SCL: Static Enforcement and Exploration of Developer Intent in Source Code

Daqing Hou
ECE Department, Clarkson University, Potsdam, New York 13699
dhou@clarkson.edu

1. INTRODUCTION

Software developers spend a large fraction of their time dealing with intent, seeking to answer questions like

- Does this code really do what it is expected to do?
- Is this module used appropriately?
- Is this code easy to modify or extend?
- Can this code be used in a different context?

As a result, developers make many mutually-dependent design decisions to express intentions about software and try to convert them into code precisely.

Moreover, effective communication of developer intent is essential to successful software evolution and reuse. To modify a piece of code, the change must not inadvertently affect other parts. When adding a feature to an existing design, the addition must respect the intent of the original design. As software evolves, such changes and extensions tend to deteriorate software structure and increase complexity, unless specific actions are taken to stop the deterioration.

SCL (Structural Constraint Language) is a language and a checker aimed to help express developer intent and check that code continues to comply with the intent. SCL works on source code directly, in much the same way as a compiler. But unlike a compiler, SCL is open in that developers can use SCL to express intent specific to their applications.

SCL treats a program as a model made of facts extracted from source code. In addition to basic facts such as inheritance, containment, and attributes of program elements, to express more sophisticated intent, SCL also makes use of control and data flow and dependences. To write formulas (a.k.a structural constraints) about the program model, SCL adopts the formalism of first-order logic. Furthermore, SCL can be extended by adding to the program model extra facts obtained from other sources (e.g., dynamic analysis) and new operations to the SCL language.

An SCL formula is evaluated recursively on its structure. At each quantifier, a finite set that forms its domain is constructed. Then the sub-formula is evaluated against each tuple of the domain. Finally, primitive operations are evaluated directly against facts in the program model. If the formula is not satisfied, diagnosis can then be produced.

ICSE '2007 Minneapolis, MN, USA

We have prototyped SCL for both C++ [1] and Java [2]. The Java SCL leverages the Eclipse Integrated Development Environment and JDT [3] and, thus, is relatively reliable so that practical use and the evaluation of precision and scalability are made possible [2].

2. EXAMPLES

```
1  def Object as class("java.lang.Object")
2  def getClass as method(Object, "getClass")
3  def subtype(c, t) as (c=t | in(c, descendants(t)))
4  def isEquals(m) as (returnType(m)=boolean
   & name(m)="equals" & sizeof(params(m))=1 &
   type(ith(params(m),0))=Object & isPublic(m))
5  def isInstanceof(e, c, o) as
6  [def t as class(literalType(ith(args(e),1)))]
7  (method(e)=instanceof & var(ith(args(e),0))=o & subtype(c,
   t))
8  def isGetClass(e, o) as
9  (method(e)=getClass & var(receiver(e))=o)
10 def guarded(e, c, o) as
11 ex ctrl:  conds(e)
12 (isInstanceof(ctrl, c, o) | isGetClass(ctrl, o))
13 def callsCasting(m, c, o) as
14 ex e:  exprs(m) holds
15 [def tt as class(literalType(ith(args(e),0)))]
16 (method(e)=cast & subtype(c, t) & var(ith(args(e),1))=o
17 & guarded(e, c, o))
18 def testsNull(e, o) as
19 ((method(e)=equality | method(e)=inequality)
20 &
21 [ def lhs as ith(args(e), 0);
22 def rhs as ith(args(e), 1); ]
23 (isLiteralNull(lhs)&var(rhs)=o |
   isLiteralNull(rhs)&var(lhs)=o)
24 )
25
26 for p:  packages, c:  classes(p), m:  methods(c)
27 (isEquals(m) =>
28 [def o as ith(params(m), 0)]
29 callsCasting(m,c,o)
30 )
31
32 for p:  packages, c:  classes(p), m:  methods(c)
33 (isEquals(m) =>
34 [def o as ith(params(m), 0)]
35 for e:  exprs(m)
36 (isGetClass(e, o) =>
37 ex c:  conds(e) testsNull(c, o) )
38 )
```

Figure 1: Part of SCL specification for equals()

We use the equals() method of class Object (public boolean equals(Object o)) to illustrate SCL. We show both the enforcement and the exploration of intent related to equals().

Table 1: SCL operations used in Figure 1

Operation	Description
class(qn)	A class with qualified name qn.
method(c, n)	A method in class c with a name n.
descendants(t)	All subtypes of type t.
ith(seq, i)	The ith element of sequence seq.
params(m)	Parameters of m.
exprs(m)	Expressions of method m.
methods(c)	Methods of class c.
args(e)	Arguments of expression e.
var(i)	Variable identifier i refers to.
method(e)	Operation e invokes.
literalType(i)	Type identifier i refers to.
conds(e)	Conditions controlling e.
packages	All packages in a project.
equality	Predicate == in Java.
inequality	Predicate != in Java.

2.1 Intent Enforcement: Contract for equals()

An equals() method tests the equivalence between two Java objects. To work properly with hash-based classes such as HashSet, the equals() in a class A must satisfy the following constraints. (1) Because the type of parameter o is Object, it must be cast into a correct type in order to compare with this object. The correct type of o must be either A or an interface implemented by A. The type can be tested using either the instanceof operator or the getClass() method. But the type-testing expression must control the casting expression. (2) If getClass() is used to test the type of o, then o must not be null. Thus the call to getClass() must be guarded by a test of o not being null.

The SCL specifications for these two constraints are shown in Figure 1. Table 1 describes the operations used in Figure 1. To help structure specifications, an expression or formula can be given a name. The named formulas and expressions in Figure 1 are explained as follows. (1) Object refers to the java.lang.Object class. (2) subtype(c,t) tests if c is a subtype of t. (3) isEquals(m) tests if m matches the equals() signature. (4) isInstanceof(e,c,o) tests if e matches o instanceof c. (5) isGetClass(e,o) tests if e matches o.getClass(). (6) guarded(e,c,o) tests if e control-depends on a type testing expression involving c and o. (7) callsCasting(m,c,o) tests if method m contains an expression that casts o to either class c or an interface implemented by c. (8) testNull(e,o) tests if e matches either e==null or e!=null.

In practice, even experienced developers seem to neglect these programming obligations. For example, these constraints have been regularly checked on the SCL code base, and several incorrect equals() have been caught. We have also found many problematic instances in other production software. These projects have been heavily tested, reviewed, and used in production, yet these defects remain. In the following, we show two examples.

The org.eclipse.ant.core package is part of the Eclipse project that integrates the Ant tool into Eclipse. The equals() of its Property class fails to test that other is not 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;
}
```

Apache Lucene is an open source, full text search engine [4]. The equals() of org.apache.lucene.index.Term fails to test 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);
}
```

2.2 Intent Exploration: Other Uses of equals()

Developers may not always be sure about whether their code is right or not. One reason for this uncertainty is that code does not always contain all the information necessary for the safe conclusion that a suspicious code pattern is indeed a defect. But often they do have some clues as to what to look for in the source code. In such a situation, it can be helpful to use the set comprehension construct of SCL to help explore the source code.

Although much can be said about the correctness of equals() with relatively strong confidence, not all questions can be phrased as constraints. For example, how is the name equals overloaded in a system? Are they all public? And which ones are defined with a parameter other than Object? These questions can be easily explored with the set comprehension facility of SCL. For example, the following set comprehension selects from a Java project all the equals() that have a non-Object parameter.

```
def otherEquals as { m |
      p: packages, c: classes(p), m: methods(c)
      name(m)="equals" & type(ith(params(m),0))!=Object
     }

info(otherEquals, "This equals() is not standard.")
```

With the help of this query, we find several cases where equals is overloaded for other purposes (e.g., boolean equals(float, float)), and cases where the standard equals() calls a delegate equals() that takes the current class as its parameter type. This practice of delegation should be discouraged, or at least the delegates should have been private. However, all such equals() delegates identified in our tests are public, and used only by the standard equals(). In this case, SCL queries helped a developer better understand the use of equals in practice.

3. ACKNOWLEDGMENTS

Thanks for the anonymous reviewers' thought-provoking comments.

4. REFERENCES

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

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

[3] The Eclipse Foundation. The Eclipse Project. Last verified: Nov. 2 2006. [Online]. Available: http://www.eclipse.org/eclipse/index.php

[4] The Apache Software Foundation. Apache Lucene. Last verified: Nov. 2 2006. [Online]. Available: http://lucene.apache.org/java/docs/index.html