Using Structural Constraints to Specify and Check Design Intent in Source Code — Ph.D. Dissertation Synopsis —

Daqing Hou
Department of Computing Science
University of Alberta
Edmonton, Alberta Canada T6G 2E8
daqing@cs.ualberta.ca

Abstract

Developers often fail to respect the intentions behind a design due to poor communication of design intent. SCL (Structural Constraint Language) helps capture and confirm aspects of design intent by using structural constraints on a program model extracted through static analysis. The original designer expresses design intent in terms of constraints on the program model using the SCL language, and the SCL conformance checking tool examines developer code to confirm that the code honors these constraints. This thesis presents the design of the SCL language and its checker, a set of practical examples where SCL has been applied, and our experience. SCL has a formal foundation, supports a wide range of design intent, is extensible for additional expressive power and checking capabilities, scales to a million lines of code, and is relatively easy to use.

1. Introduction

Managing design intent is a key activity during software construction. Software developers continuously ask questions about intent: Does this code really do what I want it to do? Is this code easy to modify or extend? Can I use this code in a different context? Am I using this interface in the way the author envisaged? Can we satisfy management's desire for adaptability to serve new business opportunities? It is the job of the developers to make the many mutually-dependent design decisions required to express these intentions in software and convert them into code precisely.

Effective communication of design intent facilitates both software evolution and reuse. To modify a piece of software, one must ensure that the change does not inadvertently affect other parts of the software. Similarly, when adding a new feature to an existing design, one must demonstrate that the extension respects the original design intent. As software evolves, such changes and extensions tend to deteriorate software design and increase software complexity, unless specific actions are taken to prevent such deterioration from happening [1].

Our goal is to create a tool that enables developers to capture intent and assist them in checking that their code continues to comply with that intent. We want the tool to work on source code directly and automatically, in much the same way as a compiler does. But different from a compiler, our tool is open and, thus, developers can add new rules to express intent specific to their own applications. The result is a specification language SCL (Structural Constraint Language) and its associated checker.

SCL treats a program as structures that consist of facts extracted from source code. Examples of such facts include the inheritance relation between two classes, the containment relation between a class and its members, and the attribute that a field is final. To express more sophisticated intent, SCL uses control and data flow and dependences. SCL can be extended by adding extra facts to the fact-base and new operations to the SCL language.

Structural constraints are assertions on these facts. The notion of structural constraints is not entirely new, and there are already structural constraints in programming languages. For example, in Java, a final method cannot be overridden in a subclass. However, in a domain-specific setting, there are often more constraints than what are currently being expressed in a programming language and checked by a compiler. We generalize such assertions, defining the SCL language for developers to express constraints particular to their own design.

Compared to previous work like [2, 3, 4], this thesis makes two contributions. The first is the design of the SCL specification language and checker. The SCL language adopts a first-order logic framework, which makes it a conceptually simple matter to extend SCL for additional expressive power and checking capabilities. The second is the application of SCL to a set of real-life examples where SCL helps enforce design intent. These examples show that SCL can support a wide range of design intent.

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 properties of the source code. Thus the original name for SCL was FCL (Framework Constraint Language) [5].

2. What is SCL and how does it work?

In this section, we introduce SCL by a C++ example, the architecture of the SCL checker, the evaluation of SCL rules, and the computational complexity of the evaluation.

2.1. An example

For certain kinds of classes, good 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 a virtual member function m.

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

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

Our constraint above, expressed in terms of this class, is: If a subclass of B overrides m, then the override must explicitly call B::m. In SCL we can specify this constraint as follows:

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

In this specification, subclasses, class, method, and exprs are functions on the structure of program source. 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
}
```

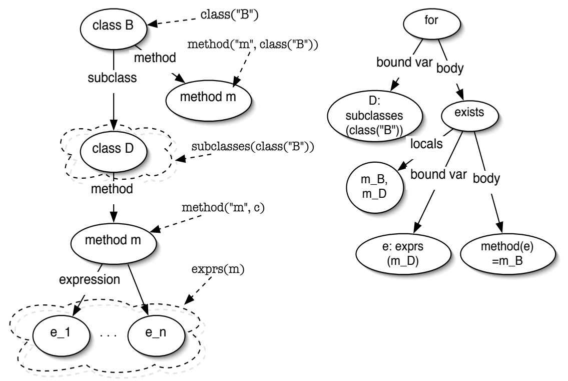

Figure 1. Evaluating SCL constraints. Left: program facts as a graph; dotted areas indicating sets of entities. Right: parse tree for the SCL constraint

To detect violations of this constraint, the C++ source is converted into a graph that is suitable for checking relationships between program elements. Then an SCL evaluator evaluates the SCL specification against the graph and emits diagnostic messages when anomalies are identified. Fig. 1 illustrates a graph representation of the example code and how the SCL specification is evaluated on the graph.

If the SCL specification is not satisfied, diagnosis for the problem will be provided. We have implemented two forms of diagnosis: via command-line output in the C++ SCL, and a graphical user interface in the Java SCL. Java diagnostics use syntax highlighting, decorations, and tooltips to provide feedback information. Optionally, HTML documents can be displayed to explain a really complex scenario.

2.2. SCL architecture and rules evaluation

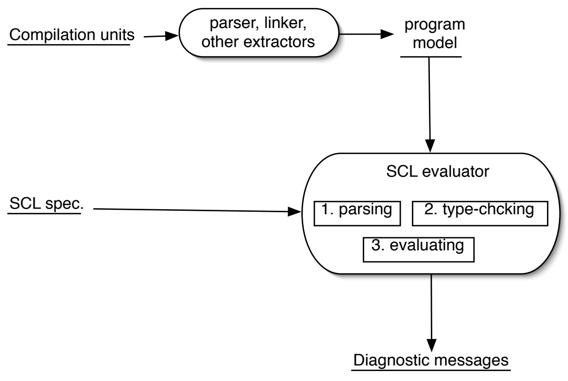

Figure 2. SCL architecture

At the core of the architecture in Fig. 2 is the SCL evaluator, which requires two inputs, an SCL formula and a program model (a.k.a. a program fact-base).

A constraint may require information transcending program units such as classes and packages. Some may even require 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. At evaluation time, the model is mapped to an object-oriented representation, where 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 formula 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 tuple of the domain. Finally primitive functions are evaluated directly against facts in the program model. Obviously there are opportunities to speed up the checking using standard techniques like caching, indexing, and even the static analysis of SCL rules themselves.

3. What have been done using SCL?

We have performed several studies to evolve SCL and to understand where and how it can be best applied, These studies show that SCL can be used to specify a wide range of design intent, and that SCL works best when there is a relatively straight-forward mapping between design intent and software structure.

3.1. Study 1: MFC (Microsoft Foundation Classes)

MFC is the initial motivation for SCL and has supplied many examples for us to work out the details of SCL. The MFC examples we use are based on over a decade of programming and consulting experience of an MFC expert [6]. They are common problems that developers encounter when using MFC to build graphical user interfaces. We develop and test 9 SCL rules on sample MFC programs.

In this study, our goal is to understand what is the right design for SCL, what features it should have in order to support these examples, and how to extend SCL systematically for additional expressiveness and checking capabilities.

This study shows that SCL can support a wide range of design intent. First, there are functional constraints such as an overridden method must call its superclass version, or the value of an expression must depend on that of a certain variable at a certain point. Second, to support modifiability and maintainability, code must follow a certain structure. This is especially true in the context of a specific project. SCL can be used to enforce such intent. Third, SCL is used to capture the use of deprecated API. Forth, an important source of errors is to forget to call a certain function. SCL is effective in catching such omissions. Finally, a designer can use SCL to notify users “flaws” in the design so that they can stay away from such flaws. In industry, it is difficult to correct a flawed design after it is released to the market. SCL can help mitigate this problem.

3.2. Study 2: C++ programming rules

Modern programming languages are providing increasingly rich features to developers. Consequently, another source of programming rules is from languages like C++ [7, 8] and Java [9]. In this study, we use SCL to specify 17 rules for C++. For example, one rule requires that C++ constructors must not call virtual member functions. A lesson learnt is that in order for SCL to be expressive, the fact-base must contain sufficient details. For example, SCL models both expressions and flow information, and provides extra facts that are not available from parsing, such as that an expression is bound to a user-defined conversion, or that a copy constructor is invoked implicitly.

3.3. Study 3: Law of Demeter and observer

In order to understand how SCL works to constrain abstract design, we use it to specify and check a design rule (Law of Demeter [10]) and a design pattern (the observer pattern [11]). We find that SCL works less effectively on abstract design than on concrete implementation, because abstract design often lacks the necessary information required to formulate SCL rules. For example, in its abstract form, the observer pattern does not specify how the state is represented and what constitutes a change to the state. Consequently, it becomes difficult to specify in SCL that notify() must be called whenever the state is changed. While a work-around does exist in this particular case, in general reasoning about abstract design in SCL needs further investigation.

4. SCL publications and current status

In addition to the dissertation [5], SCL is also documented in several other publications [12, 13, 14, 15, 16, 17]. SCL is available upon requests from the author.

We have implemented SCL for both C++ and Java. The C++ SCL is a limited proof of concept, but it has given us a good understanding of the application scope and the design tradeoffs for SCL and its checker. The Java SCL is based on the Eclipse Java Development Tools [18] and, thus, allows us to test SCL on production code bases. Using the Java SCL, we have experimented with over a million lines of code [16]. We learn that SCL can scale up for practical use in developing large software. We also learn lessons on how to write more precise SCL rules, to avoid false positives, and how to design in such a way that facilitates the use of SCL. Finally, the Eclipse environment and the SCL checker now give us an initial framework for SCL support for additional languages.

4.1. SCL as a query language

SCL supports not only assertions but also queries. By default, the evaluation is lazy, with the evaluation of the constraint terminating in false on the first counter-example. Such a constraint is essentially an *assertion*. In other cases, for reasons like diagnostics, one may want to see all the counter-examples that violate the same constraint, for which the assertion can be converted into a set comprehension, that is, a query, and evaluated.

Initially, SCL is designed to detect violations of design intent with assertions. When testing SCL, however, we realize that developers may not always know firmly about what is correct or wrong but they often do have clues as to what to look for in the source code. In such situations, a query mechanism will serve them better than an assertion. Thus, support for a set comprehension is added into SCL.

5. Acknowledgments

The author thanks H. James Hoover for supervising this thesis, Andrew J. Malton, Piotr Rudnicki, John Shillington, Kenny Wong, and many others for their discussions and support. Research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), Alberta Science and Research Authority (ASRA), and the Alberta Ingenuity Fund (AIF).

References

[1] L. Belady and M. Lehman, “A Model of Large Program Development,” *IBM Systems Journal*, vol. 15, no. 3, pp. 225–252, 1976.

[2] S. Meyers, C. K. Duby, and S. P. Reiss, “Constraining the Structure and Style of Object–Oriented Programs,” Department of Computer Science, Brown University, Tech. Rep. CS–93–12, 1993.

[3] 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.

[4] K. Mens, I. Michiels, and R. Wuyts, “Supporting Software Development through Declaratively Codified Programming Patterns,” *Expert System with Applications*, vol. 23, pp. 405–413, 2002.

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

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

[7] S. Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison–Wesley Publishing Company, 1992.

[8] ——, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison–Wesley Publishing Company, 1996.

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

[10] K. J. Lieberherr and I. Holland, “Assuring Good Style for Object–Oriented Programs,” *IEEE Software*, pp. 38–48, September 1989.

[11] E. Gamma, R. Helm, R. E. Johnson, and J. O. Vlissides, *Design Patterns-Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[12] D. Hou and H. J. Hoover, “Towards Specifying Constraints for Object–Oriented Frameworks,” in *CASCON 2001*, Toronto, Canada, November 2001.

[13] D. Hou, H. J. Hoover, and E. Stroulia, “Supporting the Deployment of Object–Oriented Frameworks,” in *Proceedings of the International Conference on the Advanced Information System Engineering (CAiSE’02)*, Toronto, ON, Canada, May 2002.

[14] D. Hou, H. J. Hoover, and P. Rudnicki, “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.

[15] ——, “Specifying Framework Constraints with FCL,” in *CASCON 2004*, Toronto, Canada, October 2004.

[16] D. Hou and H. J. Hoover, “Using SCL to Specify and Check Design Intent in Source Code,” *IEEE Transactions on Software Engineering*, June 2006.

[17] ——, “Source-Level Linkage: Adding Semantic Information to C++ Factbases,” in *International Conference on Software Maintenance*, Philadelphia, PA, USA, September 2006.

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