Specifying the Law of Demeter and C++ Programming Guidelines with FCL

Daqing Hou, H. James Hoover, and Piotr Rudnicki
Department of Computing Science
University of Alberta
Edmonton, Canada T6G 2E8

Abstract
Framework Constraint Language (FCL) was an attempt to apply static analysis to detect errors in framework usage; it can be used to specify the syntactic structures that frameworks impose on the code of framework-based applications, which can then be enforced by a checker program. FCL can also be used to encode design rules such as the Law of Demeter and programming guidelines. This paper introduces FCL and demonstrates its utility in these areas. The current version of the FCL language and associated checker is targeted at C++.

1 Introduction

Framework Constraint Language (FCL) was an attempt to apply static analysis to detect errors in framework usage. Using object-oriented application frameworks can be intricate and error-prone [8]. A framework user with shallow knowledge needs something akin to type-checking in programming languages. Towards this end we have developed a specification language called FCL and the associated mechanism for checking the conformance of framework users' code with framework constraints. The framework developer takes on the burden of specifying how to properly use the framework so that compliance by framework users can be checked mechanically. FCL has been tested through several experiments; it has been applied to architectural fragments of varying sizes: such as the Observer pattern and MFC (Microsoft Foundation Classes)'s dialog architecture [7]. We conclude that FCL can be useful to framework users.

As FCL evolves, it becomes evident that it can also be used to specify existing general design rules and programming guidelines: such as the Law of Demeter [10] (LoD henceforth) and Scott Meyers' C++ programming guidelines [13]. This paper demonstrates FCL's utility in this regard.

The current version of FCL is targeted at C++, but the basic notions apply to other strongly-typed object-oriented programming languages. C++ is an ideal candidate for the initial version of FCL. It has many features that can potentially complicate program analyses: separate compilation, inheritance, overloading, templates, exception handling, and so on. A large production framework, MFC, is publicly available [19]; extensive user experience is available in the forms of both news group discussion and essays by expert users [15]. The implementation of FCL consists of about 20,000 lines of C++. About 1,300 lines of FCL specifications have been written to date.

The rest of the paper is organized as follows: Section 2 introduces FCL; Section 3 and 4 apply FCL to specify LoD and C++ programming guidelines respectively; Section 5 compares FCL with related work and Section 6 concludes the paper and anticipates future work.

2 The FCL Language

FCL is a little language for talking about constraints on the syntactic structures of O-O programs. It is a first-order logic extended with sets and sequences: its constants include names for program elements, integers, and `global`; its term language consists of total functions reflecting the entity-relationships in the syntax of O-O programs (all syntactic elements have types, which form a subtype relation: see Section 2; the relationships between syntactic elements can be accessed through FCL functions introduced in Section 2); its atomic formulas include relational operators, set membership, subset, and predicates, such as testing whether a member function is static or not. FCL is strongly typed: all terms have static types at compile time.

Syntax

Figure 1 depicts the concrete syntax of FCL.

At the topmost level, an FCL specification consists of a sequence of interleaved declarations and formulas. Declarations can be freely interspersed among formulas so long as variables are defined before they are used. The combination of a top-level formula and all the declarations that it refers to forms *an FCL constraint*.

Each declaration binds a variable to an associated ex-

```
FCL_spec ::= Statement_list
Statement ::= Declaration | Formula
Declaration ::= Variable as Expr

Formula ::= not Formula| Formula and Formula| Formula or
Formula| Existential | Universal | Expr
Expr ::= Variable | Constant | Operation | Formula |
ExprWithVars
Existential ::= exist BVar_Declaration_list hold Formula
Universal ::= forall BVar_Declaration_list hold Formula
BVar_Declaration ::= Declaration
ExprWithVars ::= '['Declaration_list']' Expr

Operation ::= Set_op | Seq_op | Relational | FCL_fct
Set_op ::= subset(s1, s2) | belongsTo(ele,s) | s1 + s2 |
s1 - s2 | union(setOfsets) | sizeof(s) |
Set_comprehension | Set_enumeration
Set_comprehension ::= { BVar_Declaration_list | Formula |
Expr} | { BVar_Declaration | Formula}
Set_enumeration ::= '[' Expr_list ']'
Seq_op ::= member(seq, index) | first(seq) | last(seq)
Relational ::= > | >= | < | <= | =

Constant ::= true | false | Str | Int | global
Variable ::= Str
T_list ::= T*
```

Figure 1: Concrete Syntax of FCL

pression. FCL variables are assigned values at declaration time and are constant within the scope of the declaration, just as in usual predicate calculus. Logical formulas are treated as expressions that yield values of the boolean type. FCL allows one to define boolean variables with formulas as their value expressions.

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

Formulas have a mostly standard syntax, with English words in place of the conventional mathematical symbols; there are formulas of negation, conjunction, disjunction, and universal and existential quantifications.

Expressions include variable references, literal constants, function applications, and sets. Particularly FCL allows for the treatment of formulas as expressions.

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

Function applications of the form $f(e_1, \ldots, e_n)$ are probably the most frequently used expressions. FCL predefines a variety of functions. Some of them are standard set and sequence operations. Others are defined on the structure of the object model. 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`. Section 2 summarizes the FCL functions.

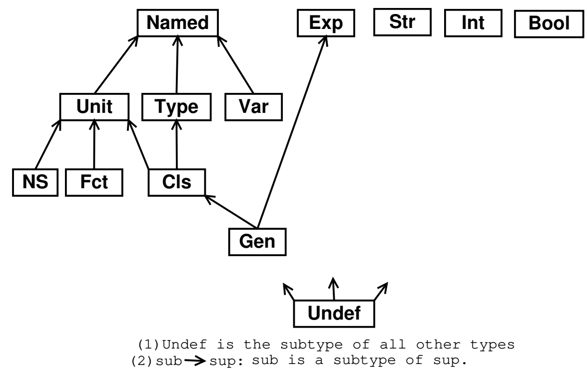

Figure 2: Basic Types and the Subtype Relation

Sets in FCL can be generated through: function applications that return a set, set comprehension, and enumeration. FCL supports two forms of set comprehension:

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

and

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

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

FCL needs sequences: an execution path consists of an sequence of expressions, a function has a sequence of parameters, and a path on an inheritance hierarchy contains all the classes from a base class to a derived one. FCL does not provide constructor functions to create new sequences from scratch; instead, sequences are return values from function applications. A sequence can be used as a set by taking the range of the sequence.

Type System
The type system of FCL includes two kinds of types: basic types (Figure 2) and compound types (sequences and sets). Basic types can be further divided into "facility" types and "domain" types. Facility types help form constraints, including:

- Str for string values,
- Int for integers,
- and Bool for boolean values.

Domain types come from the domain of programming constructs, including:

- Exp for expressions,
- Named for named program elements,
- Var for variables,
- Type for types.
- Unit for program units,
- NS for namespaces,
- Fct for functions,
- Cls for classes,
- and Gen for types generated through template instantiations.

The current FCL covers only a subset of the object model; for instance, arrays and template-generated functions are not yet included. But as it is FCL already demonstrates its utility.

A note on our choice of terminology on Unit for program units: [1] distinguishes between *program units* and *scopes*. *program units* are constructs helping structure programs, such as functions, classes, and namespaces, while the *scope of a name* is the portion of programs over which the name may be used. Thus we prefer *Unit* over *scope*.

FCL Functions

This section introduces a representative subset of FCL functions used later. The following code fragment will be used in examples.

```
class Customer{
public:
   string statement(){
        for (int i=0; i<_rentals.size(); ++i)
           result +=  _rentals[i]->getMovie()->getTitle();
   }
private:
   vector<Rental *> _rentals;
};

class Rental{
public:
    Movie * getMovie();
private:
    Movie * _movie;
};

class Movie{
public:
    Movie(const string & title);
    string getTitle();
};
```

In what follows: FCL functions will be grouped into sections; each of them will be presented in the form of a signature followed by an explanation. F T denotes all finite subsets of elements of type T while Seq T all finite sequences of elements of type T.

Constructors: A constructor allows for referring to an existing program element.

- `class`: Str $\times$ Unit $\rightarrow$ Cls
- `var`: Str $\times$ Unit $\rightarrow$ Var
- `function`: Str $\times$ Unit $\rightarrow$ F Fct
  The arguments specify the name of the program element and the program unit where the element resides. At evaluation time these functions will be bound to their corresponding syntactic elements. For instance, the member `Movie::getTitle` can be specified as `func-tion("getTitle", class("Movie", global))`. The constant `global` can be omitted; thus it can also be rewritten as `function( "getTitle", class("Movie"))`. Note that since functions can be overloaded, the return type of `function` is a set of functions.
- `ptr/ref`: Type $\rightarrow$ Type
  `ptr` (`ref`) returns a pointer (reference) type to the argument type. For example, `ptr(Movie)` denotes the type `Movie *`.

Entity Properties:

- `name`: Named $\rightarrow$ Str
  Returns the name of a program element.
- `isPrivate/isProtected/isPublic`: Var | Fct $\rightarrow$ Bool
- `isVirtual`: Fct $\rightarrow$ Bool
- `isStatic/isConst`: Var | Fct $\rightarrow$ Bool
- `type`: Exp | Fct | Var $\rightarrow$ Type
- `ptdType/ptdTypeStar`: Type $\rightarrow$ Type
  If the argument is a pointer type, then `ptdType` returns the type it points to while `ptdTypeStar` returns its base type. Otherwise, returns the type itself. For example, if the argument is `Rental **`, then applying `ptdType` to it yields the type `Rental *` while `ptdTypeStar` `Rental`.

Containment: Program units like classes and functions may contain other program elements; each program element is enclosed by a sequence of program units called its *context*.

- `class`: Unit $\rightarrow$ F Cls
- `var`: Unit $\rightarrow$ F Var
- `function`: NS | Cls $\rightarrow$ F Fct
- `exp`: Unit $\rightarrow$ F Exp
  These functions return the set of elements contained by the respective program units. For example, the return value of `function` is the set of functions defined within a namespace or class; applying `function` to class `Movie` will return a set with two functions: `Movie` and `getTitle`.

- `param`: Fct → Seq Var
  `param` returns the sequence of parameters of a function.
- `unit`: Named | Expr → Seq Unit
  `unit` returns the sequence of program units enclosing the argument, that is, its context. For example, the context for the data member `_rentals` is the class `Customer` and `global` while the context for the expression `i=0` is the member function `statement`, the class `Customer`, and `global`.

Expressions: When evaluated, an expression denotes a value and may generate side-effects. The following FCL functions examine the syntax trees of expressions.

- `arg`: Exp → Seq Exp
- `receiver`: Exp → Exp
  These two functions return the sub-expressions of their arguments: `arg` returns the sequence of arguments while `receiver` the receiver expression; if the argument has no receiver expression, for instance, in the case of an addition like `a + b`, then "undefined" is returned. The arguments of `_rentals[i]` are `_rentals` and `i` while its receiver is `_rentals`.
- `parent`: Exp → Exp
  `parent` returns the parent expressions of the argument. If the argument has no parent expression, then returns itself. For example, the parent of the expression `i` in `_rentals[i]->getMovie()` is `_rentals[i]`.
- `isBranch/isReturn`: Exp → Bool
  These two functions will return true if their arguments are conditional expressions or the argument of a return statement respectively. For example, applying `isBranch` to the expression `i<_rentals.size()` in the member function `statement` will yield true.
- `exp`: Exp → F Exp
  `exp` returns the set of all sub-expressions contained by the argument. For example, expression `_rentals[i]- ->getMovie()` contains 4 sub-expressions: the expression itself, `_rentals[i]`, `_rentals`, and `i`.
- `refd`: Exp → Named
  If the argument is a name reference, returns the entity it refers to. Otherwise, returns "undefined." For example, the names `_rentals` and `i` in `_rentals[i]` refer to the data member `Customer::_rentals` and the local variable `i` respectively; the name `Rental*` in `vector<Rental *>` refers to the pointer type to `Rental`.
- `function`: Exp → Fct
  `function` returns the function that the argument is statically bound to. For example, the expression `_rent- als[i]->getMovie()` is bound to the member `Rental- ::getMovie`.
- `var`: Exp → Var
  If the argument is a name reference to a variable, returns the variable; otherwise returns "undefined." For example, the names `_rentals` and `i` in expression `_rentals[i]` refer to the member `Customer::_rentals` and the local variable `i` respectively.

Inheritance:

- `subclass/descendant/superclass/ancestor`: Cls → F Cls
  Returns the set of subclasses (descendant/super/ancestor classes) of the argument class.

Type query and downcasts:

- `isClass/isGenerated/isType`: Named → Bool
  Query if a named element is a corresponding type.
- `type`: Named → Type
- `class`: Named | Type → Cls
- `generated`: Named | Type | Cls → Gen
  Downcasts the argument to a corresponding type. If the downcast fails, then "undefined" is returned.

Miscellany: In what follows Any denotes any FCL types.

- `isDefined`: Any → Bool
  If the argument is "undefined," returns `false`; otherwise returns `true`.
- `print/println`: Any → Bool
  Prints the textual representation of the argument and returns `true`.

FCL's Treatment of "undefined"

There are several cases that can lead to a value of "undefined." Casting an element of one type to another can generate values of "undefined." Asking for a receiver expression from an arithmetic expression will also result in "undefined."

For any function application with the value "undefined" as an argument, if the return type of the function is basic types but not boolean, then the result will be a value of "undefined." But if the return type is boolean, then the function application will return `false`. If the return type is a compound type, that is, a sequence or a set, then the function application will return empty sequence or empty set respectively.

3 The Law of Demeter in FCL

LoD is a style rule aiming at improving the quality of object-oriented design. Function calls between classes cause couplings; LoD states which couplings are acceptable and which should be avoided. Phrased in terms of source code, the law requires that each member function of a class depends only on other member functions of the class, classes of its parameters, the class itself, classes of locally-created objects, classes of data members (including those inherited), and return types of member functions. These classes are called *preferred suppliers* of the member function.

In practice exceptions may exist for LoD: some classes may be deemed as “transient” and thus their violations of LoD acceptable; global variables may be used for reasons such as efficiency; some violations may be considered acceptable because the used interfaces are stable. Thus it is necessary to produce project-specific versions of LoD to accommodate such variations and to suppress spurious error messages.

Figure 3 shows an FCL specification for LoD that checks classes defined in the global namespace. The specification is mostly straightforward; we comment only on three points: one, the set `NoCheck` can be used to define all transient classes that require no checking; two, we count all the argument classes of a *preferred* template class as *preferred suppliers*; three, we do not check global variables for violations. Also note that for space reason this version does not include the preferred supplier classes from base classes.

```
// Define the set NoCheck here!
AllClasses as class(global)-NoCheck;
forall c:  AllClasses holds
[
  // classes thru data members
  classesByMemb as { v:  var(c) |
        isClass(ptdTypeStar(type(v))) |
          class(ptdTypeStar(type(v)))
  };
   // classes thru member functions
   classesByFunction as { f:  function(c) |
         isClass(ptdTypeStar(type(f))) |
           class(ptdTypeStar(type(f)))
   };
   // types thru templates instantiations
   gen as { v:  var(c) |
         isGenerated(ptdTypeStar(type(v))) |
           generated(ptdTypeStar(type(v)))
   };
   templArgs as union({ t:  gen | true |exp(t)})
   classesByTempl as { e:  templArgs |
         isClass(ptdTypeStar(type(refd(e)))) |
           class(ptdTypeStar(type(refd(e))))
   }
 ]
 forall m:  function(c) holds
 ( isStatic(m) or
 [
   classesByParam as { v:  param(m) |
         isClass(ptdTypeStar(type(v))) |
           class(ptdTypeStar(type(v)))
   };
   classesByNew as { e:  exp(m) |
         [objClass as class(refd(arg(e,0)));]
         name(e)="new" and isClass(objClass)|
         class(refd(arg(e,0)))
   };
   preferredSuppliers as classesByParam+[c]+
     classesByMemb+classesByFunction+
     classesByNew+classesByTempl+gen;
 ]
 forall e:  exp(m) holds
 [r as receiver(e);]
 (
   // The receiver is a global variable
   [referredVar as var(r)]
   first(unit(referredVar))=global
   // Otherwise...
   or
   [
     typeOfReceiver as type(r);
     baseType as ptdTypeStar(typeOfReceiver);
   ]
     ( not isDefined(baseType) or
     belongsTo(baseType, preferredSuppliers) )
   or println(e)
 ) )
```

Figure 3: LoD in FCL

4 C++ Programming Guidelines in FCL

Object-oriented programming languages such as C++ and Java raise their level of abstraction by absorbing more sophisticated design ideas. As a result, guidelines and rules on how to properly use these languages are formulated [13, 2]. The value of specifying and enforcing these guidelines has long been recognized [9, 11]. In [14] the authors report an evaluation of 5 C++ compilers and 8 static analysis tools on enforcing 34 C++ programming guidelines: the best one can support only 19; the most important reason for failure is the lack of expressiveness. In what follows we use FCL to specify 6 of them which most tools fail to handle.

- Constructors and destructors must not call any virtual functions.

```
forall c: class(global) holds
     [ ctorAndDtor as {f: function(c) |
         name(f)=name(c) or name(f)="~"+name(c)}
     ]
     forall cd: ctorAndDtor, e: exp(cd) holds
         not isVirtual(function(e))
```

- If a class has any pointer data members, then it must define a copy constructor and an assignment operator.

```
forall c: class(global) holds
[pvar as {v: var(c) | not ptdType(type(v))=type(v)}]
(
  sizeof(pvar) = 0 or
  exists f: function(c) holds
        [p as param(f); p1 as member(p,0)]
        ( name(f) = "operator=" and sizeof(p) = 1 and
         type(p1) = ref(c) and isConst(p1) and
         type(f) = ref(c) and isConst(f) )
  and
  exists f: function(c) holds
        [p as param(f); p1 as member(p,0)]
        ( name(f) = name(c) and sizeof(p) = 1 and
         type(p1) = ref(c) and isConst(p1) )
```

```
)
```

- An assignment operator of a class `C` must have "`const C&`" as the types of both its sole parameter and return value; it must test for self-assignment and return only "`*this`".

```
forall c: class(global) holds
[asgn as function("operator=", c);
 p as member(param(asgn),0)
]
(
  sizeof(asgn)=0
  or
  type(p)=ref(c) and isConst(p) and
  type(asgn)=ref(c) and isConst(asgn)
  and
  exists e: exp(asgn) holds
      ( isBranch(e) and name(e)="==" and
      [lh as member(arg(e),0);
       rh as member(arg(e),1)]
      (name(lh)="this" or name(rh)="this") )
  and
  forall e: exp(asgn) holds
      ( not isReturn(e) or name(e)="*" and
       name(member(arg(e),0))="this") )
)
```

- If a pointer data member is assigned inside either a constructor or the assignment operator, then it must be assigned in all the other constructors and assignment operators.

```
forall c: class(global) holds
[ctorAndAsgn as {f: function(c) |
  name(f) = name(c) or name(f)="operator="}]
forall v: var(c) holds
(
  ptdType(type(v))=type(v)
  or
  forall f: ctorAndAsgn holds
    exists e: exp(f) holds
      ( name(e)="=" and
      [leftHand as member(arg(e),0)]
      var(leftHand)=v )
  or
  forall f: ctorAndAsgn holds
    not exists e: exp(f) holds
      ( name(e)="=" and
      [leftHand as member(arg(e),0)]
      var(leftHand)=v )
)
```

- If a pointer data member is assigned inside either a constructor or an assignment operator, then it must be deleted in the destructor.

```
forall c: class(global) holds
[ctorAndAsgn as {f: function(c) |
  name(f) = name(c) or name(f)="operator="};
 expInCtorAndAsgn as union({f: ctorAndAsgn| true| exp(f)});
 asgndVar as {v: var(c)| not ptdType(type(v))=type(v) and
   exists e: expInCtorAndAsgn holds
      ( name(e)="=" and
      [leftHand as member(arg(e),0)]
      var(leftHand)=v )};

 dtor as {f: function(c)| name(f) = "~"+name(c)};
 expInDtor as union({d: dtor| true| exp(d)});
 delete as {e: expInDtor| name(e)="delete"|
        var(member(arg(e),0))};
 arrayDelete as {e: expInDtor| name(e)="delete[]"|
        var(member(arg(e),0))}
]
subset(asgndVar, delete+arrayDelete)
```

- If `new` (`new[]`) is used to initialize a data member in a constructor, then `delete` (`delete[]`) must be used to free the data member in the destructor; no mismatch is allowed.

```
forall c: class(global) holds
[ctors as {f: function(c)| name(f)=name(c)};
 expInCtors as union({c: ctors| true| exp(c)});
 new as {e: expInCtors|
     name(e)="new" and name(parent(e))="=" |
     [p as parent(e); leftHand as member(arg(p), 0)]
     var(leftHand)};
 arrayNew as {e: expInCtors |
     name(e)="new[]" and name(parent(e))="=" |
     [p as parent(e); leftHand as member(arg(p),0)]
     var(leftHand)};

 dtor as {f: function(c)| name(f)="~"+name(c)};
 expInDtor as union({d: dtor| true| exp(d)});
 delete as {e: expInDtor| name(e)="delete"|
     var(member(arg(e),0))};
 arrayDelete as {e: expInDtor| name(e)="delete[]"|
     var(member(arg(e),0))};
]
( sizeof(intersect(new, arrayNew))=0 and
  sizeof(intersect(delete, arrayDelete))=0 and
  subset(new, delete) and
  subset(arrayNew, arrayDelete) )
```

5 Related Work

LoD has been specified and checked elsewhere [10, 16]. Both approaches are limited by available techniques. Thus [10] proposes to extend AspectJ to allow for either *statically executable advice*, which can be executed at compilation time, or *meta advice*, which can be executed directly over abstract syntax trees; the difference is that the latter approach allows for more flexible manipulation of syntax information. FCL follows the latter proposal.

Constraint tools. CCEL [11] and FCL differ in data model, notation, expressiveness, and applied domain. FCL has a richer and simpler data model than CCEL: for example, CCEL does not support expressions and models program elements like variables and parameters separately. FCL borrows its notation explicitly from first-order logic while CCEL adopts a syntax close to C++. Thus FCL should be more portable to other languages than CCEL. FCL is more expressive than CCEL: for example, CCEL has no way to express constraint on the cardinality of sets. FCL aims at frameworks, whereas CCEL so far has been applied mostly to programming guidelines.

Unlike FCL and CCEL, CoffeeStrainer [3] does not have a specification language; instead it requires special annotations in code. Thus CoffeeStrainer is intrusive. CoffeeStrainer depends on features such as dynamic compilation and empty interfaces; thus it is not portable to other languages such as C++. CoffeeStrainer requires application programmers to write the constraints while FCL constraints are specified by framework designers and programmers do not have to read the specifications.

Metal [6] is close to FCL's approach in that it also has a specification language that is based on finite state machine. But Metal focuses on a smaller problem of modeling the states of variables on each inter-procedural execution path. The most interesting insight of this work is that the seemingly simple technique can be effective in finding errors in real systems. Metal also corroborates our design decision of modeling expressions and control paths in FCL.

ASTLOG [4] is built on Prolog; Prolog has limited expressiveness; thus most work of ASTLOG is on extending Prolog to support things such as logical connectives, universal and existential quantifications, and set operations. Implementing these from scratch like FCL is more straightforward without the unnecessary limitations brought about by Prolog. ASTLOG is not typed. Further, ASTLOG has been used mostly to query syntactic patterns related to control statements while FCL focuses more on expressions and execution paths.

Syntactic Query Tools. Many query tools have been built to examine source code for reverse engineering or other purposes: examples include SCA [17], GraphLog [12], Pat [18], and GENOA [5]. These tools differ from FCL in the amount of details contained in program databases, their notations, and the extent to which they can tolerate false positives and false negatives. Typically reverse engineering tasks do not require low-level details such as expressions and can be less sensitive to imprecision. For example, Pat uses Prolog and a commercial CASE tool to detect structural design patterns; for such a task, even a rate of over 50 percent of false positives is considered acceptable. But this is clearly not acceptable for FCL. Pat also reports that less false positives would have been generated if the tool had supported expressions, which provides another supporting evidence for the inclusion of expressions in FCL.

6 Concluding Remarks and Future Work

FCL's formalism lends itself to automation. The developers specify a set of constraints on program structure, and conformance of a specific program to the constraints is then automatically determined by an FCL checker. Application programmers are warned the errors but not required to read FCL specifications.

Technically, FCL is more akin to type-checking. The difference can be illustrated by an example: while the empty program would pass all the type checkers, it will fail on many FCL specifications (especially those existential ones). FCL is similar to Lint [9]. The difference is on expressiveness: to explore the potential of static analysis in detecting errors, we adopt a language approach; thus FCL is more expressive than Lint, which is based on command-line options.

FCL complements testing. Testing requires executability, and execution requires completeness of both the application and the coverage of the test cases. Normally testing is also intrusive. In contrast, FCL does not require a complete application in order to work: it can check partial implementations. Thus FCL fits well with incremental development.

Ongoing work includes improving current FCL implementation, porting it to Java and integrating it into Eclipse, and conducting empirical studies to better understand its applicability.

REFERENCES

[1] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*, chapter 2: Programming Languages. Addison-Wesley, March 1978. Sections 2.9 and 2.10 discuss program units and scopes respectively.

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

[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] R. Crew. ASTLOG: A Language for Examining Abstract Syntax Tree. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 229–242, October 1997.

[5] P. T. Devanbu. GENOA-A Customizable, Front-End-Retargetable Source Code Analysis Framework. *ACM Transactions on Software Engineering and Methodology*, 8(2):177–212, April 1999.

[6] S. Hallem, B. Chelf, Y. Xie, and D. R. Engler. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of OSDI 2002*, Berlin, Germany, June 17-19 2002.

[7] D. Hou. *FCL: Automatically Detecting Structural Errors in Framework-based Development*. PhD thesis, University of Alberta, Edmonton, Alberta, Canada, December 2003. Available as technical report TR04-01 at http://www.cs.ualberta.ca.

[8] R. E. Johnson. Components, frameworks, patterns. In *Proceedings of the 1997 Symposium on Software Reusability*, pages 10–17, Boston, United States, May 17-20 1997.

[9] S. C. Johnson. Lint: a C Program Checker. Unix Documentation.

[10] K. Lieberherr, D. H. Lorenz, and P. Wu. A Case for Statically Executable Advice: Checking the Law of Demeter with AspectJ. In *AOSD'2002*.

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

[12] A. Mendelzon and J. Sametinger. Reverse Engineering by Visualizing and Querying. *Software-Concepts and Tools*, 16:170–182, 1995.

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

[14] S. Meyers and M. Klaus. A First Look at C++ Program Analyzers. *Dr. Dobb's Journal*, February 1997.

[15] J. Newcomer. FlounderCraft Ltd. Available at http://www.flounder.com.

[16] P. P. Pal and N. H. Minsky. Imposing the Law of Demeter and Its Variations. In *TOOLS'96*.

[17] S. Paul and A. Prakash. A Query Algebra for Program Databases. *IEEE Transactions on Software Engineering*, 22(3):202–217, March 1996.

[18] L. Prechelt and C. Kramer. Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns. *Journal of Universal Computer Science*, 4(12):866–882, December 1998. Available at http://www.ipd.uka.de/~prechelt.

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