Using SCL to Specify and Check Design Intent in Source Code

Daqing Hou, *Member, IEEE,* and H. James Hoover

***Abstract*— Software developers often fail to respect the intentions of designers due to missing or ignored documentation of design intent. SCL (Structural Constraint Language) addresses this problem by enabling designers to formalize and confirm compliance with design intent. The designer expresses their intent as constraints on the program model using the SCL language. The SCL conformance checking tool examines developer code to confirm that the code honors these constraints. This paper presents the design of the SCL language and its checker, a set of practical examples of applying SCL, and our experience with using it both in an industrial setting and on open-source software.**

***Index Terms*— design intent, structural constraints, program analysis, object-oriented software, SCL, FCL**

I. Introduction

A. Design Intent

The road to good software is paved with intentions. Intent is always on the mind of developers: Does this code really do what I want it to do? Am I using this interface in the way the author envisaged? Do these requirements capture what the client wants? Have we supported all the customer's use cases? Can we satisfy management's desire for adaptability to serve new business opportunities? Have we satisfied government privacy regulations? Software is ultimately evaluated in terms of how well it implements the intentions, often conflicting, of the various parties involved in its construction.

The job of developers is to make the many mutually-dependent design decisions that are required to express these intentions in software. Of all the parties involved in building software, developers are the ones that must precisely deal with intent as they convert it into code. Our goal is to develop tools to allow developers to capture intent and assist them in checking that their code complies with that intent.

Grappling with intent is not unique to software development. For example, the CAD-CAM (Computer Aided Design and Manufacturing) community [1], [2], [3] makes a loose distinction between the intended purpose of an artifact (its user intent) and the decisions made by the designer in producing the artifact (the design intent). This distinction in intent is by nature fuzzy, since design decisions are often intimately related to satisfying user intent — a device is designed this way because the user wants to use it in this particular environment. But in general, the user's ultimate intended purpose of the device is not mentioned as part of the design intent. This suggests the following operational distinction: *User intent* deals with using an artifact to solve problems in the user's world. *Design intent* deals with using a particular technology to construct an artifact for a user.

User intent is a notion from the requirements arena. Clients commission an application because they have some intended use for it. Requirements gathering is all about translating client ideas of intended use into more or less precise requirements. Expressing user intent requires an associated domain model that establishes the context of the problem to be solved. Such domain models are difficult to make precise, and the notations used to express them vary across domains. General methods to express user intent, such as Constantine and Lockwood's [4] essential use cases, are imprecise. In practice, it is the process of actually building and using an application that ultimately clarifies user intent — sophisticated users understand that software development is in essence iterative requirements gathering. Because of its broad imprecise scope, developers do not yet have any systematic formal way to deal with user intent, although there is effort towards this [5]. We do not address the general problem of representing user intent here.

Design intent is somewhat more constrained than user intent, and so perhaps more amenable to formalization. Although dependent on the context established by the application's domain model, design intent tends to focus on issues within the design, without explicitly referencing user intent. That is, design intent is more associated with the implementation technology rather than the user's problem. We can reasonably expect that for any given implementation technology the language for capturing design intent should be common across the range of applications built with that technology.

B. Design Intent for Software

Although there is no generally accepted definition, for our purposes we use the term *design intent* to refer to all the issues that developers deal with in the production of an application. The intentions of clients, management, and other parties are not the concern of this paper, except where they are influences on design intent. We also ignore the issues that developers face when they are acting in other roles, such as when they need to manage user intent.

Design intent can be loosely classified into two kinds. *Functional design intent* is concerned with the behaviour of the application. *Non-functional design intent* deals with the broader issues such as coding style, naming conventions, language idioms, design patterns, and designs specific to an

Manuscript received November 1, 2005; revised March 1, 2006. This work was supported by NSERC and the Alberta Ingenuity Fund.

Daqing Hou (daqing@cs.ualberta.ca, corresponding author) is with Avra Software Lab Inc., 9723-88 Ave, Edmonton, Alberta, Canada T6E 2R1. H. James Hoover (hoover@cs.ualberta.ca) is with the Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2E8.

application. Non-functional design intent is also present in issues such as performance and maintainability. In general, each bit of a design has both kinds of intent behind it. For example, the observer pattern is associated with specific intended behaviour involving the way changes are propagated to observers. It also has non-functional aspects such as the way in which new subject and observer classes are to be derived from their respective base classes.

Software quality depends on how effectively the developers can make design decisions in the first place, communicate them, and preserve them during development and evolution. Thus documenting design intent, implementing the intent in code, and checking that the code matches the intent are all crucial activities in building software. In practice, some design intent is represented in source code as comments and annotations, much is buried in separate documentation, and many of the key decisions exist only in the minds of the developers.

Each bit of a design a developer makes is, more or less, intended to address client requirements, deliver functionality, or support developer goals relative to the construction and evolution of the application. For every piece of design we can ask: what problem it solves (its intent), how it solves it (the design), and why it was chosen (its rationale). Quantifying the rationale behind a design decision is difficult, if not impossible. But what about the more focused problem of checking if the design captures the intent? Since design is ultimately expressed as code, can we determine if code complies with design intent?

For example, as a designer we might want to support platform flexibility (our intent). This leads us to a design decision to distribute key services over a loosely collected system of modules. The rationale for this is that our experience has shown that this style of architecture copes well with platform change. Unable to codify a notion of platform flexibility, there is no way that we can check that our design preserves our intent. But if we can codify the notion of loose coupling between modules, we may then recast our intent as "maintain loose coupling," and check that our design preserves this more focused intent.

Our goal in this paper is to introduce a language that permits developers to express aspects of design intent precisely enough to mechanically check that code conforms to the intent. Since it is neither possible nor necessary to automate all intent checking, our goal is rather modest but practical: We are interested in tools that can enforce, at least partially, the correct use of a design. Our thesis is that much of the design intent that influences quality can indeed be handled in this way.

C. Checking Conformance with Design Intent

As they build software, developers need to confirm the conformance of their code with design intent. They use four main complementary techniques to gain confidence in their products: formal verification, testing, manual code inspection, and static analysis. The order, frequency, and depth at which these are done in practice varies widely over developers and projects. Most development practices focus on conformance checking of functional design intent, using various techniques.

In formal verification, the intent is captured in a formal specification, and then a proof that the program meets the specification is attempted. The specification is usually only partially successful at encoding the full intent, and the proof can be of varying rigor. But in many situations, formal verification can give very strong evidence that the program indeed meets the specification. In reality, specifications are difficult and costly to write, and the proof process is tedious and expensive. It is rare for large pieces of software to be fully verified. But it is not unusual for mission-critical components to undergo extensive formal verification.

Testing is easier to perform than verification. Since it deals with the program directly by running it, there is no need of the extra abstractions required by more formal verification methods. Testing is weaker than verification in the sense that it can be too costly (or impossible) to obtain complete coverage of all possible execution paths. Specialized test environments may be required for systems with external events or complex user interfaces. More importantly, testing can only check execution time behavioural properties of programs. It cannot test non-functional properties such as compliance to coding standards. In fact, it is impossible to test most of the constraints presented in this paper.

Neither formal verification nor testing can grapple with non-functional intent. This is done, if at all, through manual code inspection. Manual code inspection [6], [7] is flexible in that it applies to both functional and non-functional intent, and humans can compensate for unusual conditions and practices. But manual inspection is unreliable, costly, difficult to repeat, and hard to scale to large code bases. This is so even when supported by simple tools such as Grep, or more complex program comprehension tools like Rigi [8].

The fourth technique for confirming intent is static analysis. Static analysis can check source code that cannot be executed conveniently, and examine corner cases that are often missed by testing. It can be more effective than testing in detecting certain errors such as memory leaks. Most importantly, because it is dealing with the code as written by the developer, static analysis can be used to detect violations of some kinds of non-functional intent.

Static analysis examines the source code and produces a model of the software. The model typically uses a graph structure to record relationships between entities in the code, such as program structure (for example class hierarchies) and program dynamics (for example flow graphs). The model may also hold additional artifacts, such as lexical information, inferred invariants, and previously computed query results.

Functional and non-functional design intent can then be expressed using constraints on the model. The typical compiler illustrates this. The parse tree of a program is the model. A language specification can be thought of as a collection of constraints that the compiler must enforce. For example, the compiler can confirm that each method is being called as intended, at least with respect to argument types, by checking that the model satisfies the constraint that the call must be type-compatible with the signature of the method definition.

Many rules that capture design intent are domain-specific and cannot be fully anticipated, so tools need to be both open and expressive, and extension must be relatively simple. Also, tools themselves may need to be optimized for particular application styles, so we want the constraint specifications to be as independent as possible from any particular tool. Thus a constraint specification language is needed to provide developers with the ability to express intent-capturing rules as they see fit. This motivates our introduction of SCL, the Structural Constraint Language.

This paper presents the design and implementation of both the SCL language and its checker, a set of usage examples, and our current experience with SCL. Section II discusses some of the motivation for the design tradeoffs in SCL. Section III gives an overview of SCL. Section IV defines the SCL language. Section V presents a selected subset of SCL examples. Section VI summarizes our experience with SCL. Section VII presents related work. Finally, Section VIII concludes the paper.

II. Formalizing Design Intent via SCL

A. Structural Constraint Language

SCL is a specification language and system aimed at specifying and checking structural design constraints (or rules) on a code model produced by static analysis. As currently implemented, SCL primarily constrains code structure rather than program behaviour. By design, SCL is a less expressive language than one would want for general reasoning about programs. For example, lack of support for induction in SCL means that we cannot reason about loops. On the other hand, SCL can express constraints like "call to function Y cannot appear in function X," something impossible with traditional specification languages like Z and VDM. In designing SCL we favored automation over expressiveness. By limiting the expressiveness of SCL we ensure that SCL specifications can be checked automatically. SCL can often report errors with a low rate of false positives, especially when tuned with problem-specific heuristics. However, the goal of SCL is not to prove the absence of errors but to quickly find as many errors as possible.

Many static analysis tools detect only a fixed set of errors and have no specification languages and extensibility. Tools such as Lint and PREfix [9], encode their rules as procedures that operate on the static model. Although certainly formal, this makes the tools closed in the sense that a user cannot extend a tool with new rules without modifying the tool internals. Some tools, such as SABER [10], mitigate this somewhat by having rule templates that a user can parameterize. SCL on the other hand has no built-in set of rules, and so the SCL language contains many primitives intended to capture a variety of structural features that appear in constraints.

Behavioural checking is expensive, so SCL concentrates on structure, although it does include data flow and control flow analysis capability. But compared to deep analysis tools (like [9], [11], [12], [13], and [14]), SCL does a better job in modeling structures than tracking behaviour. Since structure and behaviour are interwoven, aspects of behaviour are reflected by structures. Thus reasoning about structures can give insights about behaviour. In fact structural anomalies are often strong indicators of actual behavioural errors.

We intend SCL to be a natural part of the design and programming process. In a design phase, a designer uses SCL to specify key design constraints. Such constraints can then be used to check implementation. This is especially useful when the constraints can be mapped to code structure. One can also retrofit SCL constraints to existing systems, such as software frameworks and extensible libraries, which typically impose numerous constraints on their use.

When we first began this effort we focused on framework-based development. Object-oriented frameworks like Java Swing and Microsoft Foundation Classes (MFC) consist of hundreds of classes and extension points, all intended by the framework designers to be used in particular ways. The difficulty in understanding how to use the frameworks and detecting errors in using them are major sources of errors in framework-based development. Many of these issues can be described in terms of structural properties of the source code. Thus the original name for SCL was the Framework Constraint Language (FCL) [15], [16], [17].

B. SCL Features

SCL has the following features:

- SCL is a form of active documentation. It is important to transfer design knowledge across both spatial (among individuals) and temporal (between different time instants for the same person) boundaries. For medium (100 kloc's) and large (1000 kloc's) systems, it is difficult for a developer to manually review design consistency. Re-inspection is required to catch regression errors as the software changes. Capturing intent in SCL enables continuous automatic review.
- SCL specifications can be developed incrementally. This encourages practitioners to apply SCL, since they do not incur the up-front cost of writing full specifications. This is particularly applicable to agile development processes.
- SCL works directly on program source. Current industrial practice favors code over documentation. Working on program source allows for the incremental introduction and adoption of SCL by developers. Developers can quickly see benefits from a small initial investment.
- SCL is reasonably language-independent. SCL was designed so that it was not, in general, tied to any particular procedural or Object-Oriented language. For example, general rules of good O-O style will work on both C++ and Java code. Idiosyncratic language features can be added as extensions to the term language of SCL.
- SCL is declarative. Unlike rules expressed procedurally in terms of tool internals, declarative rules are reasonably portable and reusable. Since the rules are expressed in a well-defined language, they are themselves amenable to static analysis.

III. System Overview

We now illustrate the process of using SCL through an example in C++. Then we introduce the architecture of SCL.

A. An Initial SCL Example

Good class design recommends maintaining the principle of substitution for derived classes. When a derived class extends the behaviour of its base class, it should do so in a way that preserves base class behaviour. That is, a derived class should act like its base class when used in a base class setting. In addition, derived classes should be as loosely coupled to their ancestors as possible.

For example, suppose base class B has a method m. If class D is derived from B, then the method m of D needs to preserve the behaviour associated with B as well as handle any additional behaviour associated with the extension. Furthermore, the overriding implementation of m in D should be loosely coupled to B::m in order to permit changes in the base class implementation (and possibly its design) to occur without having to alter the derived class D. In other words, details of B::m should not be appearing in D::m.

A sign that this loose coupling is occurring is that the implementation of D::m calls its base class version B::m. Another way of thinking of this is that the new method is an extension plus a reduction (in the complexity theory sense) to the existing method. Thus we have the following constraint: An overriding method in a subclass should call its superclass version; to aid comprehension this call should be explicit, not hidden in another method.

How is this expressed in SCL? Take two files B.h and B.cpp that define a class B. Class B defines virtual member function m.

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

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

The constraint for B is: If a subclass of B overrides m, then the override must explicitly call B::m. This is specified in SCL as follows:

```
for D: subclasses(class("B")) holds
[def m_B as method("m", class("B"));
 def m_D as method("m", D)]
  exists e: exprs(m_D) holds
  method(e) = m_B
```

In this specification, subclasses, class, method, and exprs are functions on the syntactic structure of program source (see Section IV for their semantics). The specification can be read as saying "for all subclasses D of class B, there must exist an expression e in the definition of m_D (D::m) that refers to the method m_B (B::m)."

Now suppose a programmer accidentally breaks this constraint by not calling the superclass version from subclass D.

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

// D.cpp
#include "B.h"
void D::m(){
  ... // D::m does not call B::m
}
```

To check this constraint the C++ source is converted into a graph of relationships between program elements. The SCL evaluator then evaluates the SCL expression using the relationship graph to determine sets of quantified entities (for example, subclasses of B) and values of terms (for example a particular expression is a method call). Diagnostic messages are emitted when anomalies are identified. Fig. 1 illustrates a graph representation of the example code and how the SCL specification is evaluated against the graph. The SCL specification is essentially a query against the program model.

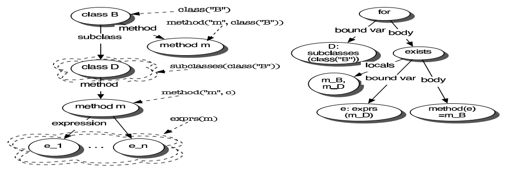

Fig. 1. Evaluating SCL constraints. Left: program facts as a graph; dotted areas indicating sets of entities that map to quantifiers in the constraint expression. Right: parse tree for the SCL constraint.

When the SCL specification is not satisfied it is important to provide sufficient information to diagnose the problem. We have implemented two forms of diagnostic messages: command-line output for the C++ SCL, and a graphical user interface for the Java SCL. Java diagnostics combines syntax highlights, decorations, and tooltips to provide feedback.

B. SCL Architecture

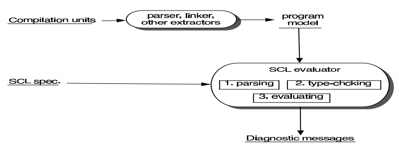

Fig. 2. SCL architecture

We have implemented SCL twice. The C++ version was a limited SCL proof of concept. The Java version allowed us to test SCL on production code bases. The Eclipse environment and the SCL evaluator now give us an initial framework for SCL support for additional languages. At the core of the architecture in Fig. 2 is the SCL evaluator, which requires two inputs, an SCL specification and a program model (or program fact database).

A constraint may require information transcending program units such as classes and packages, possibly even a whole-program model. But in general SCL needs to model only the parts of a program that are relevant to the constraint. The program model contains the declarative structure of the program gathered from both the syntactic and semantic passes of compilers, and control and data dependences. The model is mapped to a representation in which source code entities are strongly-typed objects and relationships are implemented by methods on these objects. SCL specifications are evaluated on this representation.

The actual evaluation of an SCL constraint is conceptually straight-forward. The SCL evaluator recursively descends through the structure of the SCL formula. At each quantifier it constructs the finite set that forms the domain of the quantifier. Then it evaluates the sub-formula against each element of the domain. Finally primitive functions are evaluated directly against facts in the model. Obviously there are opportunities to speed up the checking using standard techniques like caching, indexing, and static analysis of SCL itself. Performance of the current implementation is discussed in Section VI-B.

SCL constraints can be evaluated in two modes. By default the evaluation is lazy, with the evaluation of the constraint terminating in false on the first counter-example. Such a constraint is called an *assertion mode* constraint. For diagnostic reasons, one may want all the cases where the constraint is violated so as to mark up the program source. Such a constraint is called a *set mode* constraint.

IV. The SCL Language

SCL is a first-order logic with a term language for talking about programs. The syntactic structure of an object-oriented program forms a graph. Nodes represent syntactic elements such as name spaces, classes, functions, variables, and expressions. Edges represent the relationships between these elements such as a function call expression and the function that it is statically bound to, or a variable and its type. The term language of SCL consists of a set of total functions reflecting the entity-relationships in the graph representation of object-oriented programs. We now introduce SCL using the abstract syntax of Fig. 3.

At the topmost level, an SCL specification consists of a sequence of interleaved declarations and formulas. Declarations 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 *SCL constraint*.

Each declaration binds a variable to an expression that fixes the value of the variable for the extent of its scope. Logical formulas are treated as a special kind of expression that yields values of the boolean type. Thus SCL allows one to define boolean variables with formulas as their value expressions.

SCL allows one to introduce local variables for expressions through a syntactic structure called a *block*. Blocks are a simple grouping mechanism that create scopes. Each expression is

```
SCL_spec = Stmt*
Stmt = Decl | Form
Decl = [‘def’] Var ‘as’ Expr

Form = ‘!’ Form| Form ‘&’ Form| Form ‘|’ Form|
Form‘=>’Form |Form‘<=>’Form| Ex | Univ | Expr
Ex = ‘exist’ BVar_Decl+ ‘hold’ Form
Univ = ‘for’ BVar_Decl+ ‘hold’ Form
Expr = Var | Const | Op | Expr_With_Vars
BVar_Decl = Var‘:’ Expr
Expr_With_Vars = ‘[’Decl+‘]’ Expr

Op = Set_op | Seq_op | Rel | SCL_fct
Set_op = Set_compr | Set_enum | member | cardinality|...
Set_compr = ‘{’ Expr ‘|’ BVar_Decl+ Form ‘}’
Set_enum = ‘{’ Expr* ‘}’
Seq_op = ith (seq, index) | indexOf (ele, seq) |...
Rel = > | >= | < | <= | =
SCL_fct = Str ‘(’ Expr* ‘)’
Const = ‘true’ | ‘false’ | Quoted-string | Integer
|‘packages’ ...
```

Fig. 3. Abstract syntax of SCL

allowed to have at most one associated block. As usual, one variable overrides another 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 useful for structuring formulas by avoiding long or repeated expressions. Assigning a variable name to an expression can also help reveal the intent of the expression.

Logical formulas have conventional semantics with the usual negation, conjunction, disjunction, implication, equivalence, and universal and existential quantifications. We adopt the highly readable style used in Mizar [18] that uses phrases for expressing universal and existential quantifications.

Primitives include the boolean constants *true* and *false*, relational operations, and such predefined predicates as the subset relation and the set membership relation. Syntactically, these predicates are represented as function applications in the form of $f(e_1, \ldots, e_n)$. Both universal and existential quantifications are allowed to define more than one bound variable at once, binding them to the elements of the set-valued expressions.

Expressions include variable references, literal constants, function applications, and sets. SCL defines some literals specific to the source code data model, such as *global*, which represents the global name space of a given program (see Table VII for others).

Function applications of the form $f(e_1, \ldots, e_n)$ are the most common expressions. SCL 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 IV-B.

One can generate sets in three ways: through function applications that return sets as results, set enumeration, and through set comprehension. Set comprehension looks like

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

, where $e_1, \ldots, e_n$ are bound variables over 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.

SCL has no need for explicit sequence construction. Sequences are obtained from the return values of function applications. For instance, 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. The range of a sequence can be used in a context where a set is expected.

A definitional capability exists for providing abstractions in SCL specifications (see Section V for examples).

A. SCL's Type System

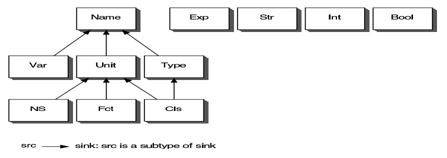

Fig. 4. Basic types and the subtype relation

SCL specifications are strongly typed to ensure that every SCL specification has a well-defined truth value. The types of SCL (not to be confused with the types of the target language) include basic types (Fig. 4) and compound types. Basic types can be further divided into facility types and domain types. Facility types, including Str for string values, Int for integers, and Bool for boolean values, help form constraints. Domain types are for program entities, including Exp for expressions, Name for named entities, Var for variables, Unit for program units that organize source code, Type for types, NS for name spaces, Fct for functions, and Cls for classes. Not shown in Fig. 4 is the type Undef for "undefined" values, considered as the subtype of all other types. Compound types are for sets or sequences of entities, such as a set of classes, or a sequence of parameters.

Fig. 4 also defines the *subtype* relation between 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. A similar definition applies to sequence types.

B. Functions on Source Code Structure

SCL views a program as a structure and asserts properties about it. Such assertions are formulated based on generic constructs from first-order logic and a rich set of total functions, which are partially presented in Tables I to VII. Some functions, such as those for templates, are not included to save space. In the tables, the notation **Set T** represents a finite set of **T**, and **Seq T** represents a finite sequence of **T**.

In the following, we briefly introduce each table (Section V contains examples of their use). First, one needs a means to refer to a program entity in SCL, which is supported by constructors shown in Table I. At the global level, a program is organized around the notion of program units: packages (name spaces for C++), classes, and functions. Table II contains functions for examining the containment relation formed by program units. Another important aspect of a program is its type information. Table III contains operations for types. Operations on expressions are listed in Table IV.

Three functions of Table IV, cd, conds, and dep, warrant extra explanation. cd and conds [19] calculate the control dependence relation among expressions. An expression w is said to be control-dependent on u if and only if there exist two execution paths from u to the end of the current function, with w on only one of them. Thus u is a "decision point" that influences the execution of w. dep captures data dependence. Intuitively, one expression is said to be data-dependent on another if the value of the latter can "influence" that of the former (see [20] for a formal definition).

C. SCL's Treatment of "undefined"

In SCL there are several ways that can lead to an "undefined" value. For example, casting an expression that is not a literal type to a literal type can generate a value of "undefined" (by literalType). Asking for a receiver expression from an arithmetic expression will also result in "undefined". (See Table IV.)

SCL designates a default value for functions that may take "undefined" as arguments, and thus make them total. For any function application that takes an "undefined" as an argument, if the return type of the function is a basic type other than boolean, then the result will be a value of "undefined". If the return type is boolean, then the function application will return *false*. If the return type is a compound type, that is, sequences or sets, then the function application will return an empty sequence or an empty set respectively.

In our experience it is convenient to map "undefined" to the default values above. For example, normally an undefined logical formula indicates the failure of a constraint. If needed, we have a predicate for testing if a value is undefined.

V. SCL Patterns of Use

The syntax and expressive power of SCL has evolved as we have applied it to a number of practical examples. These include enforcing language semantics, checking design patterns, a case study with MFC (Microsoft Foundation Classes) [21] (with a focus on the dialog architecture), analyzing a 500-kloc framework-based system, and using SCL on the SCL checker itself. We have yet to apply SCL to SCL specifications.

We now present eleven real-life examples to illustrate the use of SCL, three of which are about Java semantics and eight about MFC dialogs. These examples capture best-practices distilled by experienced industrial developers. The Java constraints are mostly derived from [22] with some improvements based on our SCL experience. The MFC examples are based on over a decade of programming and consulting experience

TABLE I
SCL CONSTRUCTORS

Operation	Description
class: **Unit×Str→Cls**	Constructors are used to specify a program entity. The first arguments specify the program unit (packages, classes, and functions) where the entity is defined, followed by details such as names of the entity. For example, the parameter-less method m of class C can be specified as method(class(global,"C"),"m"). Note that global can be omitted.
var: **Unit×Str→Var**	
function (method): **Unit×Str×Type*→ Fct**	

TABLE II
SCL SCOPE OPERATIONS

Operation	Description
classes: **Unit → Set Cls**	Returns the set of classes defined within a Unit.
exprs: **Unit → Set Exp**	Returns the set of expressions defined within a Unit.
vars: **Unit → Set Var**	Returns the set of variables defined within a Unit.
functions (methods): **Cls → Set Fct**	Returns the set of functions defined within a Cls.
params: **Fct → Seq Var**	Returns the sequence of parameters of Fct.

TABLE III
SCL TYPE OPERATIONS

Operation	Description
subclasses: **Cls → Set Cls**	Returns the set of subclasses of the argument class.
descendants: **Cls → Set Cls**	Returns the set of descendant classes of the argument class.
type: **Exp → Type**	Returns the static type of a given expression.
type: **Var → Type**	Returns the type of a given variable.
returnType: **Fct → Type**	Returns the return type of a given function.
class: **Type → Cls**	Casts a type into a class. returns "undefined" if fails.
isArray: **Type → Bool**	Returns true if Type is an array type.

TABLE IV
SCL EXPRESSION OPERATIONS

Operation	Description
receiver (primary): **Exp → Exp**	Returns the receiver of the argument. Returns "undefined" if no receiver.
args: **Exp → Seq Exp**	Returns the sequence of arguments of an expression, including the receiver.
cd: **Exp → Set Exp**	Returns the set of expressions that transitively control-depend on this expression.
conds: **Exp → Set Exp**	Returns the set of expressions that this expression control-depends on transitively.
dep: **Exp×Exp → Bool**	Returns true if the value of the first expression depends on that of the second.
uses: **Exp → Set Exp**	uses(e) returns the set of expressions that contain either expression e or a variable aliased to e.
function (method): **Exp → Fct**	Returns the function that a given expression is statically bound to.
var: **Exp → Var**	Returns the variable that a given expression is statically bound to. Returns "undefined" if not a variable.
refd: **Exp → Name**	If the argument is a reference expression, returns the referred entity. Otherwise, "undefined".
literalType: **Exp → Type**	If the argument is a reference to a type name, returns the type. Otherwise, "undefined".
int: **Exp → Int**	Returns the integer value if the expression is an integer constant, otherwise, "undefined".
isLiteralNull: **Exp → Bool**	Returns true if the expression is the null pointer (0 for C++).

TABLE V
SCL PROPERTY PREDICATES

Operation	Description
isPrivate, isProtected, isPublic: **Var|Fct|Cls → Bool**	Tests levels of access control. In C++ applicable only to class members.
isStatic: **Var|Fct|Cls → Bool**	Tests staticness. In C++ applicable only to variables and functions.
isConst (isFinal): **Var|Fct → Bool**	Tests constness. In Java applicable only to variables.
isVirtual: **Fct → Bool**	Tests whether a function is virtual.

TABLE VI
SCL MISC OPERATIONS

Operation	Description
isDefined: **Any** → **Bool**	If the argument is "undefined", returns *false*. Otherwise, returns *true*.
name: **Name** → **Str**	Returns the name of a named entity.
regex: **Str** × **Str** → **Bool**	Returns *true* if the second string matches the first regular expression.
concat: **Str** × **Str** → **Str**	Returns the concatenation of two strings.
print: **Any** → **Bool**	Prints the textual representation of the argument, and returns *true*.
error, warning, info: **P Name** × **Str** → **Bool**	Marks the set elements as errors, warnings, and infos, respectively, in the IDE.

TABLE VII
PRE-DEFINED SCL CONSTANTS

Constant	Description
cast	casting operator.
new	new operator.
assignment	the = operator.
arrayaccess, fieldaccess	[] and .
equality, inequality	equality operators == and !=.
lt, le, gt, ge	comparison operators.
int, uint, boolean, ...:	primitive types.
instanceof	Java's instanceof operator.
global	global namespace.
packages	set of packages for a Java project.

of an MFC expert [23]. They are common problems that developers encounter when using the Microsoft Foundation Classes for building graphical user interfaces. These examples collectively demonstrate that design constraints abound in practice and that SCL is effective in enforcing them.

The rest of this section is organized as follows. The three Java constraints are presented first in Sections V-A, V-B, and V-C. Next are examples from MFC dialogs: Sections V-D, V-E, and V-F are programming obligations implied by framework designs that programmers must remember to fulfill. Section V-G presents two examples about APIs whose use is prohibited in certain contexts due to framework evolution and unclean design. Finally, we show how SCL is used to enforce programming disciplines and naming conventions (Sections V-H, V-I, and V-J).

A. JAVA: Correct Signatures for equals

We begin with a simple example concerning the equals predicate for objects derived from the Java Object class. The Java designers intended (see V-B) that there should only be one public equals predicate defined for a class. Public implementations of equals in classes derived from Object must take Object as the parameter type in order to override the default base class implementation. That is, they should have signature public boolean equals(Object o).

Here is an SCL specification of this constraint, informally read as: All public methods (in classes derived from java.lang.Object) that have the name "equals" and take one argument, must return a boolean and take an Object as argument.

```
def Object as class("java.lang.Object")

for p: packages, c: classes(p), m: methods(c)
(
(name(m)="equals" & isPublic(m) &
sizeof(params(m))=1)
=>
(returnType(m) = boolean &
type(ith(params(m),0))=Object)
 )
```

A few projects (Daikon, Apache Ant, and Apache Avalon) appear to follow a pattern of defining two public equals for a class, one with parameter type Object and the other with the class as its parameter type. Although this is not wrong, there is no compelling reason to have the extra equals [22]. Such redundancy does not introduce any new functionality but it does introduce potential confusion when a programmer has to decide which to invoke. We recommend that all new projects should enforce the single-equals rule.

It is probably a bug if a legacy project with multiple public equals methods has no equals with parameter type Object. The following constraint says: If there is a public equals that does not take Object as parameter, then there must be a public equals that indeed does.

```
for p: packages, c: classes(p)
(
exists m: methods(c)
(name(m)="equals" & isPublic(m) &
sizeof(params(m))=1 & returnType(m) = boolean &
!type(ith(params(m),0))=Object)
=>
exists m: methods(c)
(name(m)="equals" & isPublic(m) &
 sizeof(params(m))=1& returnType(m) = boolean&
 type(ith(params(m),0))=Object)
 )
```

B. JAVA: Obeying the Override Contract for equals

We now do a more in-depth example that illustrates many features of SCL. A common object-oriented design technique is to have the super-class contain utility methods with well-defined contracts. These utility methods are intended to be used to implement further functionality. All implementations of the utility methods in the extending classes must obey the contracts associated with the methods.

An example of a utility method is the equals of class Object with signature public boolean equals(Object o) that implements the equality predicate for Java objects: x.equals(y) is true if object x and object y are equivalent. The contract for equals requires that it be reflexive, symmetric, and transitive. To work properly, all hash-based classes such as HashSet require the presence of an equals that satisfies this contract.

When implementing the equals predicate for a new class A, programmers must consider the following three points:

- Because the signature of equals is specified in the base class, one must test that parameter o is of a correct type. The object o has a correct type if it is an instance of class A, or an interface implemented by A. Such a test can be done using either the instanceof operator or the getClass() method defined by Object.
- Parameter o must be cast into the correct type and then compared with this. There will be a control dependency between the expression used to test the type and the expression used to cast.
- If getClass is used to test the correct type, then o must not be null. Thus the call to getClass must be guarded by a test of o being not null.

The following SCL snippet specifies the first two points:

```
for p: packages, c: classes(p), m: methods(c)
(
isEquals(m)=>
[def o as ith(params(m), 0)]
callsCasting(m,c,o)
)
```

In this snippet, isEquals and callsCasting are macros. isEquals tests if m matches equals. For brevity we skip its definition. callsCasting(m,c,o) tests if method m has an expression that casts the identifier o to class named c or an interface implemented by c:

```
def isDerived(d, s) as (d=s || in(d,
 descendants(s)))
def callsCasting(m, c, o) as
exists e: exprs(m) holds
[def tt as class(literalType(ith(args(e),0)))]
(method(e)=cast & isDerived(c, tt) &
var(ith(args(e),1))=o & guarded(e, c, o))
```

In the event that e is a cast, then tt is the target type. isDerived(c, tt) and guarded are two macros. isDerived(c, tt) tests if tt is either c or an interface implemented by c, and guarded requires that the cast expression must control-depend on a type testing expression.

```
// true if e is a correct instanceof
def isInstanceof(e, c, o) as
[def tt as class(literalType(ith(args(e),1)))]
(method(e)=instanceof & var(ith(args(e),0))=o &
isDerived(c, tt))
// true if e is o.getClass()
def isGetClass(e, o) as
(method(e)=getClass & var(primary(e))=o)

def guarded(e, c, o) as
exists control: conds(e)
(isInstanceof(control, c, o) ||
isGetClass(control, o))
```

The following SCL snippet specifies the third point in the above guidelines:

```
def isNull(e, o) as
(method(e)=equality || method(e)=inequality) &
[
def lhs as ith(args(e), 0);
def rhs as ith(args(e), 1)
]
(isLiteralNull(lhs)&var(rhs)=o ||
isLiteralNull(rhs)&var(lhs)=o)

for p: packages, c: classes(p), m: methods(c)
(
isEquals(m)=>
[def o as ith(params(m), 0)]
for e: exprs(m)
(isGetClass(e, o)
=>
ex control: conds(e) isNull(control, o))
)
```

In practice, these programming obligations are often neglected even by experienced developers. We regularly run constraints over our own SCL code base during development, and we have caught a number of incorrect equals implementations. We also found 6 problematic instances in our other case studies. These projects have been heavily tested, reviewed, and used in production, yet these errors remain. This shows that SCL can be a useful addition to a programmer's toolbox. Here are two specific examples from production open source code.

Class Property in package org.eclipse.ant.core implements equals without testing if other is null before calling other.getClass().

```
public boolean equals(Object other) {
  if (other.getClass().equals(getClass())) {
     Property elem= (Property)other;
     return name.equals(elem.getName());
  }
  return false;
}
```

Class Term in package org.apache.lucene.index implements the following equals without testing the type of o before casting it:

```
public final boolean equals(Object o) {
  if (o == null)
      return false;
  Term other = (Term)o;
  return field == other.field && text.equals(other.text);
}
```

In the industry project, SCL picked up the following implementation of equals. Interestingly enough, the original programmer knew that this implementation was incorrect, but has not corrected it yet.

```
public boolean equals(Object obj)
{
   //TODO: this implementation not technically correct
   return address.equals(((AddressAdaptor) obj).address);
}
```

C. JAVA: Array Accesses in Loops

Here is a very local application of SCL that is less concerned with abstractions and interfaces, but more focused on detecting localized signs of potential mistakes.

A common coding mistake when accessing arrays in nested loops is to misuse the index variable of one array as the subscript of a different array.

```
for (int i=0; i< a.length; ++i)
for (int j=0; j< b.length; ++j)
{
  ...a[i]... // correct
  ...a[j]...// likely to be an error
}
```

The following SCL states that if the index variable of an array is compared with its length, then any access to the same array that is control-dependent on the comparison must use the same index variable as its subscript.

```
for p: packages, c: classes(p), m: methods(c),
ce: exprs(m) holds
[def index as var(ith(args(ce), 0));
def arrLen as ith(args(ce), 1);
def array as ith(args(arrLen), 0)]
(
method(ce)=lt & isDefined(index) &
isArray(type(array)) &
method(arrLen)=fieldaccess
 =>
 for e: cd(ce) holds
 [def arrayOfe as ith(args(e), 0);
 def indexOfe as var(ith(args(e), 1))]
 (
 method(e)=arrayaccess & isDefined(indexOfe) &
 arrayOfe = array
 =>
 indexOfe = index
 )
 )
```

Note that this specification depends on the fact that length is the only pre-defined field in a Java array. If there were other possible fields we would have had to explicitly mention length in the constraint. Also there can be other ways to catch array mistakes, for example, by requiring the array be used in the loop body.

D. MFC: Enabling Tooltips for Dialog Child Controls

A dialog can be viewed as a display and input device that can contain other visual controls. In MFC programming, the default framework class for dialogs is CDialog. One can customize the behavior of a dialog by subclassing CDialog. Similarly, controls can be customized by subclassing their corresponding MFC classes. Controls may be represented as data members of a dialog's class; such members are also called "control variables". Each dialog control is associated with a constant integer called its "control ID", which uniquely identifies the control within the dialog.

Enabling tooltips for the child controls of a dialog requires two actions: to define a message handler named OnToolTipNotify in the dialog's class, which returns a tooltip text to be displayed; and to call the CWnd::EnableToolTips method within the OnInitDialog method. In practice, programmers often forget either to invoke EnableToolTips, or to define OnToolTipNotify [24]. SCL can detect such omissions.

```
DD as subclasses(class("CDialog"))
ETT as function(class("CWnd"),
 "EnableToolTips", int)

for dd: DD holds
[TTN as function(dd, "OnToolTipNotify");
Init as function(dd, "OnInitDialog")]
(
  isDefined(TTN)
  <=>
   exists e: exprs(Init) holds
     function(e)=ETT
)
```

E. MFC: Overriding CDocManager::DoPromptFileName

The next example illustrates the kinds of complex intent associated with using a framework.

CWinApp and CDocManager in Fig. 5 are two major classes of MFC's Multiple Document Interface architecture (MDI).

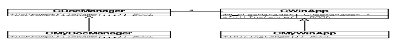

Fig. 5. Overriding DoPromptFileName

The class CWinApp is a singleton that hooks up all aspects of an MFC application; it runs the event loop logic and dispatches GUI messages that are generated due to user actions. The class CDocManager manages and coordinates documents-related classes. Each application contains a document manager to manage both the types of documents supported and the documents currently opened by the application.

MDI standardizes the look and feel of applications. In particular, each application can have two menu items: "File/Open ..." and "File/Save as ...". When either item is selected, the standard behavior is to pop up a file dialog, allowing users to choose from a list of files. The virtual method CDocManager::DoPromptFileName(..., int lFlags, ...) is responsible for popping up the dialog and displaying files satisfying a certain criterion; lFlags specifies the behavior of the file dialog using a combination of bit patterns.

In MFC one can customize the file dialog, for instance, by changing the default file filtering pattern for a specific application. Such customizations involve the following steps:

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

The bottom of Fig. 5 shows the main elements relevant to such changes: the class CMyDocManger is the subclass of CDocManager, and CMyWinApp is the subclass of CWinApp. Some of the constraints on such changes can be specified in SCL as follows:

```
CDocManager as class("CDocManager");
CMyDocManager as subclasses(CDocManager);
CWinApp as class("CWinApp");
CMyWinApp as subclasses(CWinApp);
// conditions 1 and 3
sizeof(CMyDocManager)<=1
sizeof(CMyWinApp)<=1

sizeof(CMyDocManager)=1 =>sizeof(CMyWinApp)=1
 // condition 2
 for dm: CMyDocManager holds
 [DPFN as function(dm, "DoPromptFileName") ]
 (
 isDefined(DPFN)=>
 exists e: exprs(DPFN) holds
 (
   [def lFlags as ith(params(DPFN),3);
```

```
  def argFlags as ith(args(e),3)]
  function(CDocManager, "DoPromptFileName")=
  function(e)& dep(argFlags, lFlags)
)
)
// condition 4
pDocManager as var(CWinApp, "m_pDocManager")
for myWinApp: CMyWinApp holds
[Init as function(myWinApp, "InitInstance")]
(
  isDefined(Init) &
  exists e: exprs(Init) holds
  (
  [def lhs as var(ith(args(e),0));
  def rhs as ith(args(e),1);
  def cls as literalType(ith(rhs,0))]
  function(e)=assignment & lhs=pDocManager &
  function(rhs)=new & cls=CMyDocManager
  )
)
```

The act of specifying constraints for DoPromptFileName reveals that the current design cannot support all desirable customizations. For example, DoPromptFileName creates the default file dialog as a stack object. Thus the only way to replace the default file dialog with a custom one is to copy-and-paste the body of the function and modify it. Similarly, the file filter pattern "*.*" is always used by the default file dialog. Sometimes it is desirable to not include it. Again, the only way to do this is through copy-and-pasting the function body. A re-design by introducing hook methods will solve these problems.

F. MFC: Resizing Dialogs

This example illustrates how SCL can be used to deal with a specific framework deficiency. Suppose you have a dialog with one text control and want to resize the text control as 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 has a problem. An MFC dialog may receive a WM_SIZE message even before its child controls are created, at which point a call to SetSize will cause an assertion failure. The solution is to add a data member to the dialog class to indicate that the controls are ready, and to guard all the control operations by that condition. The data member should be initialized to false in constructors and set to true at the end of the OnInitDialog method, at which point all the dialog controls must have existed. The following code snippet illustrates this technique:

```
CMyDialog
{
       BOOL initialized;
       CMyDialog(...)
       {
       ...initialized = FALSE;
       }
       OnInitDialog(...)
       {
       ...initialized = TRUE;
       }
}
```

The following SCL specification requires the presence of such a code pattern:

```
// This spec. captures assertion failures
// caused by a mismatch between Windows and MFC
// DD: user-defined subclasses of CDialog
DD as subclasses(class("CDialog"))-
{class("CCommonDialog"),class("CPropertyPage")}
cWndSet as descendants(class("CWnd"))

for ddg: DD holds
[ // OID = OnInitDialog
 OID as function(ddg, "OnInitDialog");
 OnSize as function(ddg, "OnSize", uint, int,
int);
 Ctor as {f |f:functions(ddg) name(f) =
name(ddg)};
 boolVars as {v |v: vars(ddg) type(v)=int }
 ]
 (
 isDefined(OnSize)
 =>
 exists v: boolVars holds
 (
   // assigned in both OnInitDialog and
   // Constructors
   exists e: exprs(OID) holds
   (function(e)=assignment&var(ith(e,0))=v)&
   for f: Ctor holds
   exists e: exprs(f) holds
   (function(e)=assignment & var(ith(e,0))=v)
   &// operations on controls are guarded by v
   for e: exprs(OnSize)
   (in(type(receiver(e)), cWndSet) =>
   exists condition: conds(e)
     var(condition)=v)
   )
 )
```

G. MFC: No call of CWnd::GetDlgItem, CWnd::UpdateData

Consider placing a button on a dialog. Using control variables one can define a data member for the button and send messages to the member directly, as follows:

```
CButton aBtn; // define aBtn as a data member
if(aBtn.GetCheck() == BST_CHECKED) ... // send a message
```

Historically, MFC supports a style of dialog programming without control variables, as follows:

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

where IDC_BUTTON is a control ID for the button, BST_CHECKED is a constant representing that a button is checked, and CWnd::GetDlgItem returns a pointer to CWnd, which is then downcast to CButton*. After obtaining the pointer to the MFC object in the variable aBtn, one can send button-specific messages to it.

Using GetDlgItem may cause two problems. First, it can be tedious and error-prone to change the type of the button, say to a subclass of CButton, because there can be many instances of GetDlgItem, each of which must be manually changed. Second, data may have to be maintained at two separate places: the controls and the data members of the dialog. Keeping them consistent will be a problem. Thus the best practice is to use control variables and not to call CWnd:: GetDlgItem in subclasses of CDialog. The following SCL constraint looks for violations of this best practice.

```
DD as subclasses(class("CDialog"))
```

```
GDI as function(class("CWnd"), "GetDlgItem")
for ddg: DD, f: functions(ddg), e: exprs(f)
  function(e) != GDI
```

Central to MFC's dialog design is the method CWnd::UpdateData, which transfers data from controls to the data members of the dialog class when its argument is TRUE and sends data to controls when it is FALSE. But MFC's dialog design mixes data validation and data transfer, which is inappropriate in many scenarios. A good strategy is to prohibit a dialog subclass from calling UpdateData(TRUE) using an SCL specification similar to the previous one.

H. MFC: Maintainable Combo Box Programming

A combo box control is a device for selecting one item out of a list. It displays a list of strings, each of which has a position index in the list and may be optionally associated with a data structure. Two typical wrong ways of using combo boxes are to use either the index or the string to identify the selected item.

The following code uses indices to identify colors:

```
switch(c_colorsCombo.GetCurSel())
{
  case 0: // black
         color = RGB(0, 0, 0);
         break;
  case 1: // blue
         color = RGB(0, 0, 255);
         break;
}
```

And the following code uses strings to identify colors:

```
CString s;
int index = c_colorsCombo.GetCurSel();
c_colorsCombo.GetLBText(s, index);
if (s == CString("Black"))
{
  color = RGB(0, 0, 0);
}
else if (s == CString("Blue"))
{
  color = RGB(0, 0, 255);
}
```

Both ways may cause problems for future maintenance as both indices and strings may change. Indices may change due to the addition or deletion of combo items, and thus their orders may be changed. Display strings may change due to reasons such as internationalization. The correct solution is to associate data with each item and get the data directly from the selected item.

SCL can be used to detect both symptoms. The following SCL is for the first symptom, checking that in any derived class of CDialog, there must be no equality test of the return value of CComboBox::GetCurSel with a natural number.

```
1  DD as descendants(class("CDialog"))
2  DCB as descendants(class("CComboBox"))
3  GCS as function(class("CComboBox"),
  "GetCurSel")
4
5  for dd: DD, f: functions(dd) holds
6    ! exists e: exprs(f) holds
7    (
8    function(e)=GCS&
9    in(type(receiver(e)), DCB) &
10    exists comp: uses(e) holds
11    [lhs as ith(args(comp),0);
12    rhs as ith(args(comp),1)]
13    (function(comp)=equality &
14      (int(lhs)>-1 | int(rhs)>-1))
15    )
```

Note that line 14 is due to the semantics of GetCurSel: it returns -1 if no item in a list is selected, and thus comparing its return value with -1 is legal. Also note that this constraint works regardless whether the comparison is implemented as a switch (as in the first example) or as a sequence of if's. The SCL constraint for the second symptom is similar.

I. MFC: Centralizing Transition Conditions

When implementing continuous validation, based on the current values of some other controls, one often wants to enable or disable a certain control or change its visibility by calling EnableWindow or ShowWindow. There are two typical ways of doing this. One is to disperse the logic in the event handlers responding to events such as button presses, ListBox selections, and so on. The other is to localize the control manipulation code in precisely one place in the program. The "event handler" approach can scatter code all over the place, and thus make it hard to change and maintain. Thus the localized version is preferred. The main characteristics of localized code is that all invocations of EnableWindow and ShowWindow are localized in one single method. SCL is used to ensure the presence of such a code pattern.

```
Dialogs as descendants(class("CDialog"))
CWnd as class("CWnd")
EW as function(CWnd, "EnableWindow",int)
SW as function(CWnd, "ShowWindow",int)

for d: Dialogs holds
[ EnableWindowOrShowWindow as
{f | f: functions(d), e: exprs(f)
function(e)=EW || function(e)=SW } ]
 sizeof(EnableWindowOrShowWindow)<2
```

J. General: Enforcing Naming Conventions

Large projects often establish project-specific naming conventions and coding styles. SCL contains a number of operators, such as regex, for expressing regular expression patterns on character strings, and thus can be used to enforce naming conventions. For example, in MFC programming, one convention is to require that the names of control variables be prefixed with "c_" instead of "m_" which is used only for value variables. Such constraints can be easily expressed as regular expression patterns in SCL and are inexpensive enough to be checked frequently during coding.

VI. Experience with SCL

The design of SCL has been an iterative process, driven primarily by practical examples. In addition, we adopted three development strategies. First, from the beginning we used Java SCL itself as a test case. Second, to understand the potential issues in applying SCL to industry, SCL constraints were developed and tested for the in-house frameworks of an industry partner. Third, we also used some open-source projects as test cases. Table VIII shows the Java projects that were used to test Java SCL.

Since we know our own design intent about SCL, using SCL on itself gave us a quick assessment of the correctness

TABLE VIII
JAVA TEST SUITE FOR SCL

Project	Lines of code	Description
Commercial Code Base	545 302	11 DB-centric, GUI-driven projects
SCL	23 931	SCL checker
org.apache.lucene	51 512	A library for full-text search engine
Jakarta BSF (Bean Scripting Framework)	15 712	Scripting language support for Java applications
org.apache.ant	174 252	Apache Ant
org.eclipse.ant.ui	42 648	Eclipse UI for Apache Ant
org.eclipse.ant.core	8 315	Utilities for running Apache Ant in Eclipse
Commons logging	7 573	Logging API for Apache Commons components
Avalon Framework	14 002	A component container
Avalon logkit	10 697	A logging kit for Apache Avalon Framework
Daikon 3.1.7	242 514	An invariant detector
Total	1 136 458	

of the SCL implementation. A number of rules, such as the correctness of equals, were developed, tested, and run regularly on the SCL code base. We also identified several SCL-specific rules from its design. For example, the design of SCL uses inheritance extensively, in which superclasses often impose constraints on subclasses, such as calling or overriding a method. Every so often we would forget such constraints and they would be caught by SCL. These errors of omission seem to be particularly common in incremental styles of development.

Collaborating with industry allowed us to observe how practitioners reacted to SCL. In our initial interaction with industry, our goal was to expose developers to SCL while minimizing disruption to their existing process. Experienced developers described the cases where new developers misunderstood intent and we encoded the intent in SCL. The general reaction was favorable to SCL provided that there are experts available to do the extraction and specification.

The commitment to using a tool like SCL often influences design. Identifying design constraints in sufficient detail to formalize in SCL forces developers to carefully re-examine their design. Not all constraints are directly expressible in SCL. In these cases it is often possible to create an alternative design for which it becomes feasible to state and check constraints. Often the alternative is also easier-to-use than the old design.

One design guideline that results in improved designs that also lend themselves to SCL is the following: Map behaviour onto syntactic structure in order to separate and localize concerns. The resultant designs are often easier to use and their constraints easier to state. The following two examples illustrate the merit of this rule, with the first one at the method level and the second at the class level.

Consider a case where an application requires the creation of many long methods in which action a is temporally followed by action b, and the only differences between these methods are the details of a and b. Rather than asking developers to create these methods from scratch, a better design creates two hook methods a and b and a template method ab, with ab calling a and then b. With this design, developers need to implement only the two hook methods, and the temporal property is automatically enforced by the template pattern. SCL can then be used to check the implementations of the two hook methods.

The next example is at the class level. A super-class exposed three methods, a, b, and c. The intended protocol for a subclass was to first call a once, followed by b once, and then zero or more c. Developers frequently made mistakes with the protocol. An alternative design was suggested for the super-class, where a method ab was created, calling a and b once and in that order. Subclasses are then required to call ab in constructors. This is a simpler rule for developers to follow, and a much easier constraint to state and check in SCL.

A. *Implementation*

We have implemented SCL for both C++ and Java. This practice has helped us understand the design tradeoffs for such systems. In both cases we reused existing front-ends: for C++ we chose *dxparscpp* from the Datrix tool suite [25], and for Java we chose Eclipse JDT [26]. Datrix is good enough for prototyping but not reliable enough for processing a large code base. Eclipse JDT enables us to run SCL on large code bases.

In general our experience with Eclipse is positive. Eclipse is designed for integrating other tools. As such, its API, in particular those for the Java model, is clear and stable. And we get a production-quality UI almost for free. Finally, Java is simpler than C++ in many aspects.

In our C++ implementation, program databases are populated with a linker program called *dxlinker* [27]. A parser extracts facts out of each C++ compilation unit and stores them in the Datrix schema. The linker then links the multiple graphs into a whole-program graph.

To separate SCL from the underlying techniques used to populate the databases, and to enable the reuse of the SCL implementation, we strove to standardize the interface to program databases. This is done by wrapping. For example, in Java SCL, access to program facts is obtained by wrapping AST nodes of Eclipse JDT with a parallel object model.

The JDT-based approach is more dynamic than the program-database-based approach. In JDT, information such as type binding may be computed during checking time by the underlying incremental Java compiler. In C++ SCL, all information required by SCL is computed statically.

Java SCL exploits lazy evaluation to improve performance. For example, Eclipse JDT provides two sets of APIs for modeling source entities: jdt.core and jdt.core.dom. jdt.core is lightweight in performance: It models only high-level entities such as projects, package fragments, classes and interfaces, and method signatures. jdt.core.dom models ASTs. In the implementation of Java SCL, by default program entities are modeled as jdt.core objects. When the corresponding AST node of a jdt.core object is needed, a search will be done behind the scene to locate it. The search is facilitated by an AST traversal mechanism provided by jdt.core.dom. This design decision is justified by the data shown in Table IX, where EqualsSignature and PublicFields take less time than other rules because they use only jdt.core objects, not AST nodes.

ASTs must be managed carefully to achieve optimal performance. An AST can consume a fair amount of memory, and is referenced by its nodes for binding information. Thus if a node is kept in memory, so is its AST. It is important to keep in memory only nodes that are absolutely necessary. In an early version of Java SCL, we cached the set of classes for each package, and thus effectively loaded all ASTs into the main memory as the checking proceeded. Because the packages were still referenced, ASTs could not be released even though they were no longer needed. The performance impact of caching classes was drastic: It became impossible to check certain rules on the 500 kloc industry code base.

B. Performance

SCL's performance requirements are quite loose. Constraints commonly violated during the incremental coding process are easy and fast to check, and can be done with every compile. Other more complex constraints are still feasible and useful so long as they fit in the time frame of a nightly build. More thorough analysis is justified if it catches expensive mistakes. For example, to check properties currently beyond SCL's capability, it takes PREfix four days to analyze Windows source code [28], and Saturn about 20 hours to check the Linux kernel for only one temporal rule [14]. Constraint checking, at least for SCL-like systems, is intrinsically parallel. If there are too many rules for one machine to process, the constraints can be distributed to multiple machines.

SCL is designed to be tractable. All sets generated in SCL specifications are finite. Although infinite sets (like Int) appear in the type system of SCL (Fig. 4), quantifying directly over such a set is impossible.

In general the complexity of evaluating an SCL constraint (see Section III-B) is driven by the number of nested quantifiers, and is roughly the product of the size of the sets associated with the quantifiers. For example, for the constraint presented at the beginning of Section III, if the number of subclasses is M and the maximal number of expressions within all member function m_D is N, then the complexity of that constraint is MN. Note that only a few of the quantified sets (such as number of classes) grow with program size. Many other sets (like the number of methods associated with a class) are bounded by programming styles.

TABLE IX
SCL PERFORMANCE ON JAVA TEST SUITE

Constraint	Time (mm:ss) Assertion Mode	Time (mm:ss) Set Mode
EqualsSignature	6:56	7:22
EqualsCorrectness	27:56	Not encoded
ArrayIndex	28:39	Not encoded
StringEquality	17:22	Not encoded
EqualsImpliesHashCode	23:51	26:33
PublicFields	5:35	5:44
DefineToString	34:30	43:36
All Constraints	94:50	

Table IX shows the performance data for running seven Java rules over the test suite of Table VIII, both individually and collectively. These rules are extracted from [22]. The first three have been introduced in Section V. StringEquality checks if two Java strings are compared using == instead of equals. EqualsImpliesHashCode checks that if a class defines equals, then it also defines hashCode. PublicFields checks if a class has public fields, and DefineToString checks if a class overrides toString. The three rules marked as "Not encoded" were precise enough that the set versions were not needed. Tests were conducted on a PowerBook G4 (1.5 GHz and 1 GB memory) running Mac OS v10.3.

Several factors can influence the performance of SCL.

First, in assertion mode where checking stops on the first example of a violation, the higher the quality of the checked code, the more code SCL needs to check for a constraint, and thus the more time is required. For example, our industry code base passes all the constraints but EqualsCorrectness and DefineToString, and thus is responsible for a significant portion of the total checking time (note that it contributes half of the size of the test suite).

Second, constraints involving ASTs are more expensive to check than those that do not. For example, EqualsSignature is faster than EqualsCorrectness. The former uses only entities above the class member level, which are directly available from jdt.core. The latter requires information about the body of equals, which can be obtained only by building flow graphs and doing control dependence analysis on ASTs. Moreover, the more AST manipulation involved, the longer the checking takes. For example, ArrayIndex took longer than EqualsCorrectness because ArrayIndex builds flow graphs and does control dependence analysis for *all* methods whereas EqualsCorrectness does so for only the equals method. Note that manipulating ASTs may involve both parsing and semantic analysis, and that it is expensive to handle expressions in terms of both CPU time and memory. Finally, as can be seen from the table, set mode constraints are more expensive than assertion mode ones.

Third, reducing the amount of source code to be checked can speed up the checking. Sometimes this can make a noticeable difference. For example, one constraint for our industry case study used knowledge of naming conventions and comments (Java doc) to exclude unnecessary classes from being checked, resulting in a significant time saving. These classes are generated by a tool, and thus follow highly regular conventions for naming and comments.

Finally, SCL checking can be further sped up by analyzing the structure of the constraints and exploiting techniques such as indexing or evaluating multiple constraints in one pass instead of separately. These remain as future work.

C. Accuracy and Expressiveness

The accuracy of an error-detection tool can be measured in terms of the number of false negatives and false positives that the tool generates; false negatives are errors missed by the tool (missed errors), and false positives are errors mistakenly reported by the tool (non errors). Three factors, the expressiveness of the tool, the way in which specifications are written, and a specifier's knowledge about a design, influence the accuracy of the tool.

Increasing the expressiveness of a specification language allows it to say more about a program, and reduces both false negatives and false positives. The expressiveness of SCL can be improved by adding new functions, and optionally, new types. The syntax of SCL remains the same, but the internals of the evaluator need to be modified to support the new functions; often the underlying program model also has to be augmented. However, there is a limitation on the expressiveness of SCL. It cannot in general express constraints about values, for example, that a certain state invariant must hold at a certain point in a program.

Like all other static analysis tools, SCL cannot be completely accurate. However, our experience is that the number of false positives is small enough that they have not been an issue; this is probably due to the quality of the systems we used to test SCL. Missed errors, on the other hand, are known to be hard to "count"; one way to reduce them is to cover the design space with more SCL rules.

The accuracy of SCL rules can be improved by exploiting usage patterns and design knowledge. We also provide mechanisms to separate errors from warnings and information.

Usage patterns detected from source code and other artifacts can make SCL more accurate. For example, EqualsImpliesHashCode could generate false positives because the presence of equals in a class does not necessarily require hashCode unless objects of the class are added into a hash-based collection. Thus strengthening the constraint with such usage patterns can avoid false alarms and improve its accuracy. Another example concerns detecting singleton classes. One heuristic we use is to see if the Java doc of a class contains the keyword singleton or if the class defines a getInstance method. The knowledge of a singleton class can then be used to exclude the class from being checked for EqualsCorrectness.

A related feature provided by Java SCL is set mode constraints, where a constraint can be rephrased as a set comprehension. Instead of only the first instance falsifying an assertion constraint, a set mode constraint returns all counter-examples, but the checking time can be longer. A common use of set mode constraints is to provide more accurate feedback to the user: When it is uncertain whether a violation of a constraint is an error, instead of asserting it as an error, a set mode constraint can be used to provide warnings or simply information. For example, in its crude form, a rule that recommends the definition of toString in all classes [22] should be phrased as either a warning or just a reminder because many classes, such as utility classes, do not have to define toString. Before the accuracy of such rules is improved, they can be phrased as set mode constraints.

VII. Related Work

Attempts to capture design intent have a considerable history [29], [30]. The earlier work by Minsky [31] on law-governed architecture models not only the architectural properties of the system, but the structure of the team and the process it uses. Constraints can take into account process, which means that one could also talk about management or organizational intent. A somewhat different approach is used in the IntensiVE system [32], which provides multiple views of a system. A view is a set of artifacts that satisfy a predicate, and may have alternative descriptions. Constraints are expressed as the consistency requirement that alternative descriptions of a view must represent the same set. It is not clear how the expressive power of IntensiVE compares to SCL. Furthermore, expressing constraints directly via the logic of SCL seems more natural than expressing them as consistency requirements on views.

Work on documenting and checking design patterns and software frameworks, for example [33], [34], [35], has the same general goal as SCL, but with less formality.

A different approach to detecting design errors is analyzing designs represented in notations other than programming languages. Examples include Alloy (a relational, first-order logic) [36], automated analysis of requirements [37], and consistency checking between requirements and designs [38]. This approach can provide a good understanding of the systems to be built in a relatively cheap way. A risk is that design models may miss features of, or over-simplify, the real systems. In contrast, SCL works directly on source code.

A. Error-detection Tools Closer to SCL

In [39], Engler proposes to extend compilers to leverage application-specific semantics for semantic checking, optimizations, and transformations (hence the name Meta Compilation). His follow-up work on metal/xgcc [40], [41] specifies desired program behavior as a finite state machine and traces relevant program (and variable) states inter-procedurally along execution paths. State transitions are triggered by source code patterns such as function calls. Errors are identified if the state machine enters an erratic state. SCL shares the same general objective on semantic checking with metal/xgcc but there are some differences. First, they are applied to different domains and languages; metal/xgcc works on C and finds many errors in systems software, whereas SCL is tested mostly on GUI frameworks and C++ and Java programs. Object-oriented programming languages allow program source to embody more design information, in a way amenable to checking, than does C, hence there are more opportunities for semantic checking. Secondly, the tools operate in complementary domains; metal/xgcc finds behavioural errors, such as memory errors, locks, and interrupt management. SCL has focused on domain-specific, structural design errors, although SCL also has the same kind of support for flow-analysis and call graph as metal/xg++ does [40]. SCL is particularly good at asserting structural constraints such as "X must appear in Y," which metal/xg++ cannot express.

B. Other Error-detection Tools

SABER [10] detects latent errors in J2EE-based applications. FindBugs [42] detects correctness and performance related bugs from Java programs. Both tools are based on byte-code analysis engines. SCL works on source code, which contains more information, such as comments, than byte-code. Neither of these tools has a specification language like SCL. SABER defines new rules by parameterizing existing rule templates, which limits its expressiveness.

Some rules, such as temporal safety properties and safe array bounds, require deeper analyses, which are typically path-sensitive and able to reason about values of and correlations between expressions. Examples of such tools include ESP [11] (temporal safety properties), ARCHER [12] (array bound checker), BEAM [13] (IBM tool), PREfix [9], and Saturn [14] (temporal safety properties using SAT solvers). CQual [43] handles temporal safety properties differently with a flow-sensitive but path-insensitive, type-based approach. These systems use stronger analyses than SCL. Some of them combine path-sensitivity and some form of theorem proving to prune infeasible paths, and thus improve the accuracy of analysis. However, these tools check for only a fixed set of errors. Their lack of extensibility means that they cannot find the kind of problems that SCL can. It remains as future work to integrate some of these deeper techniques into SCL.

Another category of tools take as input both source code and formal specifications, and perform stronger checking, for example, the extended static checking (ESC) project [44], Aspect [20], and LCLint [45]. A problem with such tools is that the amount of effort required to produce specifications increases with code size. They cannot detect the kind of errors that SCL can.

C. Syntax-based AST Traversal Tools

A number of systems allow developers to add analysis routines to a compiler to traverse abstract syntax trees, including Crew's ASTLOG [46] and Devanbu's GENOA [47]. ASTLOG uses an extended version of Prolog to write extensions, and GENOA provides a private language for the same purpose. The main difference is that these systems have a weaker program model, are limited to syntax-based tree traversal, and do not have control and data flow information. They have been used mostly to query syntactic patterns such as fall-through cases and dangling else's.

D. Reverse Engineering Tools

Several reverse engineering tools, such as Grok [48], GraphLog [49], and SCA [50], can also be made to express some design intent. Since their goal often is to obtain high level views of software systems, reverse engineering tools do not require the same kind of low-level facts as SCL does, such as expressions. Grok [48], with its ability to work on any extracted fact base representable as a graph, is particularly interesting. Grok uses a typeless binary relational algebra, as contrasted to SCL's typed assertions. An interesting question is the relationship between the expressiveness of the two systems. For example, SCL can talk about ternary relations directly, while they are more convoluted to express in Grok.

Another interesting approach is the Ptidej system [51] which has the goal of using idioms, design patterns, and architectural patterns to improve the quality of object-oriented systems. Patterns are described in terms of their structure, and then approximate matches are made to detect matches and near misses to the patterns. Set mode constraints in SCL could be used similarly, but the approximate matches of Ptidej are beyond SCL.

E. Unified Modeling Language (UML)

Given the current popularity of OMG's UML [52], one may wonder why we did not take a UML meta model for C++ and Java and use OCL on top of it. A few practical obstacles prevent us from adopting this approach. First, no existing tools fully support the kind of exploration we have done with SCL; this is not surprising since our experience with implementing SCL for C++ and Java indicates that engineering such an infrastructure is likely to be challenging. Second, the UML meta model does not contain sufficient information needed for specifying design constraints: although expressions are modeled, there is no support for name resolution and, more seriously, flow analysis. These are not a problem for UML since it is a design modeling language and thus can focus on high level entities, but they are critical for SCL. Third, in general OCL reasoning is not tractable, while SCL is designed to be. Fourth, OCL's design and syntax are arguably not ideal for us to adopt. For example, OCL's three collection operations, *select*, *collect*, and *iterate*, can be simply modeled by SCL's set comprehension; its style of cascading expressions makes it hard to recognize the scope of quantifiers. SCL is designed to be close to standard logic notations. One benefit of this design decision is that we could reason about SCL specifications in formal systems such as Mizar [18].

VIII. Summary and Future Work

SCL is based on sound fundamentals. It is expressive enough to be useful while still being feasible to machine check million-line production systems. Beyond design constraints, SCL is also useful in catching implementation errors, and enforcing style rules for maintainability and coding conventions. SCL constraints can be reused since they are expressed declaratively and de-coupled from any particular checking tool. SCL is a first step toward formalizing design intent.

Performance remains an issue. Although the checking cost scales roughly linearly with the rule count, there are some obvious places where cached partial results from one constraint evaluation can be used to speed up later ones.

When a rule fails, the root cause is not always clear. SCL's ability to generate diagnostic messages at points of failure inside a constraint can be improved. More complete diagnosis needs to inspect the broader context of a constraint. SCL also needs to be made more developer-friendly. In particular, like all logics, SCL's expressiveness comes at the expense of a rather heavyweight notation. To assist average developers in using SCL we are looking at various techniques, such as wizards, constraint patterns, and online help.

Not all rule failures are signs of violated intent. The user needs to detect false positives and then adjust the constraint to suppress them in future analyses. Handling the false positives can be done by adjusting the rule itself (in a case where the rule was too broad), dealing with the special case by adding a filter, or perhaps by annotating the code to indicate that this particular case is legal. Often, it may be better to alter the architecture or coding style to permit a simpler constraint, since a complex rule may be indicative of an overly complex design. How to strike a balance between these is an open question.

There is the larger question of how to deal directly with behaviour and functional intent. Except for some limited flow graph analysis, SCL cannot directly deal with functional intent. To capture behaviour more directly in SCL requires deeper code analysis capabilities. Since much of intent is behavioural, we want to be able to give temporal logic type of constraints. We might also want static analysis to extract code fragments to be sent through the verification process. How much behaviour can or should be represented in a model resulting from static analysis is an open issue.

SCL should in principle work on dynamically typed languages such as Smalltalk and Perl. But the amount of type information that can be automatically discovered is likely dependent on programming styles. It is not clear what design intent constraints look like in these languages.

Finally, it remains to be demonstrated that code conformance with design intent is actually a cost-effective way to produce high quality code. This can only be done through further industrial practice.

Acknowledgments

We thank the Natural Sciences and Engineering Research Council of Canada (NSERC), Alberta Science and Research Authority (ASRA), and the Alberta Ingenuity Fund (AIF) for research funding; plus Andrew J. Malton, Piotr Rudnicki, John Shillington, Kenny Wong, and many others for their discussions and support. We also thank the referees for their thorough reviews and comments.

References

[1] C.-H. Shih and B. Anderson, "A Design/Constraint Model to Capture Design Intent," in *SMA '97: Proceedings of the fourth ACM symposium on Solid Modeling and Applications*. New York, NY, USA: ACM Press, 1997, pp. 255–264.

[2] L. Horvath and I. Rudas, "Human Intent Description as a Tool for Communication between Engineers," in *SMC '99: Proceedings of the 1999 IEEE International Conference on Systems, Man, and Cybernetics*, vol. 2, 1999, pp. 348–353.

[3] ——, "Modeling Behavior of Engineering Objects Using Design Intent Model," in *IECON '03. The 29th Annual Conference of the IEEE Industrial Electronics Society*, 2003, pp. 872–876.

[4] L. L. Constantine and L. A. D. Lockwood, *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. ACM Press, 1999.

[5] R. Biddle, J. Noble, and E. Tempero, "Essential Use Cases and Responsibility in Object-Oriented Development," in *CRPITS '02: Proceedings of the twenty-fifth Australasian conference on Computer science*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2002, pp. 7–16.

[6] M. Fagan, "Design and Code Inspection to Reduce Errors in Program Development," in *IBM System Journal*, vol. 15, no. 3, 1976, pp. 182–211.

[7] D. P. Freedman and G. M. Weinberger, *Handbook of Walkthroughs, Inspections, and Technical Reviews*. New York, NY, USA: Dorset House, 1990.

[8] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey, "Structural Redocumentation: A Case Study," *IEEE Software*, vol. 12, no. 1, pp. 46–54, 1995. [Online]. Available: citeseer.ist.psu.edu/article/wong95structural.html

[9] W. Bush, J. Pincus, and D. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," *Software: Practice and Experience*, vol. 30, no. 7, pp. 775–802, 2000.

[10] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. D. Johnson, A. Kershenbaum, and L. Koved, "SABER: Smart Analysis Based Error Reduction," in *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM Press, 2004, pp. 243–251.

[11] M. Das, S. Lerner, and M. Seigle, "Path-sensitive Program Verification in Polynomial Time," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[12] Y. Xie, A. Chou, and D. Engler, "ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors," in *Proceedings of ACM SIGSOFT 2003 Conference on Foundations of Software Engineering*, 2003.

[13] D. Brand, "A Software Falsifier," in *Proceedings of IEEE 2002 International Symposium on Software Reliability Engineering*, 2002.

[14] Y. Xie and A. Aiken, "Scalable Error Detection Using Boolean Satisfiability," in *Proceedings of POPL 2005*, 2005.

[15] D. Hou, "FCL: Automatically Detecting Structural Errors in Framework–Based Development," Ph.D. dissertation, University of Alberta, Edmonton, Alberta, Canada, December 2003.

[16] D. Hou, H. J. Hoover, and P. Rudnicki, "Specifying Framework Constraints with FCL," in *CASCON 2004*, Toronto, Canada, October 2004.

[17] ——, "Specifying the Law of Demeter and C++ Programming Guidelines with FCL," in *Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'04)*, Chicago, IL. USA, September 2004.

[18] The Mizar Organization. Mizar Project. [Online]. Available: http://www.mizar.org

[19] K. Pingali and G. Bilardi, "Optimal Control Dependence Computation and the Roman Chariots Problem," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 3, pp. 462–491, May 1997.

[20] D. Jackson, "Aspect: Detecting Bugs with Abstract Dependences," *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 2, pp. 109–145, April 1995.

[21] G. Shepherd and S. Wingo, *MFC Internals: Inside the Microsoft Foundation Classes Architecture*. Addison Wesley, 1996.

[22] J. Bloch, *Effective Java Programming Language Guide*. Addison-Wesley, 2001.

[23] J. Newcomer. FlounderCraft Ltd: MVP (Microsoft Valued Professionals) Tips, Techniques, and Goodies. [Online]. Available: http://www.flounder.com

[24] Microsoft. How to Display Tool Tips After Calling EnableToolTips. [Online]. Available: http://support.microsoft.com/kb/q140595/

[25] R. C. Holt, A. E. Hassan, B. Lague, S. Lapierre, and C. Leduc, "E/R Schema for the Datrix C/C++/Java Exchange Format," in *Proceedings of Working Conference on Reverse Engineering*, 2000, pp. 349 – 358.

[26] Eclipse Foundation. JDT: Java Development Tools. [Online]. Available: http://www.eclipse.org/jdt

[27] D. Hou and H. J. Hoover, "Source-level Linkage: Adding Semantic Information to C++ Factbases," May 9 2006, submitted.

[28] J. Larus. Righting Software. SE Software Engineering Conference Keynote (PowerPoint). [Online]. Available: http://research.microsoft.com/l̃arus

[29] Y.-H. Lin and S. Meyers, "CCEL: The C++ Constraint Expression Language–An Annotated Reference Manual (Version 0.5)," Department of Computer Science, Brown University, Tech. Rep. CS–93–23, 1993.

[30] B. 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.

[31] N. H. Minsky, “Law-Governed Regularities in Object Systems; Part 1: An Abstract Model,” *Theory and Practice of Object Systems (TAPOS)*, vol. 2, no. 1, 1996.

[32] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts, “Co-Evolving Code and Design with Intensional Views: A Case Study,” *Computer Languages, Systems & Structures*, vol. 32, no. 2-3, pp. 140–156, July-October 2006, special issue on Smalltalk.

[33] G. Froehlich, J. Hoover, L. Liu, and P. Sorenson, “Hooking into Object-Oriented Application Frameworks,” in *Proceedings of the 1997 International Conference on Software Engineering*, Boston, Mass., May 1997.

[34] M. F. Fontoura, W. Pree, and B. Rumpe, “UML-F: A Modeling Language for Object–Oriented Frameworks,” in *Proceedings of ECOOP 2000*, 2000.

[35] G. Florijn, M. Meijers, and P. van Winsen, “Tool Support for Object–Oriented Patterns,” in *Proceedings of ECOOP’97*, October 1997, pp. 472–495.

[36] D. Jackson, “Alloy: A Lightweight Object Modeling Notation,” *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 2, pp. 256–290, April 2002.

[37] J. M. Atlee and J. Gannon, “State-based Model Checking of Event-driven System Requirements,” *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 24–40, June 1993.

[38] M. Chechik and J. Gannon, “Automatic Analysis of Consistency Between Requirements and Designs,” *IEEE Transactions on Software Engineering*, vol. 27, no. 7, July 2001.

[39] D. R. Engler, “Interface Compilation: Steps Toward Compiling Program Interfaces as Languages,” *IEEE Transactions on Software Engineering*, vol. 25, no. 3, pp. 387–400, May/June 1999.

[40] D. R. Engler, B. Chelf, A. Chou, and S. Hallem, “Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions,” in *Proceedings of OSDI 2000*, September 2000.

[41] S. Hallem, B. Chelf, Y. Xie, and D. R. Engler, “A System and Language for Building System–Specific, Static Analyses,” in *Proceedings of PLDI 2002*, Berlin, Germany, June 17–19 2002.

[42] D. Hovemeyer and W. Pugh, “Finding Bugs Is Easy,” in *Companion of OOPSLA 2004*, 2004, onward! track.

[43] J. Foster, T. Terauchi, and A. Aiken, “Flow-sensitive Type Qualifiers,” in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[44] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, “Extended Static Checking for Java,” in *Proceedings of PLDI’02*, Berlin, Germany, June 17–19 2002.

[45] D. Evans *et al.*, “LCLint: A Tool for Using Specifications to Check Code,” in *Proceedings of FSE’94*, New Orleans, LA, USA, December 1994.

[46] R. Crew, “ASTLOG: A Language for Examining Abstract Syntax Tree,” in *Proceedings of the USENIX Conference on Domain–Specific Languages*, October 1997, pp. 229–242.

[47] P. T. Devanbu, “GENOA-A Customizable, Front–End Retargetable Source Code Analysis Framework,” *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 2, pp. 177–212, April 1999.

[48] R. C. Holt, “Structural Manipulations of Software Architecture Using Tarski Relational Algebra,” in *Working Conference on Reverse Engineering*, 1998, pp. 210–219. [Online]. Available: citeseer.ist.psu.edu/holt98structural.html

[49] A. Mendelzon and J. Sametinger, “Reverse Engineering by Visualizing and Querying,” *Software–Concepts and Tools*, vol. 16, pp. 170–182, 1995.

[50] S. Paul and A. Prakash, “A Query Algebra for Program Databases,” *IEEE Transactions on Software Engineering*, vol. 22, no. 3, pp. 202–217, March 1996.

[51] O. Kaczorol, Y.-G. Guéhéneuc, and S. Hamel, “Efficient Identification of Design Patterns with Bit-vector Algorithm ,” in *Proceedings of the* 10^{th} *European Conference on Software Maintenance and Reengineering*, G. A. di Lucca and N. Gold, Eds. IEEE Computer Society Press, March 2006.

[52] Object Management Group. Unified Modeling Language 2.0. [Online]. Available: http://www.uml.org

Daqing Hou holds the BSc (1992) and MSc (1995) degrees of software from Peking University, China, and a PhD (2003) in computing science from the University of Alberta. His main technical interests include software design and program analysis, specifically the nature of software structures and how they can be formally represented and reasoned to support software engineering. He is currently refining and applying SCL at Avra Software Lab Inc., Edmonton, Alberta, Canada. He has taught software engineering at the University of Alberta.

H. James Hoover received a BSc (1978) in computing science from the University of Alberta, and MSc (1980) and PhD (1987) degrees in computer science from the University of Toronto. He is currently a professor in the Department of Computing Science at the University of Alberta. In parallel with academic interests, in both theoretical computer science and software engineering, he has industrial experience building design tools for the telecommunications and petrochemical domains. His main research interest is in the pragmatic application of formal methods to software construction.