many shortcomings have been identified [VJ99, Jac99]. OCL is still not amenable to automatic analysis. As it is, OCL is more a notation for human communication than for automation, yet far from natural languages.

A summary of OCL's shortcomings is as follows:

- OCL is driven by the desire to support the practical use of object-oriented languages, as such, it is too implementation-oriented, thus is not as well-suited for problem analysis as Alloy.
- OCL is more expressive than Alloy; it has integer and string datatypes, and sequences. But it does not have transitive closure. And attempt to use operations to simulate transitive closure turns out to be unsafe [VJ99].
- Giving OCL a semantics is likely to be challenging because of its many features such as the elaborate type system, type casting, multiple inheritance, and iteration construct.
- The syntax of OCL has some shortcomings. OCL's expressions are stacked in the style of Smalltalk, which makes it hard to see the scope of quantified variables. Unlike Alloy, navigations in OCL are applied to atoms and not sets of atoms. Attributes are modeled as partial functions in OCL, and result in expressions with undefined values.

Although OCL has been used to specify constraints for the UML meta model, it is not suitable for our purpose. OCL supports quantification formulas and standard set operations, but it does not contain sufficient information about source code: for example, neither expressions nor control paths are supported by OCL. This is all right for a design modeling language such as UML but not for FCL. As mentioned, OCL's syntax is not standard but

chooses to keep close to the object-oriented notations, presumably to make programmers feel familiar. FCL is designed to be close to classic notations. One benefit of this design decision is that we may do inference about FCL specifications in formal systems such as Mizar [Org]. At last, although tools are available for OCL, we did not find they are suitable for FCL; for example, the Dresdren OCL toolkit works only at the first-order level: it allows only for specifying pre- and post-conditions and class invariants but not constraints on syntactic structures.

8.2.3 Other Notations

Interaction contracts [HHG90] are the first notational constructs proposed to specify behavioral compositions. Particularly, it focuses on specifying the obligations that the participating objects should fulfill in order to function correctly. The aim is to explicitly capture the dependencies among cooperating objects, which otherwise will be buried in the code of classes and methods. Contracts and FCL are close in that both want to specify how objects can correctly interacted with each others. There are also important differences between them. Contract specifications need to be complete and self-contained whereas FCL specifications can be partial. Contracts can contain statically uncheckable invariants while FCL is fully automated. However, contracts can be more expressive than FCL due to its informality.

The Wright language [AG97, AGI98, SG96] aims to describe software architectures, especially concurrency-related ones. It is an extra layer beyond code, meant to be used as a stand-alone design language, and there is no support to ensure the consistency between the code and the Wright specification.

In contrast, FCL works directly on the syntax and semantics of specific object-oriented programming languages. Instead of full architectures, FCL specifies checkable constraints on the code of framework extensions. The applicable scope of FCL is limited in that it is designed to ensure the appropriate reuse of object-oriented frameworks rather than arbitrary software. But since frameworks are concrete implementations of software architectures, we can also regard FCL as a specification language on ad hoc object-oriented software architectures, and we believe that this study can further our understanding of software architectures.

8.3 Error Detection Tools

8.3.1 (Partial) Specification based

Aspect [Jac95] is a static analysis technique that aims to detect errors of missing dependences from within procedure implementations. A simple specification language is defined so that, for each procedure, the specifier can specify dependences that should exist between its

prestate and poststate. A dataflow analysis is then performed on each procedure, any dependences that exist in the specification but not in the result of the dataflow analysis will be reported as errors. To deal with pointers and pointer aliases, *reconfiguration assertions* are introduced; a reconfiguration assertion states all the possible bindings that may exist between a pointer value at the poststate and pointers of the prestate. It is a special case of dependence assertions.

Specifications written directly in terms of *implementation* variables are vulnerable to changes in representation. To cope with this problem, the concept of *aspects* is introduced; *aspects* are simply names for abstract components of abstract types. For example, a two-dimensional vector can have two abstract components, X and Y, as its coordinates. The dependence and reconfiguration assertions are written with aspects instead of concrete names, so clients of the abstract type see only the division into aspects, with the actual representation remaining hidden. *Abstraction functions* are then used to associate *representation* with *aspects*.

A summary of the important features of Aspect follows:

- It requires specifications, albeit simple, for each procedure.
- It works in a modular fashion, that is, checking is performed on one procedure at a time.
- It never produces false positives, provided that the specifications are correct.
- It may produce false negatives, though.
- It is incremental in that it does not require the completeness of the project to work. Therefore, it can precede testing.
- Compared with verification, its specifications are cheaper and easier to write.
- It detects errors of omission, which complements state-based techniques such as type checking, which detects errors of commission. The empty procedure, *SKIP*, satisfies most type specifications, but no nonempty Aspect specification.

The LCLint tool [E+94, Eva96], subsequently renamed as Splint [E+], exploits simple annotations to explicitly specify the otherwise hidden assumptions at interface points about the return value of functions, parameters, and global variables. The tool then combines these knowledge with static analysis to detect errors in a modular fashion. For example, by default a pointer is considered being not null; one can use *null* to annotate the fact that it can take the null value. A function that uses this pointer without proper check of the nullness of the pointer would then contain an error. Similarly, if the memory pointed by a pointer is not allowed to be shared, one can use *only* to annotate it. In [Eva96], 15 annotations are defined to help capture dynamic memory errors.

The ESC (Extended Static Checking) tools [DLNS98, LNS00, FLL+02] shares the same goal of detecting errors from code as many other tools. It is static yet can detect such

errors as dereferencing null pointers, array index errors, and race conditions and deadlocks, which conventional static analysis like type checking cannot find, hence the name *extended static checking*. ESC exploits the automated program verification approach rather than static analysis. Therefore, it requires specification support for both procedural abstraction and data abstraction. Verification is carried out in a modular fashion; for each module, a verification condition is generated and a program verifier then attempts to prove it. ESC sacrifices soundness for error detection: it is more interested in a failed proof; then an error message can be produced. Tools for both Modula-3 [DLNS98] and Java [FLL+02] have been constructed. A methodology for verifying programs with the appearance of data abstraction and information hiding is generated with the ESC/Modula project [LN02].

The Vault [DF01] language allows users to specify typestates for resources, such as memory regions, files, and sockets, to track their safe uses. Programs written correctly using the language would then be protected from the *resource management* related bugs. Despite the language approach adopted, the notions of *keys* and *type guards* are very close to the annotations of LCLint [Eva96]; indeed, they are meant to solve the same *resource management problem*. Historically, language adoptions have been an erratic process. In contrast, analysis tools work immediately.

8.3.2 Model checking based

Dawson Engler's metacompilation (MC) research group at Stanford investigates the uses of both static analyses and model checking for finding bugs from code. Their static analysis allows programmers to add simple system-specific compiler extensions that automatically check or optimize the code. They also build a model checker, CMC, for C, and apply it to several case studies [MPC+02, ME03].

The static analysis is based on an extension language and a back-end engine added to the *gcc* compiler. The extension language, *metal*, is defined to specifying analyses, and the back end, *xgcc*, is used to execute the analyses efficiently. The detail of *metal* and *xgcc* is documented in [HCXE02, CHE02], but a brief summary follows:

- The extension language is program object-centered; it can specify the states of any program objects, such as pointers, as a finite state machine. In particular, erroneous states are explicitly modeled by the state machine; when the state machine reaches such states, error messages will be reported.

```
state decl any_pointer v;

start: { kfree(v) } ==> v.freed;

v.freed: { *v } ==> v.stop,
       { err("using %s after free!", mc_identifier(v); }
       | { kfree(v) } ==> v.stop,
       { err("double free of %s after free!", mc_identifier(v); }
```

;

This example specification can detect two errors about a pointer, dereferencing freed memory and double frees, by tracing its state. v is a (specification) variable representing a pointer variable in the checked program. A variable can have states; for example, v has two states, *freed* and *stop*. The current state of a variable such as v is written as *v.freed*. The alphabet of the state machine consists of code patterns; for example, *kfree(v)* is actually a free call on v, and *v is an expression dereferencing v.

- The *gcc* compiler is used to construct an AST for each C file. The *xgcc* engine then reads in the ASTs and performs analyses on the control flow graphs of all functions not called by any others, e.g., the main function.
- The engine essentially traverses each AST one path at a time with a depth-first search, evaluating the current state of each path and using the state to drive the transitions of the finite state machine model.
- The traversal performed by *xgcc* is interprocedural, and a simple, path-sensitive analysis is used to eliminate nonexecutable paths.

The most interesting insight of this work is that the seemingly simple technique can be effective in finding errors in real systems. The paper [C+00a] shows how *metal* is applied to the Stanford FLASH machine's embedded cache coherence protocol code. The paper [E+00] discusses a set of small extensions that found roughly 500 bugs in Linux, OpenBSD, and the Xok exokernel; the extensions are usually less than 100 lines. It also uses extensions to find hundreds of optimization opportunities in heavy-tuned software.

CMC [MPC+02] is a model checker that directly executes C and C++ programs and performs model checking on the monitored state space. CMC has some limitations, though. The current CMC assumes an *event driven model* of the checked system. Its correctness properties are coded as boolean functions directly in terms of the saved states. Furthermore, the AODV case study in [MPC+02] does not present how temporal properties can be supported. Without temporal properties, CMC seems to be more an advanced testing tool than a model checker.

Based on the experience applying both approaches to three non-trivial case studies, [ME03] presents an interesting comparison between the two approaches and some lessons learned:

- Static analyses generally can find more errors than model checking. There are two reasons for this: first, models and environments are abstractions of the underlying implementation and many details are suppressed from them, thus model checking tends to miss many errors that static checking can directly get from code; second, model checking, similar to testing, can only find errors from executed paths; incomplete

environment modeling is bound to miss many errors on the non-executed paths.

- Easy-of-inspection really matters; tool users prefer to having the priority of the errors ranked, especially when facing thousands of errors.
- Model checking is better at finding errors involving global invariants and values, thus complements static analyses.
- The more analyses applied to find an error, the harder the error is to reason about and to fix. Furthermore, the higher the chances are for one of the analyses to go wrong.
- Model checking requires manual construction of both the models and (optionally) the environments within which the models are supposed to function, these tasks can be both expensive and error prone. Errors in the models can trigger false positives. Moreover, applying approximation and abstraction during model construction suppresses implementation details, thus can miss real errors in code. It is crucial to reuse the efforts invested in constructing the models.

One more thing worth noting is that the implementation model of the subject systems (the FLASH protocol [C+00a], AODV [MPC+02], and TCP [ME03]) is closer to the finite state machine model of the model checker. These systems are all implemented in C.

In the paper "Lightweight Analysis of Object Interactions" [JF01], Jackson and Fekete outline a method for automatically detecting design errors that are related to object interactions. The formal notation Alloy [Jac02] is used to represent the abstract program. With Alloy, the heap structure of the system under analysis is explicitly modeled; a global relation that maps an object reference to its abstract value is included in the program state. Methods called but not under reasoning are specified declaratively. In order to analyze whether a particular method maintains a certain property, an interaction diagram is extracted from the method; all the methods called within the diagram are specified, and their specifications are conjuncted to form a formula, with different variables explicitly representing the states between the method calls. The formula is then conjoined with the specifications for the other parts of the system, that is, the heap, classes, and methods. The combined specifications are an assertion that the system should support. Alloy checks this by finding a counter example-that is, a model of the negation of the formula. The formula is solved by first translating it to a propositional boolean formula, and then applying an off-the-shelf SAT solver to the boolean formula. An example of the so-called comodification problem is used to illustrate the whole process.

A number of challenges have also been pointed out. For example, one is that given an interested property, how to automatically extract a behavioral skeleton like an interaction diagram, from the source code, using some form of static analysis. Another is how to deal with conditional and loop statements so that a formula can be formed and the analysis can be carried out.

8.3.3 Static analysis based

This subsection reviews two works most closely related to FCL. They share the similar goals with FCL. One has special notations and the other depends on annotations on code. In both cases the checking is automated.

8.3.4 CCEL (C++ Constraint Expression Language)

CCEL [LM93, MDR93] and FCL have different contents in the data models for the underlying programming language (C++). CCEL models only at the C++ interface level. It does not consider for example, expressions. FCL not only covers expressions, but also provides some rudimentary intraprocedural control flow analyses. This turns out to be fairly useful. FCL at present handles more constructs than CCEL, for instance, template instantiations, pointer and reference types, namespaces, function pointers, and so on.

CCEL models program elements separately. This is unnecessary and inappropriate. For example, parameters, data members, and variables were treated as three independent elements, so were member functions and free functions. In contrast, FCL has a different way of modeling. For instance, by introducing the notion of program units, FCL makes it possible to distinguish parameters, data members, and ordinary variables by examining their respective program units. Therefore, FCL significantly simplifies the data model. This is important. It simplifies concepts and makes it possible for a simpler FCL.

FCL has developed a better formalism than CCEL and is more expressive than CCEL. FCL bases itself explicitly on the first-order logic with the additional support of sets and sequences, whereas CCEL at best does so implicitly. For example, it is not clear whether CCEL can support arbitrary levels of nested universal and existential quantifications. With sets, FCL can express constraints on the size of sets, so that one can say something like "the size of the set is 1." CCEL has no way to express this.

CCEL defines CCEL variables through a syntax that mimics C++. For example, in CCEL one needs to specify a member function of a class as follows:

```
Class C;
MemberFunction C::mfunc;
mfunc.name() == "aName";
```

whereas in FCL:

```
mfunc as function("aName",C);
```

The advantage of our notation is that it is more likely to be portable, since it does not depend on the special syntax of any particular programming language. More importantly, by using well-known mathematical operations such as set comprehension, FCL specifications should be clearer in terms of semantics and thus easier to learn.

FCL aims at frameworks whereas CCEL so far has been mostly applied to generic design principles or implementation guidelines. It seems that frameworks bring FCL a unique advantage by providing more specific contexts, such that the tools can cover a broader range of common mistakes, and at the same time, generate fewer spurious ones. In fact, even Meyers himself admitted that "it quickly became apparent that the great majority of the guidelines used by good C++ programmers were too difficult to formalize, or had too many important exceptions, to be blindly enforced by a lint++[CCEL]- like program" [Mey92b].

8.3.5 CoffeeStrainer

CoffeeStrainer [Bok99] is a framework that allows programmers to specify constraints on the structure of Java programs. It works on the Java AST [BS98] to enforce constraints on programming conventions, type definitions, and type uses.

CoffeeStrainer does not have a specification language, instead, it depends on special annotations on the checked code. To use it, a programmer has to define an empty interface to express the constraints, and manually marks the class to which they want to apply the constraints with the interface. The constraints are implemented as Java code that works on the AST, and is hidden as comments within the body of the empty interface. At compile time, once the framework detects such interfaces, it will extract and dynamically compile the constraint code and apply the constraints to the corresponding types of AST nodes. CoffeeStrainer relies on naming conventions to relate the constraint code to the type of node to which it is applicable. For example, a constraint implemented by the method "checkField" would be applicable to all the fields within a checked class.

Clearly, because of its dependence on features such as dynamic compilation and empty interfaces, it is not easy to port CoffeeStrainer to other languages such as C++. In addition, to apply it programmers would have to annotate their code with the constraint code. It is not clear whether this "intrusiveness" is desirable. Furthermore, CoffeeStrainer requires programmers to write the constraints. Typically, if one knows the constraints, then one should be able to check them manually right away, without bothering writing them down; it seems hard to justify the need of such a tool.

8.3.6 Hybrid Approaches

Pattern-Lint [SSC96b] aims at monitoring the compliance of a software system with its high level design models using program databases and a Prolog engine. *Pattern-Lint* explores three types of *design models*, *concrete but low level rules* that can be phrased directly in terms of programming elements, *architectural level rules* such as design patterns and architectural styles, and design heuristics such as *low coupling and high cohesion*. Realizing that general design rules can have multiple concrete implementations, and that checking the conformance

to such heuristics as low coupling and high cohesion is largely human-dependent, the authors adopt a hybrid approach: in addition to static analyses, techniques of dynamic analyses and visualization [SSC96a] are also combined to help query and visualize the implementation, not only to detect violations, but also to increase the confidence that an implementation is faithful to a certain design model. Examples taken from the *Choices* [C+93] framework is used throughout the paper. More details can be found from Sefika's Ph.D. thesis [Sef96].

8.4 Other Related Works

Documentation The problem of framework usability has led to the proposal of several documentation approaches ranging from prescriptive to descriptive, and from informal to formal. Prescriptive methods describe how the framework should be used while descriptive ones describe only the design of the framework and users have to deduce how to use it. Examples of both prescriptive and informal approaches include patterns [Joh92], cookbooks [KP88], and hooks [FHLS97]. Examples of formal but descriptive methods include the interaction contracts [HHG90] and the interface contracts [Mey92a]. Other methods such as design patterns [GHJV94] and metapatterns [Pre95] are both prescriptive and descriptive.

Tools Support for Framework Instantiation Several authors have been working on providing tool support for framework instantiation. Active cookbooks [PPSS95] is a semi-automated tool that can enact recipe descriptions, providing users an interactive interface that guides them through the instantiation process. However, recipes do not explain design rationales; they describe only how the problem can be solved using the framework. The primary drawback of the approach is its inflexibility, namely, users have to either follow the recipe up to the last detail or abandon the tool completely. Ortigosa et al [OCM00] present another tool that utilizes intelligent agent technology to assist framework instantiation. The tool asks users to select from a list of functionalities and based on the selection, an agent elaborates a sequence of programming activities that should be carried out in order to implement it. In [FBML00], Fontoura et al propose to use DSLs (Domain Specific Language) to describe the variation points of frameworks. The user uses the provided DSLs to specialize variation points and the application is generated by transforming the DSL descriptions into the underlying implementation language. These works are fundamentally different from ours: first, they are all synthetic rather than analytic approaches; this may lead to the difficulty of integrating with existing development processes. Second, all the approaches work only on the structural aspects of programs such as class skeletons and method signatures; none of them support method implementation.

Several industrial frameworks have IDEs (Integrated Development Environment) associated with them [IBM, Mic]. In particular, the Visual C++ IDE provides tools such as GUI builder, resource editor, class wizard and application wizard to help the user instantiate the MFC framework; some of these tools can even generate code. However, tools cannot serve best unless they are mastered by the users, as some expert MFC users have criticized: "automation does not help much until the user knows exactly what is happening behind the scene" [SW96, New].

Specialization Interfaces Several authors have been working on issues related to the specialization interface. Kiczales and Lamping [KL92] discuss the issues in the design of class libraries and emphasize the importance of documenting internal dependencies. Lamping [Lam93] proposes to extend type systems to formalize not only the client interface but also the specialization interface so that the latter can also benefit from the automatic checking provided by type systems. However, the expressiveness of type systems is limited. Stata and Guttag [SG95] propose a methodology for the specialization interface based on specifications rather than type systems. Their approach partitions a specialization interface into method groups and associating substates with them; programmers are required to re-verify a whole group whenever any element within it is changed by a subclass. Reuse contracts [SLMD95] document the design relevant part of a specialization interface. A reuse contract is a set of method signatures, each associated with a specialization clause. A specialization clause names the signatures of those methods of the same class that are crucial for the design of the particular method. Furthermore, several operators on reuse contracts will be applied each time a subclass is created or a class is changed by developers. Reuse contracts and their operators serve as structured documentation and facilitate the propagation of changes to reusable assets by indicating how much work is needed to update applications built previously and where and how to test and adjust these applications.

Object Types and Subtyping Conventional type rules for object-oriented programming languages such as the covariance/contravariance rules are more about "syntax" than "semantics." Nierstrasz [Nie95] proposed the notion of "regular type" to characterize the non-uniform availability of object services. Liskov and Wing [LW94] present two definitions for the subtype relation that differ on the treatment of *history properties*. One definition demands the explicit specification of constraint rules in the supertype; any subtype has to verify that they preserve the rules. The other definition requires that any new method of the subtype be expressed in terms of the methods of the supertype, resulting in an *extension map*. In addition, both definitions require that (1) values of both supertype and subtype satisfy type invariants and (2) behavior con-

formance between the same methods of supertype and subtype. The authors further distinguish between extension subtypes and constrained subtypes; both are applied in practice. For example, commonly framework builders implement part of a class and let the users extend the uncertain part by subclassing it and adding new methods and variables. It is also common for users to override a method and provide a more constrained version. These are in general related to our work because we should at least be aware of the formal semantics of object types and subtyping.

Chapter 9

Summary and Future Work

9.1 Summary of Result and Contributions

Object-Oriented frameworks are composed of collaborating classes that provide standard solutions to a family of problems commonly encountered among applications in some domain [JF88]. Framework builders provide mechanisms, the variation points, that enable developers to use the framework to construct their specific application [Deu89].

The size and complexity of frameworks and their notorious lack of design and intended-usage documentation make framework-based development a learning-intensive and error-prone process [HHY03]. Commonly framework users misunderstand the relation between their application and how the framework designer intended the framework to be used, resulting in overly complex solutions or subtle bugs.

For the framework user with shallow knowledge, something more akin to type-checking is desirable. That is, framework developer takes on the burden of describing/specifying how to properly use the framework so that compliance by the framework user can be checked mechanically. Although correct type matching is no guarantee that a function is called properly, it does catch many common mistakes. We would like something similar to apply to framework use.

We use the term *framework constraints* to denote the knowledge that a user needs to know in order to use a framework properly. The idea is to formalize the framework constraints on hot spots and check whether a framework instantiation satisfies these constraints. Our goals are to create specification languages and tools that enable framework builders to encode their knowledge about the intended use of the framework and use the knowledge to check user applications.

This thesis focuses on the structural aspect of framework constraints. A specification language, FCL (Framework Constraint Language), is designed to express the constraints on the structure of source code. The feasibility and effectiveness of FCL are demonstrated through several case studies. We conclude that it is indeed useful to framework users.

FCL is a typed first-order logic extended with set and sequence operations. Its term language is a special set of total functions, reflecting the entities and relationships of programming languages. This formalism lends to the automation of FCL.

Technically, FCL is more akin to type-checking. The difference can best be illustrated by an example: while the empty program would pass all the type checkers, it can hardly pass any non-trivial FCL specifications.

Several case studies have been done to gain knowledge on how FCL can be better designed and used, and whether it can be helpful to practice. The following are the general guidelines that we learn from our experience:

- In general, the more specific the context is, the more effective FCL tends to be. Many examples show that the components that FCL is used to constrain are highly specialized; they often assume a great deal of context. FCL can be more effective under this kind of circumstances.

- Specifiers should know the design well and avoid immature generalization. However, this often implies that the specifiers should be the designers or somebody who works closely with them.

- Specifying against specific symptoms sometimes can be more economical.

In practice FCL has found both errors of omission and errors of commission. Many errors are design errors. They are caused by misunderstanding and/or being unaware of properties and interaction at system interfaces. The errors are both system- and domain-specific. They are different from the generic implementation errors such as dereferencing null pointers or array bound overflow. Independent of the domains of the programs being inspected, they can occur in any of them.

In short, this work proposes to extend the technique of type checking and apply it to framework-based development. A summary of the main result and contributions are as follows:

1. The design and implementation of FCL make the bulk of the work. A model of the static properties of C++ programs is presented, and an FCL is defined based on the model. A formal semantics is also defined for FCL.

2. The feasibility and the potential usefulness of the approach are then demonstrated by applying FCL to real frameworks. Specific lessons learned from the experience are reported. These lessons are important for both the use and future development of FCL.

   To use a framework is to learn its design; many problems originate from a lack of

understanding about its architecture, design pitfalls, and evolution. FCL has been used to perform the following kinds of tasks:

- Detecting omission of programming obligations;
- Detecting violations of programming constraints implied by a design;
- Detecting violations of programming constraints implied by a "negative design;"
- Enforcing programming disciplines;
- Helping with the evolution of both the framework and its intended use.

9.2 Future Work

Framework-based development has become indispensable in modern software engineering. The difficulty of framework use probably will stay with us as long as we program; thus it is important to both further develop FCL in particular and deepen our understanding of framework-based development in general. In the following, I outline some possible future work:

Designing FCLs for other languages like Java and oo Perl In general, I believe that it should be straightforward to develop FCLs for other languages. Based on experience with developing FCL for C++, the development of FCL can be divided into following aspects:

- Parsing and type analysis;
- Designing a model of static properties of programs;
- Designing and grafting FCL onto the model.

Parsing and type analysis generally have been well understood; thus they are largely a matter of engineering though it can be challenging to get them right. To get a model of static properties of programs demands a deep understanding of the semantics of the subject language. I believe our experience with C++ can be helpful to dealing with Java and oo Perl. Last, it is clear that FCL can be divided into domain independent part, that is, the first-order logic and set and sequence operations, and domain dependent part, namely, the term language and the type system. Therefore, it should be straightforward to grafting FCL to models developed for other languages. In fact, it would be an interesting experiment to mold the current FCL into a framework and develop other FCLs as its instances.

Applying FCL It is necessary to apply FCL to more cases and more importantly, to observe how well it works in assisting real programmers.

Another direction is to investigate whether one can use FCL when developing new frameworks. All examples so far are retrofits; how would it look like to apply FCL

when one is designing a new framework? For example, when should framework developers spend time encoding constraints with FCL? In addition to framework-based development, would FCL be helpful to routine software development?

FCL as a basis for other program manipulation tasks such as refactorings [F+99] and other program transformation.

Architectures and frameworks Software architectures are about the gross structures of large software systems [BCK98]. The most important thing that a framework delivers is its software architecture [Joh97]. Many problems of using frameworks seem to come from the lack of understanding of the architectures of the frameworks. There can be many reasons for this state of practice including, for example, the competence of the user. I am particularly interested in documenting framework architecture for the purposes of both human communication and formal analysis.

Applying model checking to framework-based development Model checking has been applied to both software requirements specifications [ABB+96, AG93] and source code [D+97, MPC+02, JF01]. Our previous attempt on model checking frameworks with Spin [Hol91] was not successful [HHS02]. An interesting research question remains: Whether and how can one leverage model checking techniques in framework-based development?

Bibliography

[ABB+96] Richard J. Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *Software Engineering Notes*, 21(6):156–166, November 1996.

[AG93] Joanne M. Atlee and John Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, June 1993.

[AG97] Robert J. Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[AGI98] Robert J. Allen, David Garlan, and James Ivers. Formal modeling and analysis of the HLA component integration standard. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, November 1998.

[AU78] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*, chapter 7: Syntax–Directed Translation. Addison–Wesley, March 1978.

[B+02] Don Batory et al. Achieving extensibility through product–lines and domain–specific languages: A case study. *ACM Transactions on Software Engineering and Methodology*, 11(2):191–214, April 2002.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[BCS00] Don Batory, R. Cardone, and Y. Smaragdakis. Object oriented frameworks and product lines. In *Proceedings of the First Software Product Line Conference*, Denver, Colorado, August 28–31 2000.

[BMMB98] Jan Bosch, Peter Molin, Michael Mattsson, and PerOlof Bengtsson. Obstacles in object-oriented framework-based software development. *ACM Computing Survey's Symposia on Object Oriented Application Frameworks*, 1998.

[Bok99] Boris Bokowski. CoffeeStrainer: Statically–checked constraints on the definition and uses of types in Java. In *Proceedings of European Software Engineering Conference/Foundation of Software Engineering*, Toulouse, France, September 6–10 1999.

[BR89] Ted J. Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, volume 2, pages 1–17. ACM Press, 1989.

[Bro87] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, pages 10–19, April 1987.

[BS98] Boris Bokowski and Andre Spiegel. Barat: A front–end for Java. Technical Report B–98–09, Freie University Berlin, December 1998.

[C+93] Roy H. Campbell et al. Designing and implementing Choices: An object–oriented system in C++. *Comm. of the ACM*, 36(9):117–126, September 1993.

[C+00a] Andy Chou et al. Using meta–level compilation to check FLASH protocol code. In *Proceedings of ASPLOS 2000*, 2000.

[C+00b] James R. Cordy et al. The TXL programming language, version 10. Kingston, Ontario, Canada, January 2000.

[CEH+94] Mariano P. Consens, Frank Ch. Eigler, Masum Z. Hasan, Alberto O. Mendelzon, Emanuel G. Noik, Arthur G. Ryman, and Dimitra Vista. Architecture and applications of the Hy+ visualization system. *IBM Systems Journal*, 33(3):458–476, 1994.

[CGK98] Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios. A C++ data model supporting reachability analysis and dead code detection. *IEEE Transactions on Software Engineering*, 24(9):682–693, September 1998.

[CHE02] Benjamin Chelf, Seth Hallem, and Dawson R. Engler. How to write system–specific, static checkers in Metal. In *Proceedings of PASTE 2002*, 2002. Invited paper.

[CNR90] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[Cod70] E. F. Codd. A relation model of data for large shared data banks. *Comm. of the ACM*, 13(6), June 1970.

[Cre97] Roger Crew. ASTLOG: A language for examining abstract syntax tree. In *Proceedings of the USENIX Conference on Domain–Specific Languages*, pages 229–242, October 1997.

[D+97] Mathew Dwyer et al. Model checking graphical user interfaces using abstractions. In *LNCS 1301, Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 244–261, September 1997.

[D+03] Premkumar Devanbu et al. Modularity in the new millenium: A panel summary. In *Proceedings of the 25th International Conference on Software Engineering*, pages 723–724, May 2003.

[DCMS02] Thomas R. Dean, James R. Cordy, Andrew J. Malton, and K. A. Schneider. Grammar programming in TXL. In *Proceedings of the 2nd International Workshop on Source Code Analysis and Manipulation (SCAM 02)*. IEEE, October 2002.

[Deu89] L. Peter Deutsch. Design reuse and frameworks in the smalltalk–80 system. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, volume 2, pages 57–71. ACM Press, 1989.

[Dev99] Premkumar T. Devanbu. GENOA–a customizable, front–end retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology*, 8(2):177–212, April 1999.

[DF01] Robert DeLine and Manuel Fahndrich. Enforcing high–level protocols in low–level software. In *Proceedings of PLDI 2001*, Snowbird, Utah, USA, June 1 2001.

[Dij68] Edsger W. Dijkstra. The structure of the "THE"–multiprogramming system. *Comm. of the ACM*, 11(5), May 1968.

[Dij70] Edsger W. Dijkstra. Notes on structured programming, second edition. Technical Report 70-WSK-03, Department of Mathematics, Technological University Eindhoven, April 1970. EWD 249.

[DiL95] Paul DiLascia. Meandering through the maze of mfc message and command routing. *Microsoft System Journal*, July 1995.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, December 18 1998.

[DRW96] Premkumar T. Devanbu, David S. Rosenblum, and Alexander L. Wolf. Generating testing and analysis tools with Aria. *ACM Transactions on Software Engineering and Methodology*, 5(1):42–62, January 1996.

[E+] David Evans et al. Splint manual. Available at http://lclint.cs.virginia.edu.

[E+94] David Evans et al. LCLint: A tool for using specifications to check code. In *Proceedings of FSE'94*, New Orleans, LA, USA, December 1994.

[E+00] Dawson R. Engler et al. Checking system rules using system specific, programmer–writtern compiler extensions. In *Proceedings of OSDI 2000*, 2000.

[Eva96] David Evans. Static detection of dynamic memory errors. In *Proceedings of PLDI 96*, Philadelphia, PA, USA, May 1996.

[F+99] Martin Fowler et al. *Refactorings–Improving the Design of Existing Software.* Addison Wesley, 1999.

[FBML00] Marcus F. Fontoura, Christiano Braga, Leonardo Moura, and Carlos J. Lucena. Using domain specific languages to instantiate object-oriented frameworks. *IEE Proceedings–Software*, 147(4), 2000.

[FHLS97] Garry Froehlich, Jim Hoover, Ling Liu, and Paul Sorenson. Hooking into object-oriented application frameworks. In *Proceedings of the 1997 International Conference on Software Engineering*, Boston, Mass., May 1997.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Ramie Stata. Extended static checking for Java. In *Proceedings of PLDI'02*, Berlin, Germany, June 17–19 2002.

[Fro02] Garry Froehlich. *Hooks: Aiding the Use of O–O Frameworks.* PhD thesis, University of Alberta, Edmonton, Alberta, Canada, September 2002.

[FS97] Mohamed Fayad and Douglas C. Schmidt. Special issue on object–oriented application frameworks, October 1997. Comm. of the ACM.

[FSH01] Rudolf Ferenc, Susan Elliott Sim, and Richard C. Holt. Towards a standard schema for C/C++. In *WCRE 2001: Working Conference on Reverse Engineering*, Stuttgart, Germany, Oct. 2–5 2001.

[GAM96] William G. Griswold, Darren C. Atkinson, and Collin McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proceedings of IEEE Workshop on Program Comprehension 1996*, Berlin, Germany, March 29–31 1996.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Springer–Verlag, 1993.

[GHJV94] Erich Gamma, Richard Helm, Ralph E. Johnson, and John O. Vlissides. *Design Patterns-Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson R. Engler. A system and language for building system–specific, static analyses. In *Proceedings of OSDI 2002*, Berlin, Germany, June 17–19 2002.

[HFC76] A. Nico Habermann, Lawrence Flon, and Lee Cooprider. Modularization and hierarchy in a family of operating systems. *Comm. of the ACM*, 19(5), May 1976.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings of ECOOP/OOPSLA 90*, Ottawa, Canada, 1990.

[HHL+00] Richard C. Holt, Ahmed E. Hassan, Bruno Lague, Sebastien Lapierre, and Charles Leduc. E/R schema for the Datrix C/C++/Java exchange format. In *Proceedings of Working Conference on Reverse Engineering*, pages 349 – 358, 2000.

[HHR03] Daqing Hou, H. James Hoover, and Piotr Rudnicki. dxlinker: Generating real C++ ASGs. Technical Report TR03-22, Department of Computing Science, University of Alberta, December 26 2003.

[HHS02] Daqing Hou, H. James Hoover, and Eleni Stroulia. Supporting the deployment of object–oriented frameworks. In *Proceedings of the International Conference on the Advanced Information System Engineering (CAiSE'02)*, Toronto, ON, Canada, May 2002.

[HHY03] Daqing Hou, H. James Hoover, and Changyu Yin. The framework use problem: A preliminary study with GUI frameworks. In *Proceedings of the First Midwest Software Engineering Conference*, DePaul University, Chicago, IL, USA, May 2003.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hol91] Gerald Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Hol97] Richard C. Holt. An introduction to TA: the tuple-attribute language. Available at http://plg.uwaterloo.ca/~holt/papers/ta.html, 1997.

[Hol98] Richard C. Holt. Structural manipulation of software architecture using Tarski relational algebra. In *WCRE 1998: Working Conference on Reverse Engineering*, Honolulu, Hawaii, USA, Oct. 12–14 1998.

[Hol02] Richard C. Holt. Introduction to the Grok language. Available at http://plg.uwaterloo.ca/~holt/papers/grok-intro.html, May 5 2002.

[HWW02] Richard C. Holt, Andreas Winter, and Jingwei Wu. Towards a common query language for reverse engineering. Technical report, Institute for Computer Science, University Koblenz–Landau, August 2002.

[IBM] IBM. VisualAge for Java. 1991–1999. Available at http://software.ibm.com/software/ad/vajava.

[Inc89] Apple Computer Inc. Macappii programmer's guide, 1989.

[Int98] International Standards Organization (ISO). *Programming languages – C++*. ISO/IEC 14882:1998(E), September 1998.

[Jac95] Daniel Jackson. Aspect: Detecting bugs with abstract dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

[Jac99] Daniel Jackson. A comparison of object modeling notations: Alloy, UML and Z. MIT Laboratory for Computer Science, August 11 1999.

[Jac02] Daniel Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JF01] Daniel Jackson and A. Fekete. Lightweight analysis of object interactions. In *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, October 2001.

[Joh92] Ralph E. Johnson. Documenting frameworks with patterns. In *Proceedings of OOPSLA 92*, Vancouver, Canada, 1992.

[Joh97] Ralph E. Johnson. Components, frameworks, patterns. In *Proceedings of the 1997 Symposium on Software Reusability*, pages 10–17, Boston, United States, May 17–20 1997.

[JW96] Daniel Jackson and Jennette Wing. Lightweight formal methods. *IEEE Computer*, 29(4):16–30, April 1996.

[KL92] Gregory Kiczales and John Lamping. Issues in the design and specification of class libraries. In *Proceedings of OOPSLA 92*, pages 435–451, Vancouver, Canada, 1992.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Programming*, 1(3), August-September 1988.

[Kru92] Charles W. Krueger. Software reuse. *ACM Computing Survey*, 24(6):131–183, 1992.

[Lam93] John Lamping. Typing the specialization interface. In *Proceedings of OOPSLA 93*, pages 201–215, 1993.

[LH89] Karl J. Lieberherr and Ian Holland. Assuring good style for object–oriented programs. *IEEE Software*, pages 38–48, September 1989.

[Lin84] M. A. Linton. Implementing relational views of programs. In *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment*, May 1984.

[LM93] Yueh-Hong Lin and Scott Meyers. CCEL: The C++ constraint expression language–an annotated reference manual (version 0.5). Technical Report CS–93–23, Department of Computer Science, Brown University, 1993.

[LN02] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.

[LNS00] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Report 2000–002, Compaq Systems Research Center, October 2000.

[LR95] David A. Ladd and J. Christopher Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, November 1995.

[LW94] Barbara H. Liskov and Jannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

[MDH01] Andrew J. Malton, Thomas Dean, and Richard C. Holt. Union schemas as the basis for a C++ extractor. In *WCRE 2001: Working Conference on Reverse Engineering*, pages 59 – 67, Stuttgart, Germany, Oct. 2–5 2001.

[MDR93] Scott Meyers, Carolyn K. Duby, and Steven P. Reiss. Constraining the structure and style of object–oriented programs. Technical Report CS–93–12, Department of Computer Science, Brown University, 1993.

[ME03] Madanlal Musuvathi and Dawson R. Engler. Some lessons from using static analysis and software model checking for bug finding. In *Proceedings of SoftMC'03*, Boulder, CO, USA, July 18–19 2003. Invited paper.

[Mey88] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. International Series in Computer Science, Prentice Hall, 1988.

[Mey92a] Bertrand Meyer. Applying design by contract. *IEEE Computer*, October 1992.

[Mey92b] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison–Wesley Publishing Company, 1992.

[Mey96] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison–Wesley Publishing Company, 1996.

[Mic] Microsoft. Visual Studio. Available at http://msdn.microsoft.com/vstudio/.

[MK97] Scott Meyers and Martin Klaus. A first look at C++ program analyzers. *Dr. Dobb's Journal*, February 1997.

[ML] Ian Moon and Jie Li. Private communication (Nov. and Dec. 2003): Ian is using and extending the Prothos framework at the University of Alberta; Jie is using the Jakarta Struts framework, an open-source implementation of the J2EE standard.

[MMW02] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Expert System with Applications*, 23:405–413, 2002.

[MN96] Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, July 1996.

[MOTU93] Hausi A. Mueller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem identification. *Software Maintenance and Practice*, 5:181–204, 1993.

[MPC+02] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of OSDI 2002*, December 2002.

[MS95] Alberto Mendelzon and Johannes Sametinger. Reverse engineering by visualizing and querying. *Software–Concepts and Tools*, 16:170–182, 1995.

[New] Joseph Newcomer. FlounderCraft Ltd. Available at http://www.flounder.com.

[Nie95] Oscar Nierstrasz. Regular types for active objects. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[OCM00] Alvaro Ortigosa, Marcelo Campo, and Roberto Moriyon. Towards agent–oriented assistance for framework instantiation. In *Proceedings of OOPSLA 2000*, Minneapolis, MN, USA, October 2000.

[Org] The Mizar Organization. Mizar project. Available at http://www.mizar.org.

[Par01a] David L. Parnas. Designing software for ease of extension and contraction. In Daniel M. Hoffman and David M. Weiss, editors, *Software Fundamentals: Collected Papers by David L. Parnas*, pages 269–286. Addison Wesley, 2001.

[Par01b] David L. Parnas. On a "buzzword": Hierarchical structure. In Daniel M. Hoffman and David M. Weiss, editors, *Software Fundamentals: Collected Papers by David L. Parnas*, pages 161–168. Addison Wesley, 2001.

[Par01c] David L. Parnas. On the criteria to be used in decomposing systems into modules. In Daniel M. Hoffman and David M. Weiss, editors, *Software Fundamentals: Collected Papers by David L. Parnas*, pages 145–154. Addison Wesley, 2001.

[Par01d] David L. Parnas. On the design and development of program families. In Daniel M. Hoffman and David M. Weiss, editors, *Software Fundamentals: Collected Papers by David L. Parnas*, pages 193–213. Addison Wesley, 2001.

[Par01e] David L. Parnas. Predicate logic for software engineering. In Daniel M. Hoffman and David M. Weiss, editors, *Software Fundamentals: Collected Papers by David L. Parnas*, pages 51–63. Addison Wesley, 2001.

[PP94a] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, June 1994.

[PP94b] Santanu Paul and Atul Prakash. Supporting queries on source code: A formal framework. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):325–348, September 1994.

[PP96] Santanu Paul and Atul Prakash. A query algebra for program databases. *IEEE Transactions on Software Engineering*, 22(3):202–217, March 1996.

[PPSS95] Wolfgang Pree, Gustav Pomberger, Albert Schappert, and Peter Sommerlad. Active guidance of framework development. *Software–Concepts and Tools*, 16:94–103, 1995.

[Pre95] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development.* Addison Wesley, 1995.

[RW88] Charles Rich and Richard C. Waters. The programmer's apprentice: A research overview. *IEEE Computer*, 21(11):10–25, 1988.

[RW91] David Rosenblum and Alexander Wolf. Representing semantically analyzed C++ code with Reprise. In *Proceedings of the Third C++ Technical Conference*, pages 119–134, Berkeley, CA, 1991. USENIX Assoc.

[Ryd03] Barbara G. Ryder. Dimensions of precision in reference analysis of object–oriented programming languages. In *Proceedings of ETAPS'03*, April 2003. Invited talk.

[SB02] Yannis Smaragdakis and Don Batory. Mixin layers: An object–oritented implementation technique for refinements and collaboration–based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, April 2002.

[Sef96] Mohlalefi Sefika. *Design Conformance Management of Software Systems: An Architecture–Oriented Approach.* PhD thesis, University of Illinois at Urbana–Champaign, Urbana, Illinois, USA, 1996.

[SEH03] Susan Elliot Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of ICSE 2003*, Portland, USA, May 2003.

[SG95] Raymie Stata and John V. Guttag. Modular reasoning in the presence of subclassing. In *Proceedings of OOPSLA 95*, pages 200–214, 1995.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, April 1996.

[SLMD95] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of OOPSLA 96*, pages 268–285, 1995.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual.* Second edition, Prentice Hall, 1992.

[SSC96a] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Architecture–oriented visualization. In *Proceedings of OOPSLA 1996*, 1996.

[SSC96b] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring the compliance of a software system with its high level design models. In *Proceedings of ICSE 1996*, pages 387–396, 1996.

[Str99] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, Reading, MA, 1999. Third Edition.

[SW96] George Shepherd and Scot Wingo. *MFC Internals: Inside the Microsoft Foundation Classes Architecture.* Addison Wesley, 1996.

[UW97] Jeffrey D. Ullman and Jenniffer Widom. *A First Course in Database Systems.* Prentice Hall, 1997.

[VJ99] Mandana Vaziri and Daniel Jackson. Some shortcomings of OCL, the object constraint language of UML. a response to OMG's request for Information on UML 2.0, MIT Laboratory for Computer Science, December 7 1999.

[VL89] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain–specific graphical editors. In *Proceedings of the ACM User Interface Software and Technologies'89 Conference*, November 1989.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, 1999.

[WKR02] Andreas Winter, Bernt Kullbach, and Volker Riediger. An overview of the GXL graph exchange language. *Software Visualization, LNCS 2269*, pages 324–336, 2002.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product–line Engineering: A Family–based Software Development Process*. Addison Wesley, 1999.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object–oriented systems. In *Proceedings of TOOLS–USA 98*, Santa Barbara, CA, August 3–7 1998. IEEE.

Appendix A
FCL: Abstract Syntax and Semantics

This appendix introduces the abstract syntax (Table 9.1), type rules (static semantics), and the dynamic semantics of FCL. Meyer's notation [Mey88] is adopted. The concrete syntax can be found in Chapter 3.

A.1 Notational Conventions

The denotational method specifies the meaning of a programming language through associating with each construct T two functions of the following general forms:

$$V_T : T \rightarrow \mathsf{Bool}$$
$$M_T : T \rightarrow D_T$$

where T can be, for instance, *Statement*, *Declaration*, *Formula*, and so forth.

The function V_T is a predicate. It yields the value *true* if and only if its argument is a valid instance of the program construct T. The sets of constructs are called *syntactic domains*. The set of validity functions for a language defines all the "constraints" to which its programs have to conform. They form the *static semantics* of the language.

For each construct T, the function M_T denotes its dynamic semantics with a set of mathematical objects, which is also called its *denotation*. For each T, D_T represents the set of mathematical objects; the set may vary for the various constructs. The set of denotations for all the constructs of a language defines the *semantic domains* of the language.

Most of the M functions will turn out to be "higher order" functions that yield functions as results. To highlight the specific nature of these functions, their arguments will be enclosed in square brackets rather than the ordinary parentheses, as in $M_T[t]$.

The following common mathematical operators will also be used:

1. $indexing : Seq\ T \times Int \rightarrow T$: get the ith element of a sequence. Syntactically, it is written as *s(i)*, where s is the sequence and i the index.

2. $tail : Seq\ T \rightarrow Seq\ T$: get a new sequence from the argument by removing its first element.

3. $add\_to\_head : T \times Seq\ T \rightarrow Seq\ T$: get a new sequence whose first element is the element argument and whose tail is the sequence argument.

4. $\biguplus$: "overriding union" of two functions f, g: $\mathbf{X} \rightarrow \mathbf{Y}$:

$$\begin{aligned}
&h \equiv f \biguplus g : \\
&\mathbf{dom}\ h = \mathbf{dom}\ f \cup \mathbf{dom}\ g; \\
&h(x) = f(x) \text{ if } x \in \mathbf{dom}\ f \text{ and } x \notin \mathbf{dom}\ g; \\
&h(x) = g(x) \text{ if } x \in \mathbf{dom}\ g.
\end{aligned}$$

These notational conventions are adopted from Meyer's book [Mey88].

A.2 Static Semantics

FCL is strongly typed so that when evaluating an FCL specification, an FCL checker will not suffer any run-time errors. This is ensured by putting a set of constraints on the structure of FCL specifications. The constraints comprise the static semantics of FCL.

From Object Model to Type System

An object model presents the entity-relationship model within a certain domain; the classes are entities, and the relationships are modeled by the operations that the classes support. Given such a model, one can design a logical language for it by assigning a *sort* to each entity and a *total function* to each operation supported by the entity. An inheritance relation between a pair of classes can be translated into a *subtype/subset* relation between their corresponding sorts.

However, sometimes it may not be necessary to assign sorts for all classes. Fig. 4.1 depicts the object model for C++, and Fig. 9.1 the basic types and the subtype relation for

FCL_spec	::=	sl: Statement_list
Statement	::=	Declaration | Formula
Declaration	::=	v: Variable; expr: Expr
Formula	::=	Negation | Conjunction | Disjunction |
		Existential | Universal | Expr
Expr	::=	Variable | Constant | Operation | Formula | ExprWithVars
Negation	::=	f: Formula
Conjunction	::=	f_1: Formula; f_2: Formula
Disjunction	::=	f_1: Formula; f_2: Formula
Existential	::=	bVarl: BVar_Declaration_list; f: Formula
Universal	::=	bVarl: BVar_Declaration_list; f: Formula
BVar_Declaration	::=	Declaration
ExprWithVars	::=	vl: Declaration_list; expr: Expr
Operation	::=	Set_op | Seq_op | Relational | FCL_fct
Set_op	::=	Subset | Memberof | union | difference | family_union |
		card | Set_comprehension | Set_enumeration
Set_comprehension	::=	bVarl: BVar_Declaration_list; p: Formula; ele: Expr
Set_enumeration	::=	exprl: Expr_list
Seq_op	::=	Seq_member | Seq_indexOf
Relational	::=	> | >= | < | <= | =
Constant	::=	true | false | Str | Int | global
Variable	::=	id: Str
T_list	::=	T*

Table 9.1: The Abstract Syntax of FCL

the current version of FCL. Upon comparing them, one may find that, for instance, the class cAsgNode and the class cTyped have no corresponding sorts in FCL. This is because we have not found examples where assigning sorts for them would be necessary. If the operations supported by these classes are useful, we can move them downwards to the corresponding sorts of their subclasses. For example, retrieving the context of an identifier is defined as an operation for the class cAsgNode, but since the class is not modeled in FCL, a function unit (Table 3.2) has to be defined on two sorts, Name and Exp, which correspond to two of its subclasses, cName and cExpGeneralized, respectively.

Type System of Current FCL

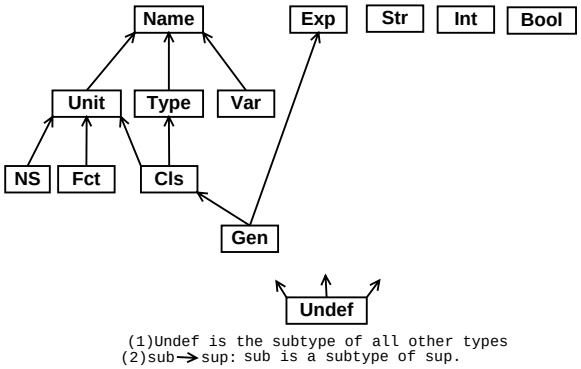

Figure 9.1: Basic types and the subtype relation

The type system of FCL includes two kinds of types: basic types (Fig. 9.1) and compound types. Basic types can be further divided into "facility" types and "domain" types. "Facility" types help form constraints, including Str for string values, Int for integers, and Bool for boolean values. "Domain" types come from the problem domain of programming constructs, including Exp for expressions, Var for variables, NS for namespaces, Fct for functions, Cls for classes, Name for named entities, Unit for program units, Gen for types generated through template instantiations, and Type for types.

The current FCL covers only a subset of the object model; for instance, pointer and reference types, arrays, builtin types, and generated functions are not included.

The set of basic types can be represented by Type_value_basic defined as follows:

Type_value_basic ≡ { Str, Int, Bool, Exp, Var, NS, Fct, Cls, Name, Unit, Gen, Type, Undef }

Compound types are types for sequences and sets:

Seq_type ≡ { <t> | t : {Var, Exp, Unit}}, where <t> represents the set of sequences the type of whose elements is t.

Set_type ≡ { **F** t | t:Type_value_basic ∪ Set_type }, where **F** t represents the power set of the type t.

Given a type t, the function build_set_type creates a new type, which is the set of t: build_set_type(t: Type_value_basic ∪ Set_type) = **F** t.

Given a set type of the form **F** t, the function elementType returns its element type: elementType(**F** t) = t.

Undef is the type for the undefined value.

Fig. 9.1 also defines the *subtype* relation between basic types. Besides basic types, subtype relations can also exist between compound types. One set type is the subtype of another if and only if the base type of the former is the subtype of the latter. Similar definition holds for sequence types.

The subtype relation is necessary for the set union operation; For a set union expression to be valid, FCL requires that the base types of both its operand sets have one and only one common ancestor in Fig. 9.1. This notion will be further formalized in section A.3.

Finally, the set of types that FCL supports, Type_value, is defined as follows:

Type_value ≡ Type_value_basic ∪ Set_type ∪ Seq_type

Given a type, the predicate is_Set can tell whether it is a set type.

Type Maps

Whenever one says that an instance of some construct is "correct," one really means that it is valid in a certain context, in which each identifier has a designated type. In compiler design, this notion of context corresponds to "symbol tables", which can be modeled by a function $Type\_map$:

$$Type\_map : Variable \rightarrow Type\_value$$

FCL variables are only *implicitly* typed; that is, one does not declare a variable in the form of "*v: T*", where T is a named type. Instead, one writes "*v: Expr*", where the type of *Expr* can be inferred, and the type of v is assigned as that of *Expr*. In the following, we will use the function *type_val*, which evaluates the type of an expression:

$$type\_val : Expr \times Type\_map \rightarrow Type\_value$$

whose definition will be deferred to the end of this section.

For a construct T in the context of a type map, the function $typing_T$ constructs a type map for the new variables defined in T:

$$typing_T : T \times Type\_map \rightarrow Type\_map$$

where T can be *Declaration*, *Declaration_list*, *BVar_Declaration*, *BVar_Declaration_list*, or *Statement*,

A declaration yields a type map consisting of only one pair, whose name is that of the variable, and whose type is that of the expression:

$$typing_{Declaration}[dcl : Declaration, tm : Type\_map] \equiv \{ < dcl.v, type\_val[dcl.expr, tm] > \}$$

The type map that a list of variable declarations introduces is the "overriding union" of the type maps of each of the variables, from head to tail:

$$\begin{array}{l}
typing_{Declaration_list}[vl : Declaration_list, tm : Type_map] \equiv \\
\quad \text{if } vl.length = 0 \text{ then } \emptyset \\
\quad \text{else} \\
\qquad \text{given } firstTM = typing_{Declaration}[vl(0), tm] \\
\qquad \text{then} \\
\qquad\quad firstTM \uplus typing_{Declaration_list}[vl.tail, newTM] \\
\qquad \text{end} \\
\quad \text{end}
\end{array}$$

Both quantification formulas and set comprehension introduce a list of bound variables. The type map for the list is the "overriding union" of the type maps of all the variables, from left to right:

$$\begin{array}{l}
typing_{BVar_Declaration_list}[bVarl : BVar_Declaration_list, tm : Type_map] \equiv \\
\quad \text{if } bVarl.length = 0 \text{ then } \emptyset \\
\quad \text{else} \\
\qquad \text{given } firstTM = typing_{BVar_Declaration}[bVarl(0), tm] \\
\qquad \text{then} \\
\qquad\quad firstTM \uplus typing_{BVar_Declaration_list}[bVarl.tail, newTM] \\
\qquad \text{end} \\
\quad \text{end}
\end{array}$$

The type map of each bound variable is defined as follows:

$$\begin{array}{l}
typing_{BVar_Declaration}[dcl : Declaration, tm : Type_map] \equiv \\
\quad \text{given } setType = type_val[dcl.expr, tm]; \\
\qquad setElementType = elementType(setType); \\
\quad \text{then} \\
\qquad \{ < dcl.v, setElementType > \} \\
\quad \text{end}
\end{array}$$

Note that the type of a bound variable is the type of the elements of the set whereas the type of a declared variable is that of its expression. For example, in c as classSet;, c has the type of classSet whereas in forall c being classSet holds ..., c has the type of the element type of classSet.

The following is the *typing* function for statements:

$$
\begin{array}{l}
typing_{Statement}[st : Statement, tm : Type\_map] \equiv \\
\quad \text{case } st \text{ of} \\
\quad\quad Declaration : typing_{Declaration}[st, tm] \\
\quad\quad Formula : \emptyset \\
\quad \text{end}
\end{array}
$$

If the statement is a formula, it does not introduce any new variables that can be used by subsequent statements, therefore, the function returns the empty set; Otherwise, it is a declaration and the function returns the new type map defined by the declaration.

An outline of the function *type_val* is as follows:

$$
\begin{array}{l}
type\_val : Expr \times Type\_map \rightarrow Type\_value \\
\\
type\_val[exp : Expr, tm : Type\_map] \equiv \\
\quad \text{case } exp \text{ of} \\
\quad\quad Variable : tm(exp) \\
\quad\quad Constant : type\_val_{Constant}[exp] \\
\quad\quad Formula : \mathsf{Bool} \\
\quad\quad Operation : type\_val_{Operation}[exp, tm] \\
\quad\quad ExprWithVars : type\_val[exp.expr, tm \uplus typing_{Declaration\_list}[exp.vl, tm]\ ] \\
\quad \text{end}
\end{array}
$$

A.3 Static Semantic Functions

An FCL specification consists of a sequence of statements. The specification is valid if and only if the statements list is valid:

$$
\begin{array}{l}
V_{FCL\_spec} : FCL\_spec \rightarrow \mathsf{Bool} \\
V_{FCL\_spec}[p : FCL\_spec] \equiv V_{Statement\_list}[p.sl, \emptyset]
\end{array}
$$

A statement list is valid if and only if all of its statements are valid:

$$
V_{Statement\_list} : Statement\_list \times Type\_map \rightarrow \mathsf{Bool}
$$

$$
\begin{array}{l}
V_{Statement\_list}[sl : Statement\_list, tm : Type\_map] \equiv \\
\quad \text{if } sl.length = 0 \text{ then } true \\
\quad \text{else } V_{Statement}[sl(0), tm] \wedge \\
\quad\quad V_{Statement\_list}[sl.tail, tm \uplus typing[sl(0), tm]] \\
\quad \text{end}
\end{array}
$$

Note how the type map for subsequent statements is updated in the above function.

$$
V_{Statement} : Statement \times Type\_map \rightarrow \mathsf{Bool}
$$

$$
\begin{array}{l}
V_{Statement}[st : Statement, tm : Type\_map] \equiv \\
\quad \text{case } st \text{ of} \\
\quad\quad Declaration : V_{Declaration}[st, tm] \\
\quad\quad Formula : V_{Formula}[st, tm] \\
\quad \text{end}
\end{array}
$$

The validity of a declaration is equivalent to the validity of its expression:

$$
V_{Declaration} : Declaration \times Type\_map \rightarrow \mathsf{Bool}
$$

$$
V_{Declaration}[dcl : Declaration, tm : Type\_map] \equiv V_{Expr}[dcl.expr, tm]
$$

The validity of an expression is the result of following case analyses:

$$
\begin{array}{l}
V_{Expr} : Expr \times Type\_map \rightarrow \mathsf{Bool} \\
V_{Expr}[exp : Expr, tm : Type\_map] \equiv \\
\quad \text{case } exp \text{ of} \\
\qquad Variable : V_{Variable}[exp, tm] \\
\qquad Constant : true \\
\qquad Formula : V_{Formula}[exp, tm] \\
\qquad ExprWithVars : V_{ExprWithVars}[exp, tm] \\
\qquad Set\_comprehension : V_{Set\_comprehension}[exp, tm] \\
\qquad Operation : V_{Operation}[exp, tm] \\
\quad \text{end}
\end{array}
$$

Of course, a name reference is valid if and only if there is a variable with the same name in the type map:

$$V_{Variable} : Variable \times Type\_map \rightarrow \mathsf{Bool}$$

$$V_{Variable}[v : Variable, tm : Type\_map] \equiv v \in \text{ dom } tm$$

An expression can have a list of local variables. Such an expression is valid if and only if its expression is valid under the new type map:

$$V_{ExprWithVars} : ExprWithVars \times Type\_map \rightarrow \mathsf{Bool}$$

$$
\begin{array}{l}
V_{ExprWithVars}[exp : ExprWithVars, tm : Type\_map] \equiv \\
\quad \text{given } newTM = tm \uplus typing[exp.vl, tm] \\
\quad \text{then} \\
\qquad V_{Expr}[exp.expr, newTM] \\
\quad \text{end}
\end{array}
$$

The validity of formulas are also the result of case analyses:

$$V_{Formula} : Formula \times Type\_map \rightarrow \mathsf{Bool}$$

$$
\begin{array}{l}
V_{Formula}[f : Formula, tm : Type\_map] \equiv \\
\quad \text{case } f \text{ of} \\
\qquad Negation : V_{Negation}[f, tm] \\
\qquad Conjunction : V_{Conjunction}[f, tm] \\
\qquad Disjunction : V_{Disjunction}[f, tm] \\
\qquad Existential : V_{Existential}[f, tm] \\
\qquad Universal : V_{Universal}[f, tm] \\
\qquad Expr : V_{Expr}[f, tm] \wedge type\_val[f, tm] = \mathsf{Bool} \\
\quad \text{end}
\end{array}
$$

Note that for an expression to be a formula, it not only has to be a valid expression, but also has to be of the Bool type.

A negation, conjunction, or disjunction is valid if and only if their constituent formulas are valid; we omit the validity functions since they are obvious.

The validity functions of both universal and existential quantifications are the same, the following only presents the one for existentials:

$$V_{Existential} : Existential \times Type\_map \rightarrow \mathsf{Bool}$$

$$
\begin{array}{l}
V_{Existential}[ext : Existential, tm : Type\_map] \equiv \\
\quad V_{BVar\_Declaration\_list}[ext.bVarl, tm] \wedge \\
\quad V_{Formula}[ext.f, tm \uplus typing[ext.bVarl, tm]]
\end{array}
$$

A set comprehension is valid if and only if all of its three parts, the list of bound variables, the "filter" predicate, and the "element" function, are valid:

$$V_{Set\_comprehension} : Set\_comprehension \times Type\_map \rightarrow \mathsf{Bool}$$

$$\begin{array}{l}
V_{Set_comprehension}[sc : Set_comprehension, tm : Type_map] \equiv \\
\quad \text{if } V_{BVar_Declaration_list}[sc.bVarl, tm] \text{ then} \\
\quad\quad \text{given } newTM = tm \uplus typing[sc.bVarl, tm] \\
\quad\quad \text{then} \\
\quad\quad\quad V_{Formula}[sc.p, newTM] \wedge \\
\quad\quad\quad V_{Expr}[sc.ele, newTM] \\
\quad\quad \text{end} \\
\quad \text{else } false \\
\quad \text{end}
\end{array}$$

The type of set comprehension is calculated as follows:

$$type\_val_{Set\_comprehension} : Set\_comprehension \times Type\_map \rightarrow \mathbf{Set\_type}$$

$$\begin{array}{l}
type_val_{Set_comprehension}[sc : Set_comprehension, tm : Type_map] \equiv \\
\quad \text{given } newTM = tm \uplus typing[sc.bVarl, tm]; \\
\quad\quad elementType = type_val_{Expr}[sc.ele, newTM] \\
\quad \text{then} \\
\quad\quad build_set_type(elementType) \\
\quad \text{end}
\end{array}$$

For a bound variable list to be valid, all of its elements have to valid:

$$V_{BVar\_Declaration\_list} : BVar\_Declaration\_list \times Type\_map \rightarrow \mathsf{Bool}$$

$$\begin{array}{l}
V_{BVar_Declaration_list}[bVarl : BVar_Declaration_list, tm : Type_map] \equiv \\
\quad \text{if } bVarl.length = 0 \text{ then } true \\
\quad \text{else} \\
\quad\quad V_{BVar_Declaration}[bVarl(0), tm] \wedge \\
\quad\quad V_{BVar_Declaration_list}[bVarl.tail, tm \uplus typing[bVar(0), tm]] \\
\quad \text{end}
\end{array}$$

The definition of a bound variable is valid if and only if its set expression is valid and the set expression is of set types:

$$V_{BVar\_Declaration} : Declaration \times Type\_map \rightarrow \mathsf{Bool}$$

$$\begin{array}{l}
V_{BVar_Declaration}[bVar : Declaration, tm : Type_map] \equiv \\
\quad \text{given } exprType = type_val[bVar.expr, tm] \\
\quad \text{then} \\
\quad\quad V_{Expr}[bVar.expr, tm] \wedge is_Set(exprType) \\
\quad \text{end}
\end{array}$$

Before discussing the typing rules for set operations union and set enumeration, the *lub* (least upper bound) operator for two types has to be defined first. For two types t_1 and t_2,

$$\begin{array}{l}
lub : Type_value \times Type_value \rightarrow Type_value \\
lub(t_1, t_2) \equiv \\
\quad \text{if } \exists t : (t \geq t_1 \wedge t \geq t_2 \wedge \nexists t' : t' < t \wedge t' \geq t_1 \wedge t' \geq t_2) \\
\quad \text{then } t \text{ else } null \\
\quad \text{end}
\end{array}$$

where $t_1 < t_2$ denotes that t_1 is a subtype of t_2. The subtype relation is introduced in section A.2.

Now the validity function for set union can be defined as follows:

$$
\begin{array}{l}
V_{union} : union \times Type\_map \rightarrow \text{Bool} \\
V_{union}[u : union, tm : Type\_map] \equiv \\
\quad \text{given } t_1 = type\_val[u.l, tm]; t_2 = type\_val[u.r, tm]; \\
\quad \text{then} \\
\quad\quad V_{Expr}[u.l, tm] \wedge V_{Expr}[u.r, tm] \wedge \\
\quad\quad is\_Set(t_1) \wedge is\_Set(t_2) \wedge lub(t_1, t_2) \neq null \\
\quad \text{end}
\end{array}
$$

And that for set enumeration can be defined as follows:

$$
\begin{array}{l}
V_{Set\_enumeration} : Set\_enumeration \times Type\_map \rightarrow \text{Bool} \\
V_{Set\_enumeration}[se : Set\_enumeration, tm : Type\_map] \equiv \\
\quad \text{if } se.exprl.length = 0 \text{ then } true \\
\quad \text{else given } eType = elementType(type\_val_{Set\_enumeration}[se.exprl.tail, tm]) \\
\quad\quad V_{Expr}[se.exprl(0), tm] \wedge V_{Set\_enumeration}[se.exprl.tail, tm] \wedge \\
\quad\quad lub(type\_val[se.exprl(0), tm], eType) \neq null \\
\quad \text{end}
\end{array}
$$

$$
\begin{array}{l}
type\_val_{Set\_enumeration} : Set\_enumeration \times Type\_map \rightarrow \text{Set\_type} \\
type\_val_{Set\_enumeration}[se : Set\_enumeration, tm : Type\_map] \equiv \\
\quad \text{if } se.exprl.length = 0 \text{ then } build\_set\_type(null) \\
\quad \text{else given } tailType = elementType(type\_val_{Set\_enumeration}[se.exprl.tail, tm]) \\
\quad\quad build\_set\_type(lub(type\_val_{Expr}[se.exprl(0), tm], tailType)) \\
\quad \text{end}
\end{array}
$$

A.4 Dynamic Semantics

Semantic Domains

The dynamic semantic domain of FCL, $Value$, is defined as follows:

$$Value \equiv \{ * \} \cup \bigcup Type\_value$$

* is a special symbol for the "undefined" value. Each element of the set Type_value, if viewed by *intent*, represents a type; if, however, viewed by *extent*, it represents the set of elements that are characterized by the type. In the definition of $Value$, types are viewed from the *extent* point of view.

At the top level, a program can be denoted as a function that maps from names to namespaces:

$$Program \equiv Str \rightarrow NS$$

It has only one pair within it, that is, the global namespace:

$$Program = \{\langle global, the\_global\_namespace \rangle\}$$

At the next level, each namespace can contain zero or more nested namespaces, types, variables, and functions. Thus NS can be characterized as follows:

$$
\begin{array}{l}
NS \equiv ns : Str \rightarrow NS; type : Str \rightarrow Type; \\
\quad var : Str \rightarrow Var; fct : Str \rightarrow \mathbf{F}\ Fct
\end{array}
$$

That is, a namespace can be formalized as an aggregate of functions. Note that we do not intend to use this equation as a definition of NS, for otherwise this would be a recursive domain equation that has no solution under the usual category of sets. Instead, we intend to use it to characterize a given NS.

In a similar way one can give other constructs denotations. We omit them because formal treatments for these construncts would be quite tedious without increasing the clarity of presentation; instead, we choose to explain them informally in Section 3.2.2.

Dynamic Semantic Functions

The dynamic semantics of FCL specifications are characterized by *State*, a function from FCL variables to the elements of $Value$:

$$State : Variable \rightarrow Value$$

The meaning of the FCL constant *global* is defined as follows:

$$M_{global} : Program \rightarrow State$$
$$M_{global}[p : Program] \equiv p(global)$$

An FCL specification consists of zero or more constraints. Evaluating the specification against a given program yields a sequence of boolean values, one for each constraint. Thus, the meaning of a specification can be defined as follows:

$$M_{FCL\_spec} : FCL\_spec \rightarrow Program \rightarrow \textbf{Seq}\ \mathsf{Bool}$$
$$M_{FCL\_spec}[spec : FCL\_spec] \equiv \lambda p : Program \cdot$$
$$\quad \text{given } startState = \{\langle global, M_{global}[p]\rangle\}$$
$$\quad \text{then } M_{Statement\_list}[spec.sl, startState]$$
$$\quad \text{end}$$

That is, the evaluation of a specification starts with the statement list and the startState. The semantic function for the statement list is defined as follows:

$$M_{Statement\_list} : Statement\_list \times State \rightarrow \textbf{Seq}\ \mathsf{Bool}$$
$$M_{Statement\_list}[sl : Statement\_list, s : State] \equiv$$
$$\quad \text{if } sl.length = 0 \text{ then } \langle\rangle$$
$$\quad \text{else case } sl(0) \text{ of}$$
$$\quad\quad Declaration : M_{Statement\_list}[sl.tail, s \uplus M_{Declaration}[sl(0), s]\ ]$$
$$\quad\quad Formula : add\_to\_head(M_{Formula}[sl(0), s], M_{Statement\_list}[sl.tail, s])$$
$$\quad\quad \text{end}$$
$$\quad \text{end}$$

Thus, the notion of a program p being correct with regard to a specification *spec* can be defined as:

$$\forall b \in M_{FCL\_spec}[spec](p) \cdot b$$

In the above definition, the evaluation of each declaration generates a variable associated with a value, and the evaluation of each formula generates a boolean value.

The semantic function for declarations is as follows:

$$M_{Declaration} : Declaration \times State \rightarrow (Variable \rightarrow Value)$$
$$M_{Declaration}[dcl : Declaration, s : State] \equiv$$
$$\quad \{\langle dcl.v, M_{Expr}[dcl.expr, s]\rangle\}$$

The semantic function for formulas is as follows:

$$M_{Formula} : Formula \times State \rightarrow \mathsf{Bool}$$
$$M_{Formula}[f : Formula, s : State] \equiv$$
$$\quad \text{case } f \text{ of}$$
$$\quad\quad Negation : !M_{Formula}[f.f, s]$$
$$\quad\quad Conjunction : M_{Formula}[f.f_1, s]\ \wedge\ M_{Formula}[f.f_2, s]$$
$$\quad\quad Disjunction : M_{Formula}[f.f_1, s]\ \vee\ M_{Formula}[f.f_2, s]$$
$$\quad\quad Existential : M_{Existential}[f, s]$$
$$\quad\quad Universal : M_{Universal}[f, s]$$
$$\quad \text{end}$$

An existential formula is *true* if and only if there is at least one assignment to its bound variables satisfying the formula:

$$M_{Existential} : Existential \times State \rightarrow \mathsf{Bool}$$
$$M_{Existential}[ext : Existential, s : State] \equiv$$
$$\quad \text{given } bindings = M_{BVar\_Declaration\_list}[ext.bVarl, s]$$
$$\quad \text{then } \bigvee\{M_{Formula}[ext.f, s \uplus b] \mid b \in bindings]\}$$
$$\quad \text{end}$$

A universal formula is *true* if and only if all of the assignments to its bound variables satisfy the formula:

$$
\begin{array}{l}
M_{Universal} : Universal \times State \rightarrow \mathsf{Bool} \\
M_{Universal}[uni : Universal, s : State] \equiv \\
\quad \text{given } bindings = M_{BVar\_Declaration\_list}[uni.bVarl, s] \\
\quad \text{then } \bigwedge\{M_{Formula}[uni.f, s \uplus b] \mid b \in bindings]\} \\
\quad \text{end}
\end{array}
$$

A list of bound variable definitions generates a set of "bindings". Each "binding" is a set of name-value pairs:

$$
\begin{array}{l}
M_{BVar\_Declaration\_list} : BVar\_Declaration\_list \times State \rightarrow \mathbf{F}\ (Variable \rightarrow Value) \\
M_{BVar\_Declaration\_list}[dl : BVar\_Declaration\_list, s : State] \equiv \\
\quad \text{if } dl.length = 0 \text{ then } \emptyset \\
\quad \text{else given } pairs = \{dl(0).v\} \times M_{Expr}[dl(0).expr, s]; \\
\qquad\qquad restBindings = M_{BVar\_Declaration\_list}[dl.tail, s \uplus p] \\
\qquad \text{then } \{\{p\} \uplus r \mid p \in pairs \wedge r \in restBindings\} \\
\qquad \text{end} \\
\quad \text{end}
\end{array}
$$

Each expression yields a value. The value depends on the type of the expression:

$$
\begin{array}{l}
M_{Expr} : Expr \times State \rightarrow Value \\
M_{Expr}[exp : Expr, s : State] \equiv \\
\quad \text{case } exp \text{ of} \\
\qquad Constant : M_{Constant}[exp, s] \\
\qquad Variable : M_{Variable}[exp, s] \\
\qquad Formula : M_{Formula}[exp, s] \\
\qquad ExprWithVars : M_{ExprWithVars}[exp, s] \\
\qquad Operation : M_{Operation}[exp, s] \\
\quad \text{end}
\end{array}
$$

Each constant has a real value as its denotation; we omit their semantic functions. The semantics of $Variable$ is defined as follows:

$$
\begin{array}{l}
M_{Variable} : Variable \times State \rightarrow Value \\
M_{Variable}[v : Variable, s : State] \equiv s(v)
\end{array}
$$

The semantics of $ExprWithVars$ is defined as follows:

$$
\begin{array}{l}
M_{ExprWithVars} : ExprWithVars \times State \rightarrow Value \\
M_{ExprWithVars}[exp : ExprWithVars, s : State] \equiv \\
\quad M_{Expr}[exp.expr, s \uplus M_{Declaration\_list}[exp.dl, s]\ ]
\end{array}
$$

The semantics of the declaration list is defined as follows:

$$
\begin{array}{l}
M_{Declaration\_list} : Declaration\_list \times State \rightarrow \mathbf{F}(Variable \rightarrow Value) \\
M_{Declaration\_list}[dl : Declaration\_list, s : State] \equiv \\
\quad \text{if } dl.length = 0 \text{ then } \emptyset \\
\quad \text{else given } pair = \{\langle dl(0).v, M_{Expr}[dl(0).expr, s]\} \\
\qquad \text{then } pair \uplus M_{Declaration\_list}[dl.tail, s \uplus pair] \\
\qquad \text{end} \\
\quad \text{end}
\end{array}
$$

In the abstract syntax, set and sequence operations, relational operations, and functions on the object model are under the category of operations. Since the semantics of most of the set, sequence, and relational operations are standard, we will present the semantic

function for only set comprehension. FCL_fct represents all the FCL functions on source code model; these functions are described in Section 3.2.2.

The semantic function of $Set\_comprehension$ is defined as follows:

$$
\begin{array}{l}
M_{Set\_comprehension} : Set\_comprehension \times State \rightarrow Set\_type \\
M_{Set\_comprehension}[sc : Set\_comprehension, s : State] \equiv \\
\quad \text{given } bindings = M_{BVar\_Declaration\_list}[sc.bVarl, s] \\
\quad \text{then } \{ele \mid M_{Formula}[sc.p, s \uplus b] = true, ele = M_{Expr}[sc.ele, s \uplus b], ele \neq *, b \in bindings\} \\
\quad \text{end}
\end{array}
$$

Note that if the result of the element function is "undefined," the element will not be included in the set. Thus set comprehensions always return "valid" sets.