Supporting the Deployment of Object-Oriented Frameworks

Daqing Hou, H. James Hoover, and Eleni Stroulia

Department of Computing Science, University of Alberta
Edmonton, Alberta Canada T6G 2E8
{daqing,hoover,stroulia}@cs.ualberta.ca

Abstract. Although they are intended to support and encourage reuse, Object-Oriented application frameworks are difficult to use. The architecture and implementation details of frameworks, because of their size and complexity, are rarely fully understood by the developers that use them. Instead, developers must somehow learn just enough about the parts of the framework required for their task. Faced with a framework problem, the developer will ask for assistance or muddle through using a trial-and-error approach. In many cases, they will not learn what the framework designer had in mind as the proper solution to their problem, and thus misuse the framework.
This paper is a preliminary look at the kinds of problems faced by framework users, and how the framework developer can assist in mitigating these problems. Our goal is to develop mechanisms for detecting when the framework user has violated the conditions of use intended by the framework developer, using static analysis of structure, and dynamic analysis of behavior.

Keywords Object-Oriented frameworks, framework deployment, static analysis, model checking

1 Introduction

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

While a deep understanding of general framework based development [4] remains a research problem, there are many frameworks used for production development. Having chosen a framework, how does the development team address the problem of correct usage of the chosen framework?

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

Experience with using industrial strength frameworks has shown that in order for frameworks users to understand and properly use a framework, precise, concise, and complete documentation is needed. However, textual and diagrammatic documents are informal and in general, we do not know yet how to judge whether a programmer has understood a document [7]. Other conventional approaches such as framework design review, manual code inspection, and testing can also be helpful.

Frameworks are supposed to capture commonality in a way that makes reuse easier. But applying most current frameworks requires a nontrivial body of knowledge about the framework on the part of users. Lack of understanding makes debugging difficult because it can be hard to follow a thread of execution that is mostly buried in the framework code. Testing is similarly difficult, since it often requires a fundamental understanding of the architecture of the framework.

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

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

We investigate the feasibility of two technologies, namely, static analysis and model checking, to the problem. Along that line, framework constraints can be categorized into *structural constraints* and *behavioral constraints*. Structural constraints can be evaluated by parsing and analyzing source code while behavioral constraints could be dealt with by model checking.

The rest of section 1 formulates some related framework concepts. After that, section 2 analyzes the framework use problems and causes. The bulk of the paper, from section 3 to 5 describes our proposed solution and experience: Section 3 describes a preliminary version of our specification language FCL for structural constraints. Section 4 gives a specific example for FCL. Section 5 describes our experience with using SPIN/Promela to model check frameworks. Finally, section 6 concludes the paper, summarizing the identified problems and some future work.

1.1 Related Concepts of OO Frameworks

For the clarity of presentation, this subsection defines the following terms: *framework builder, framework user, framework architecture, framework classes, inter-*

nal classes, framework derived classes, framework parts, application classes, hot spots, framework instantiation, framework instance/application, and *framework constraints.*

People who build the framework are referred to as *framework builders* and people who use the framework are called *framework users.* When no confusion occurs, we may also use *builders* and *users* instead. The process of building applications based on frameworks is called *framework instantiation* and the generated application is referred to as *framework instance* or simply *application* when appropriate.

Framework architecture (or *architecture*) defines the typical software architecture for problems in the domain. It comprises both abstract and concrete classes, which can be collectively referred as *framework classes.* Some framework classes are *internal classes* and are not intended to be visible to framework users. Only concrete framework classes can be internal. In most cases, framework users create additional classes by subclassing abstract classes. We call these additional classes *framework derived classes* and the abstract classes *hot spots* [14] or *variation points.* Sometimes, framework builders can predict potential subclasses and provide them as accompanying parts of the framework. For example, in MFC [15] (Microsoft Foundation Classes), class `CCtrlView` has the predefined concrete subclasses `CEditView`, `CListView`, `CRichEditView`, and `CTreeView` [17]. These additional classes capture common implementations of the framework architecture and are referred to as *framework parts.* For those parts of the application design not covered by frameworks, new classes need to be developed to fulfill the actual applications requirements. We refer to these new classes as *application classes.* Collectively, framework derived classes and application classes are sometimes called a *framework extension.*

Thus, in the context of frameworks, an application can be composed of one or more framework architectures, framework parts of each framework (if any), framework derived classes, and application classes.

Framework constraints (see section 3) are rules and conventions for a framework to which any framework extension has to conform. Quite often, these rules and constraints are either implicit in or missing from existing documentation. Making them explicit and checkable should help framework users verify that they have used the framework as intended.

2 Problems Faced by Framework Users

Any tools that purport to support framework use should initially focus on the most common problems. What kinds of problems do framework users face in practice, and is it possible to mitigate them? Although frameworks vary in size and complexity, users face the same kinds of problems. We began by examining two frameworks, Microsoft's MFC and our own CSF (Client/Server Framework). MFC is probably the most successful industrial foundation framework. CSF was developed to as part of our research on frameworks at the University of Alberta and has been used in an undergraduate project course for several terms.

2.1 Common Types of Questions on Frameworks

We did a short investigation of MFC's architecture and implementation, collecting and analyzing questions which users posted to two MFC news groups, `comp.os.mswindows.programmer.tools.mfc` and `microsoft.public.vc.mfc`. We then developed a preliminary classification taxonomy for these questions, consisting of the following five areas.

- "Where" Questions
  Developers often need to introduce application-specific behavior to override or specialize the default behavior of the framework, but do not know where to start. Due to the difficulty of understanding framework code and improper documentation, frequently framework users are unclear or even unaware about the "template methods" they should reuse and the nature of the "hook" methods they should develop [16,8].
- "How to" Questions
  Many questions asked about MFC are of the "how to" type. Examples of this category are "how to translate an Enter key into Tab key so that you can use the Enter key to traverse widgets in a way similar to using the Tab key", or "how to change an Edit's background at run time". To answer this type of questions, the application developer needs to understand the architecture of the underlying framework, specifically, the message and command routing logic [5].
- "Why" Questions
  Quite often, errors in the developers' assumptions about the framework architecture lead to improper extensions that exhibit bizarre behavior. The precise error is difficult to localize, since it is not necessarily in the application code itself and debugging requires a deeper understanding of the framework architecture and possibly its code structure.
- "Does it support" Questions
  Quite often, ambiguities in the framework documentation result in the developer wondering whether or not a particular feature required of the application is possible to implement within the framework. These issues are especially critical, because application development cannot even start without having a resolution for them.
- "What If Not Supported" Questions
  Finally, a large number of questions deal with deficiencies of the framework. Sometimes, a poor choice of framework leads to architectural conflicts or mismatch [9] between the desired application and the framework, in which case little can be done. Often there are simply gaps in the framework where it was never designed to accommodate the users' desired extensions. In this case, it may be possible to evolve the framework to fill these gaps in a manner consistent with the original framework designers intent.

Our study of the use of the CSF framework and the questions asked by the undergraduates students who used it in their course projects revealed, not surprisingly, the same types of questions.

2.2 Underlying Causes

It is difficult to communicate the intended use of a framework to the users who need to develop applications. Sheer size can be one factor: a framework with hundreds of classes, like MFC, is quite daunting to learn even for an expert. Often domain-specific knowledge must first be acquired in order to appreciate how to use the framework [3]. Even small frameworks can have complex behavior that is not obvious from reading static artifacts like code. Here is a brief list of common hurdles in understanding a framework.

Environmental Dependency Frameworks are usually built on top of a certain platform, depend on some supporting techniques, and serve a specific domain. For example, several key design decisions of MFC are based on the Microsoft implementation of the concepts of message queues, event looping and dispatching, window structure, window class, window procedure, and window subclassing. As a result, to understand and use MFC, a programmer must first get familiar with the Windows operating system.

Structural Coupling One problem in using and understanding frameworks is the numerous and varied dependencies between their classes. These structural dependencies comprise an important part of the complexity of frameworks. There are several ways for two classes to work together. Suppose that we have two objects, X and Y, and X needs a reference, *ref*, to Y. Four techniques are commonly used to retrieve the reference:

- X creates and uses Y and then discards it. *ref* is a local variable of some method of X's class; the type of *ref* (i.e. a class) is statically compiled into X's class;
- *ref* is an instance variable of X and references Y. This is a more flexible approach because *ref* can be changed at run-time;
- Y is passed to X as a parameter of some method. This is even more flexible because the responsibility of obtaining a reference no longer lies in X's class;
- Y is retrieved from another object. This object can, for instance, be a factory or a collection.

For example, the Observer pattern [8], used in this paper to illustrate our method, uses techniques 3 and 4. Firstly, the *observer* is registered as being interested in certain event originating from the *subject*. This is done using technique 3: *observer* is passed to the *subject* as a parameter of its `attach` method and the *subject* stores the reference to the *observer* in its collection (usually a list). Later, whenever the event occurs, the *subject* retrieves the previously stored reference from its collection property and calls the *observer*'s `update` method.

Behavioral Coupling O-O systems have complex message dispatch mechanisms that make behavior difficult to comprehend. One example of behavioral coupling is the interaction and dependency between the so-called template [16,8] and hook methods. Template methods define the generic flow of control and interaction between objects; hook methods enable the customization of the behavior of a template method, through interface implementation, or class extension and method overriding [14,10]. Although in theory we have accumulated a body of knowledge on the forms of method interaction, in practice it is not easy for users to recognize the nature of the interaction from the code. A message from the MFC newsgroup amply demonstrates this:

> "This (*when or if you should call the base class version (of functions) from within your overridden version? ... And if you do call the base class version, should you call the base class version before the code in your version or after*) is (*what*) I think one of the most confusing things about MFC... I really think the documentation for all these `CWnd` virtual functions and message handlers should spell this out clearly, case by case. As it is, the only way is to look at the source code for the base class, but since these functions may or may not be implemented at each level of the class hierarchy it is not that easy."

3 The Framework Constraints Language – FCL

In our study of frameworks, we have encountered recurrent types of rules and conventions regulating how to correctly use a framework; we call these rules *structural constraints*. Simple examples of structural constraints can be the cardinality of certain objects, their creation and deletion, and method invocation etc. To formally specify structural constraints we are in the process of developing a language called FCL (Framework Constraints Language). The language is still very preliminary and evolving rapidly as we use it.

This section introduces the main types of constraints that have been identified to date. To help understand the grammar (the appendix to this paper), the corresponding non-terminal symbols are also provided for the description of each construct. A concrete example of the FCL specification for the Observer pattern is given in the next subsection.

An FCL specification starts with a class list section. The section can be used to organize classes into groups such as framework classes, abstract classes, and concrete classes (*ClassListSection*) etc. The class list section is followed by one or more units (*Unit*). Each unit consists of a scope (*Scope*) and a list of constraints. A scope can specify a group of classes either explicitly through class names or through pattern matching. The scope is then followed by the list of constraints (*Constraints*). All classes in the scope must conform to the constraints.

Constraints are further decomposed into primitive and compound ones. Primitive constraints for classes can be either about variables (*OnVar*) or about methods (*OnMethod*). Constraints on the definition of variables (*OnVar*) include their types, cardinality, visibilities, and access scope (i.e., *iVar* for instance variable, *gVar* for global variable, and *lVar* for local variable).

A method constraint has two parts: a method signature and a body (*OnMethod*). The signature identifies the method that will be restricted. The body contains constraints that the method has to satisfy. The constraints on methods (*MethodConstraint*) can be method invocation (*MethodCall*) and sequence (*Sequential*). They can be used to specify causal and temporal relations among methods, i.e., `m1 calls m2` and `m1; m2`, respectively. There are also two predefined predicates on variables, *use* and *change*, which allow one to specify constraints on the usage and modification of variables.

A constraint may depend on other constraints. For instance, "*if you subclass X then you must also subclass Y*". Compound constraints are formed by connecting constraints with the usual logic connectives, implication, negation, conjunction, and disjunction (*Constraint*).

Some example constraints are as follows:

– For a framework class X, one constraint can be "*framework users must/must not subclass X*" (*Subclass*). In order to specify limits on the number of subclasses, the integral range is introduced. It includes greater than, less than or equal, and their combination (*Dim*). Integral ranges can also be used to restrict the number of variables and methods.
– Although the advocated mechanism for using a whitebox framework is inheritance, framework builders may also permit users to augment class X directly. Correspondingly, other constraints on X can be "*must/mustn't add certain instance variables*", "*cardinality and/or type of instance variables Y must/mustn't satisfy certain condition*" (*OnVar*), "*must/mustn't add/call certain methods*" (*OnMethod*). These can also be applied to framework derived classes.
– For a framework derived class X, constraints may be "*class X must/mustn't override a certain method*" (*MethodHead*), "*must/ mustn't add certain instance variables*", "*cardinality and/or type of instance variables Y must/mustn't satisfy certain condition*" (*OnVar*), "*must/mustn't add a certain method*" (*OnMethod*).
– Constraints on a hook method can be: "*must/mustn't delete/change/use parameter variable*" (*Predicate*), "*must/mustn't define variables*" (*OnVar*), "*must call methods before/after some point*" (*MethodCall* and *Sequential*), etc. (*MethodConstraint*)

4 An Example: Structural Constraints for the Observer Pattern

Design patterns can be seen as small frameworks made of a few classes. Although we could have given other examples, for the purpose of presentation, we choose to use the Observer pattern to discuss some of the constraints on the pattern and how to specify them in FCL.

As shown in figure 1, `Subject` and `Observer` are the framework classes of the Observer pattern. A Subject may have one or more Observers associated with it. All observers are notified whenever the Subject changes its state. In response, each Observer will query the Subject to synchronize its state with the Subject.

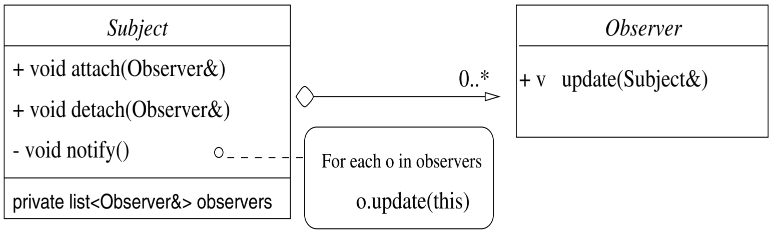

Fig. 1. Class diagram for the observer pattern

```
//Abstract classes
abstractclass = [Subject, Observer]
frameworkclass = [abstractclass, model, view]
```

These two statements define two class lists, one for the abstract classes, `Subject` and `Observer`, the other for `Subject`, `Observer`, and their subclasses, `model` and `view`.

```
subclass model of Subject conforms to
{
        iVar vars (>0)
        private(vars)
        method methods (>1)
        exists m: methods, v: vars
        { change(v); notify()
        }
}
```

This constraint requires that, in the framework extension, there must be exactly one subclass of class `Subject`, which is referred to as `model` (here `model` is only a place holder, the actual subclass can use any name, say `clock`).

The statements in the curly brackets are constraints on the class `model`. The first statement, `iVar vars (>0)`, says that class `model` must define at least one extra instance variable. However, we can predict nothing more about their types and cardinalities. The second statement requires these variables to be private. Similarly class `model` also must define at least two methods (see the third statement) because at least one method is needed to change state and the other to query the state (note that `method` is a keyword of FCL whereas `methods` is only an FCL variable that represents the set of methods of model). The last statement says that there must be at least one method, say `m`, in class `model`, which satisfies the following requirements:

- firstly, `m` changes some variable `v` in set `vars`;
- secondly, `m` must call the inherited `notify` method;
- and thirdly, the change must happen before the `notify` method is called.

```
subclass view (>=1) of Observer conforms to
{
        override update(Subject s)
        {
                s.methods
                !s.m
                !delete(s)
                !change(s)
        }
}
```

This constraint requires that in the framework extension, there must be at least one subclass of class Observer. These subclasses are collectively referred to as `view`. All classes in `view` must override the `update` method. Furthermore, this `update` method is required to call some methods of the actual parameter `s`, but not `m` (recall that `m` is defined in Subject and calls `notify` and `notify` invokes `update` in turn). The method `update` is also not permitted to delete or change `s`.

```
oneclass in !frameworkclass conforms to
{
        model o
        view v (>0)
        o.attach(v)
        o.m
        o.detach(v)
}
```

This statement requires that there must exist exactly one application class, which is referred to as `oneclass`, satisfying the followinf constraint: inside class `oneclass`, there is exactly one object of `model` created (`model o`) and at least one object of class `view` created (`view v(>0)`); there must be the invocations of methods `attach`, `m`, and `detach` of the object `o`.

```
!frameworkclass conforms to
{
        !model::notify()
        !view::update(*)
}
```

This statement says that in all classes except those in `frameworkclass`, there must not be the direct invocations of the methods `model::notify` and `view::update`.

5 Behavioral Constraints

Informally, behavioral constraints are the requirements that are put on applications by the framework but cannot be checked by static program analysis.

Typically such constraints specify a pattern of (two and more) states and/or events that characterizes applications' behavior. Examples of behavioral constraints are as follows:

1. Object interface protocols which are the FSM (finite state machine) specification of the method invocation order of objects. One formal treatment of object protocols appears in [13];
2. Control reachability, e.g., all hook methods must eventually be called, or, in MFC, all user defined message handlers must eventually be called;
3. Livelock-freedom, e.g., no livelocks are permitted in GUI programming.
4. Target existence, e.g., an object must exist when others send message to it.

A variety of temporal logics might be used for encoding behavioral constraints. We experimented with linear temporal logic (LTL) in our work, which is supported by a mature model checker, SPIN [11]. LTL is a propositional logic with the standard connectives `&&`, `||`, `->`, and `!`. It includes three temporal operators: `<>`p says that p holds at some point in the future, `[]`p says that p holds at all points in the future, and the binary p `U` q operator says that p holds at all points up to the first point where q holds. An example LTL specification for the response property "all requests for a resource are followed by granting of the resource" is `[]`(request `->` `<>`granted).

Many efforts have been made to apply model checking to software artifacts including requirements specifications [1,2], architectures [12], and implementations [6]. When model checking software, one describes the software as a finite state-transition system, specifies system properties with a temporal logic formula, and checks, exhaustively, that all sequences of states satisfy the formula. Specifically, in the case of the model checker SPIN, its finite state model is written in the Promela language and its correctness properties in LTL. Users specify a collection of interacting processes whose interleaved execution defines the finite state model of system behavior. SPIN performs an efficient nonempty language intersection test to determine if any state sequences in the model conform to the negation of the property specification. If there are no such sequences then the property holds, otherwise the sequences are presented to the user as exhibits of erroneous system behavior.

In this approach, framework builders are responsible for constructing the validation model of the framework and defining its properties. Together with the framework, the validation model and the properties will be delivered to framework users to model-check the application. Since frameworks are meant for massive reuse, we argue that the framework builders' investment in constructing the validation model and defining the properties will be quickly amortized.

One of the major problems in model checking software is model construction: given real world software components built with sophisticated programming constructs which give rise to gigantic state spaces, generate correct compact representations for tractable model checking. In the next subsection, we describe our experience with using Promela to manually construct finite state models.

5.1 Using Promela to Model The Observer Pattern

We used the following strategy to model the behavior of the observer pattern. Classes are modeled as processes. These processes are embedded in an *atomic* statement to eliminate unnecessary interleaving execution.

Given a class P, to create an object for it, we use `run P(parameter list)`. Each object has a unique id, which is modeled by process id. Therefore, to create an object and get a reference to it, use `byte id; id = run P(parameter list)`. Note that we have defined `id` as type of byte, this is because currently SPIN supports at most 256 processes (that is, 256 objects in our case) simultaneously. Once created, the object keeps observing a rendezvous channel for messages intended for it.

In a sequential program, at any time, there is only one active method. Therefore, we define one single global rendezvous channel to synchronize method invocation among all objects. All objects keep polling the channel to look for messages directed to them. For an object, if there is no message or the message is not for it, it's blocked (this is implemented with the *eval* and *_pid* feature of Promela). The rendezvous channel is defined as follows:

```
chan Rendezvous = [0] of {
object,returnChannel,message,p1,p2, ...,pn}
```

This statement defines the variable `Rendezvous` as a rendezvous channel (as indicated by the dimension of 0) and the message format for the channel. A message consists of following elements:

- `object`: specifies the message reception object;
- `returnChannel`: is used to synchronize caller and callee;
- Method name (`message`): is type of `mtype`, which is the default enumeration type of SPIN; Currently, we assume no method name conflict among all classes and treat all method names as enumeration members of this type; If there are more than 256 messages, we could also model them as integer constants.
- `n`: is the largest number of parameters of all the methods.

In general, methods can also be modeled as processes. However, because of the potential state space explosion problem, we recommend inlining them into class processes whenever possible. Instance variables are modeled as local variables of processes.

Method invocation is synchronized using rendezvous. For each invocation, the caller site provides: object id (`object`), a rendezvous channel (`returnChannel`), method name (`message`), and a list of parameters (`p1, p2, ..., pn`).

The following two code segments illustrate how method calls are modeled. The caller site uses the following pattern:

```
...
//Each caller must define one local rendez-vous
chan returnChannel = [0] of {int};
...
```

```
Rendezvous!object,returnChannel,message,p1,p2, ..., pn
returnChannel? returnValue;
// vp: Variable Parameters
returnChannel? vp1;
...
returnChannel? vpm;
```

The callee site uses the following pattern:

```
do
::Rendezvous?this,returnChannel,msg,p1,...,pn
        if
        ...
        ::msg == message -> do something;
               returnChannel!returnValue;
               returnChannel!vp1;
               ...
               returnChannel!vpm;
        ...
        :: msg == delete -> break;
               returnChannel!NULL
        fi
od
```

In the code, `this` represents the callee's pid. `returnChannel`, `msg`, and `p1` to `pn` are local variables of the callee process.

The purpose of `returnChannel` is to prevent the caller from proceeding unless the callee returns. When the callee finishes, it first sends the caller its return value, then the value of each variable parameter, if any.

Inheritance is also modeled. Each object of a concrete subclass automatically creates one internal object for each of its superclasses. All messages that are not recognized by one object are dispatched to its superclass objects.

For the Observer pattern, a small model was manually constructed. To test it, several artificial bugs were seeded into the model.

– The first bug is that the user code calls `update` directly;
– The second is sending a message to an already-deleted observer object;
– The third is terminating the program without freeing an object.

The frameworks constraints specifying the permitted patterns on method calls should ideally be expressed as trace clauses over the traces of the system. However, as discussed below, SPIN is limited in its capability for specifying event patterns. Instead, the constraints were specified using assertions.

The result is somewhat dissatisfying: model checking quickly found all the bugs, but it is not clear how well this technique generalizes and extends.

LTL supports only the specification of properties of state sequence instead of action/event sequence. But sometimes we need to write properties directly in terms of events, such as method invocation and return. On the surface, it seems that SPIN/Promela's trace assertion can be used for this purpose. But our initial experiment revealed several problems:

– SPIN can only trace events on a global channel;
– It does not support non-determinism;
– When specifying properties for sequential programs, constructs that can distinguish between causal relation and temporal relation between events are needed.

Specially, SPIN seems not to be designed to support the last point above. Actually, during our exploration of SPIN, we have found one bug in SPIN's implementation of trace assertion. This shows that probably the trace mechanism has not been extensively used [Gerard Holzmann, Personal Communication] in practice. Although we could work around by defining extra boolean variables to mark the invocation and return of method, thus express properties on events by state based LTL formula, this method is neither efficient nor straightforward.

Due to space limitation, readers are referred to [18] for further details.

6 Conclusions and Further Work

Many structural constraints are feasible to check. Checking these could achieve benefits similar to what compilers have done with static type checking.

A checker program is envisaged that understands the FCL specification, parses the application, and performs the conformance checking. In addition, our FCL definition is still in the demonstration stage, thus needs revision to add new types of constraints. One of our ongoing tasks is to formalize the core architecture of MFC and identify checkable structural constraints.

Checking behavioral constraints is more problematic. Since we use processes to model objects, one problem is how to model the visibility of objects. In addition, languages such as C++ and Java permit direct reference to the public variables of objects, thus the capability of modeling visibility would also be needed. Earlier versions of SPIN did support remote referencing, which could be used to achieve this. Unfortunately, to support partial order reduction of the state space, this feature has been removed from the newer versions. Although SPIN is not suited, other formalisms such as CTL (Computation Tree Logic) are worth investigating.

By the very nature of frameworks, the validation model must be incomplete. Therefore, there are two other important problems left:

– how to extract the other part of the validation model from application classes, and
– how to combine the two models to get a complete validation model of the application so that we can run model checking on it.

Further investigation to these problems is also needed.

Finally, section 2 only briefly summarizes the types of questions that framework users may have: they are by no means either the best classification or complete. Further empirical study is needed to provide more fine–grained knowledge.

Acknowledgements

The authors thank Marsha Chechik, Jim Cordy, Garry Froehlich, Amr Kamel, Rudolf Keller, Andrew Malton, and Keny Wong for their discussion and assistance during various stages of the project. We also want to thank the three anonymous reviewers for their very insightful comments on the paper.

This work was supported by the Natural Sciences and Engineering Research Council of Canada, and the Alberta Science Research Authority.

References

1. R. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. Reese. Model Checking Large Software Specifications, Software Engineering Notes, 21(6):156–166, November 1996. 160
2. J. Atlee, J. Gannon. State-based Model Checking of Event-driven System Requirements, IEEE Transactions on Software Engineering, 19(1):24–40, June 1993. 160
3. A. Birrer, T. Eggenschwiler. Frameworks in the Financial Engineering Domain: An Experience Report, Proceedings of ECOOP 93, 1993. 155
4. J. Bosch, P. Molin, M. Mattsson and P. Bengtsson. Obstacles in Object-Oriented Framework-based Software Development, ACM Computing Surveys Symposia on Object-Oriented Application Frameworks, 1998. 151
5. P. DiLascia. Meandering Through the Maze of MFC Message and Command Routing, Microsoft System Journal, July 1995. Available at <http://www.microsoft.com/msj/0795/dilascia/dilascia.htm> 154
6. M. Dwyer, V. Carr, and L. Hines. Model Checking Graphical User Interfaces Using Abstractions, In LNCS 1301, pages 244–261. Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 1997. 160
7. G. Froehlich, H. J. Hoover, L. Liu, P. G. Sorenson. Hooking into Object-Oriented Application Frameworks, Proceedings of the 1997 International Conference on Software Engineering, Boston, Mass., May 17-23, 1997. 152
8. E. Gamma, R. Helm, R. E. Johnson, J. O. Vlissides. Design Patterns–Elements of Reusable Object-Oriented Software, Addison Wesley, 1994. 154, 155
9. D. Garlan, R. Allen, J. Ockerbloom. Architectural Mismatch or Why it is so hard to build systems out of existing parts, Proceedings of the 17th International Conference on Software Engineering, April 1995. 154
10. R. Helm, I. M. Holland, D. Gangopadhyay. Contracts: Specifying behavioral Compositions in Object-Oriented Systems, Proceedings of ECOOP/OOPSLA 90, Ottawa, Canada, 1990. 156
11. G. Holzmann. Design and Validation of Computer Protocols, Prentice Hall, Englewood Cliffs, NJ, 1991. 160
12. G. Naumovich, G. Avrunin, L. Clarke, and L. Osterweil. Applying Static Analysis to Software Architectures. In LNCS 1301. The 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 1997. 160
13. O. Nierstrasz. Regular Types for Active Objects, Object-Oriented Software Composition, O.Nierstrasz and D.Tsichritzis eds, Prentice Hall, 1995, pp.99–121. 160
14. W. Pree. Design Patterns for Object-Oriented Software Development, Addison Wesley, 1995. 153, 156

15. G. Shepherd, S. Wingo. MFC Internals: Inside the Microsoft Foundation Class Architecture, Addison Wesley, 1996. 153
16. R. J. Wirfs-Brock, B. Wilkerson, and L. Wiener. Designing Object-Oriented Software, Prentice Hall, Englewood Cliffs, NJ, 1990. 154, 155
17. Microsoft Developer Network. Available at <http://www.msdn.microsoft.com> 153
18. Promela Model for the Observer Pattern. Available at <http://www.cs.ualberta.ca/˜daqing/frameworks/so> 163
19. The Client Server Framework Web Site. Available at <http://www.cs.ualberta.ca/˜garry/framework>

A Appendix: FCL Grammar

```
Legend note: all symbols starting with capital letters are nontermi-
nal. ::=, [], {}+, | and empty are preserved as meta-symbols. All
others are terminal symbols.
LType and LMethodProt are programming language specific definition
of type and method signature, which are omitted.

FrameworkConstraint ::= ClassListSection Toplevel

     ClassListSection ::= ClassList | ClassList ClassListSection
         ClassList::= ListName = "[" List "]"
         List     ::= Class | Class , List
         Class    ::= Id | ListName
         ListName ::= Id

     Toplevel ::= Unit | Unit Toplevel
         Unit ::= Scope [conform|conforms] to { Constraints }
                  | Unit->Unit
     Scope ::= ListName | SubClass | !Scope
               | NameDecl in Scope
         SubClass ::= subClass NameDecl of Class
         NameDecl ::= Id [(Dim)]
         Dim      ::= BasicDim | BasicDim, BasicDim
         BasicDim ::= Constant | > Constant | <= Constant

     Constraints ::= Constraint | Constraint Constraints

     Constraint  ::= OnVar | OnMethod |
         Constraint -> Constraint | !Constraint |
         {Constraints} | Constraint "||" Constraint
         OnVar ::= Type NameDecl | Predicate ( Ids )
               Type ::= LType | iVar [LType] | lVar [LType]
                        | gVar [LType]
         OnMethod   ::= MethodHead { MethodConstraints }
                        | MethodConstraint
            MethodHead ::= [override] LMethodProt | Qualified
                           | method NameDecl
               Qualified  ::= [exists] {Id: Id [,]}+
                            | forall {Id: Id[,]}+
            MethodConstraints ::= MethodConstraint
                            | MethodConstraint MethodConstraints
            MethodConstraint  ::= OnVar | Predicate ( Ids )
                            | MethodCall [(Ids)] | Sequential
                            | MethodConstraint->MethodConstraint
                            | !MethodConstraint
                            | { MethodConstraints }
                            | MethodConstraint "||" MethodConstraint
               MethodCall ::= [Class::] Id | Id.Id
               Predicate  ::= use | change | delete
                            | public | protected | private
               Sequential ::= MethodConstraint ; Sequential
                            | MethodConstraint ; MethodConstraint

/*
 *Simple non terminal symbols
 */
Constant ::= integer constant
Id       ::= Identifier
Ids      ::= empty | Id  | Id, Ids | *
```