

# Verifying Framework-Based Applications by Establishing Conformance

Peter Molin  
University College of Karlskrona/Ronneby  
Department of Computer Science and Business Administration  
Soft Center  
S-372 25 Ronneby  
Sweden  
Peter.Molin@ide.hk-r.se

## Abstract

The use of object-oriented frameworks is one way to increase productivity by reusing both design and code. In this paper, a framework-based application is viewed as composed by a framework part and an *increment*. It is difficult to relate the intended behaviour of the final application to specific increment requirements, it is therefore difficult to test the increment using traditional testing methods.

Instead, the notion of increment *conformance* is proposed, meaning that the increment is designed conformant to the intentions of the framework designers. This intention is specified as a set of composability constraints defined as an essential part of the framework documentation. Increment conformance is established by verifying the composability constraints by means of code and design inspection. Conformance of the increment is a necessary but not sufficient condition for correct behaviour of the final application.

## 1 Introduction

The object-oriented framework is a concept that aims at reducing software development costs by systematic reuse of both design and code. Applications can be built by instantiating one or more frameworks. Throughout this paper, the notion of *increment* of an application is used to describe the design and code provided by the application developer. The *framework part* denotes the parts provided by the frameworks, see figure 1.

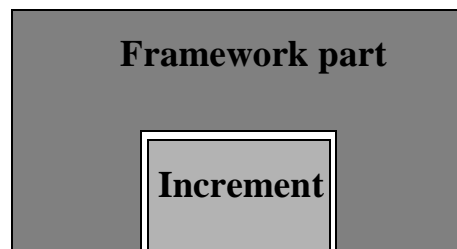
For large and complex applications built by combining and adapting several frameworks, it is advisable to perform local verification of the increment, and not rely solely on final application testing. Such local certifiability is essential for managing large software systems [1]. Traditionally, the increment would be treated as a component and consequently verified by testing it against its behavioural specification. However, there is a difficulty in specifying the behaviour of the increment, since it has no behaviour of its own; instead it contains several adaptations and modifications of the default behaviour defined by the framework or frameworks. It is thus difficult to test the adaptations using traditional testing methods.

The notion of *conformance* is introduced in this paper, which in informal terms means that the increment is designed in a way that conforms to the intentions of the framework designers. Instead of testing or verifying the behaviour of the increment, only its conformance is verified. For this purpose, it is necessary that all the requirements, design rules and constraints defined by the framework, are identified and documented as a set of *composability constraints*. When a framework-based application is constructed, the increment must be verified against these

composability constraints. Verification of typical composability constraints (see subsection 3.1) requires often the use of code or design inspection.

One benefit of this method is that the constraints are available when the design of the increment starts, thereby making it possible to design the increment in a conformant way from the beginning. Another benefit is that the method promotes a “black-box” reuse approach to frameworks where less knowledge and understanding of framework internals are required. The “black-box” approach is necessary for large scale applications where several frameworks may be used.

This paper starts with some definitions, followed by a problem statement. In the subsequent section the conformance concept is discussed further, together with a categorisation of composability constraints. A small framework example is given in the fourth section.



*Fig. 1 Framework-based application*

## 1.1 Definitions

Frameworks have been discussed in more detail by several other researchers [2], [3], [4], [5] and [6]. This paper uses the following definitions based on a simple model of generating an application based on one or more frameworks. The application engineer designs *application classes*, which are either new increment classes, or subclasses from the frameworks or classes composed by framework or application classes. The *increment* is defined as the application classes, see figure 2.

At run time, objects are instantiated either directly based on framework classes or based on application classes. When investigating all the objects of the application, three different kinds of objects can be found: pure increment, pure framework, and framework-based objects. All these objects interact in certain ways. The code executed by the latter object category, framework-based objects, contains code both produced by the framework designer and the application engineer.

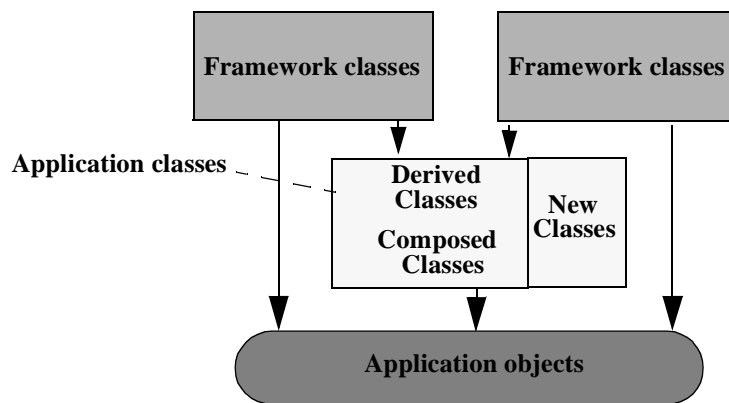


Fig. 2 Different objects in a framework-based application.

## 2 Problem statement

Before the problems can be stated, the importance of local certifiability must be stressed. A requirement for scalability is, according to Weide and Hollingsworth [1], that each component of a software system can be certified locally. The problems addressed in this paper are:

- **The problem of specifying the behaviour of the increment.** Even if it is possible to specify the behaviour of an actual application, and a well-documented description of the framework is available, it is difficult to deduce a specification of the increment.
- **The problem of verifying the behaviour of the increment.** It is difficult to test the increment, since it is difficult to express its specification. Furthermore, testing probably requires considerable effort to design test drivers which simulate parts of the framework behaviour. Improving cost efficiency by reuse presupposes that the verification cost of the framework-based application is small and in proportion with the effort of producing the application.
- **The problem of combining local certifiability with framework-based software development.** How can the promising technique of object-oriented frameworks be combined with the sound principle of local certifiability in a cost-effective way?

## 3 Conformance

A partial solution to these problems is suggested by the introduction of the concept of *conformance*. Increment conformance is defined as the fulfilment of all *composability constraints*. A composability constraint is a necessary requirement on the increment imposed by the used frameworks. Conformance means that the increment is designed and implemented according to the intentions of the framework developers. However, it does not imply that the application will behave as intended, though non-conformance certainly indicates that the resulting application will not behave as intended. Thus, conformance is a necessary, but not sufficient, condition for correct behaviour.

The proposed concept affects the development process of the framework, in which it is necessary to identify and document *all* composability constraints required for an increment that can be integrated with the framework.

Until a formal composability constraint language is defined, the constraints must be verified by

code or design inspection.

### 3.1 Constraint categories

This subsection contains a description of constraint categories. All of the categories are well-known, and a reference to typical work in this area is provided. The most important work related to this subject is the *Contracts* concept defined by Helm et al. [7], where a formal invariant applicable to a set of framework classes is defined. The contract must be fulfilled by all subclasses of these classes. Such contracts cover constraints on interrelation between framework and framework based classes. The *Framework Description Language* suggested by Wilson and Wilson [8], includes another category of constraints, namely, the specification of classes that must be instantiated and classes that must be subclassed.

Problems with inheritance have been covered by a large number of researchers. A formal approach with *class invariants* has been suggested by Meyer [9] and McGregor and Dyer [10]. Aksit and Bergmans discuss the special problems of inheritance and state in [11]. Synchronisation and inheritance is the subject of work carried out by Matsuoka et al. [12]. These results can be seen as a way of identifying constraints on subclassing which are examples of composability constraints.

In addition to these more object-oriented constraints, there exists a large number of additional constraints, for example, resource sharing constraints, or atomicity constraints in concurrent systems. Some of these constraints are summarised in the paper concerning context-dependent constraints by Molin [13].

There are additional constraints regarding the structure of the final application. The following list of constraints are based on the usage rules defined by Mattsson [6]:

- Framework-based objects, where specific methods must be called at a predefined frequency.
- Constraints on the cardinality and the existence of certain framework-based objects.
- Constraints on creation of dynamic/static framework-based objects.
- Constraints on instantiation order.
- Constraints on recursive calls and re-entrance.

## 4 Example

A small example is presented in this section, with only a few composability constraints. The example is a small simulation framework defined by Budd [14]. The framework consists of two visible classes, *Simulation* and *Event*. The following code example in C++ represents parts of the class declarations:

```
class event {
public:
    //...
    virtual void processEvent() = 0;
}

class simulation {
public:
    //...
    void run();
    void scheduleEvent ( event &);
    int done;
    //..
};
```

From the descriptive text in the book, and from the given code examples, the following list of composability constraints can be deduced:

- Class *simulation* must be instantiated or subclassed and instantiated.
- Class *event* must be subclassed and instantiated.
- Subclassed *events* must be created dynamically.
- Created *events* and *simulation* classes must not be explicitly deleted by the increment.
- Pointers and references to such objects are not valid after *run* is called.
- Pointers and references to *event* objects are not valid if the variable *done* is set to true.
- *processEvent* may not be called directly from the application part.

In order to show conformance of the application classes, it is necessary to verify these constraints.

## 5 Conclusion and future work

Conformance verification is proposed as a means of local verification of an increment when building a framework-based application. The verification is performed by confirming that the increment fulfils a number of well-defined composability constraints. Code and design inspection is proposed as the preferred verification method. Conformance verification requires the identification and documentation of all composability constraints as an important part of framework development and documentation.

Conformance can be seen as a possible solution to the composability problem, where conformance indicates that a component fulfils all requirements enforced by the remainder of the system. Correct behaviour and conformance are thus two requirements that can be defined regarding a component. Behaviour can be tested locally, conformance cannot, and must therefore rely on source code tools or manual inspection.

One obvious research direction is a formalisation of the composability constraints. Another direction is to focus on methods that identify all the constraints defined by a framework. Finally, a research suggestion is to automate the conformance verification process.

## Acknowledgements

I would like to thank my colleagues, Jan Bosch and Michael Mattsson, for fruitful discussions on this subject.

## References

- [1] Weide, B., Hollingsworth, J., "Scalability of Reuse Technology to Large Systems Requires Local Certifiability", *Proceedings of the 5th Annual Workshop on Software Reuse*, October 1992.
- [2] De Champeux, D., Lea, D., Faure, P., *Object-Oriented System Development*, Addison-Wesley, 1993
- [3] Cotter, S., Potel, M., *Inside Taligent Technology*, Addison-Wesley, 1995.
- [4] Johnson, R., Foote, B., "Designing Reusable Classes", *Journal of Object-Oriented Programming*, Vol. 1, No 2, pp. 22-35, June 1988.
- [5] Lajoie, R., Keller, R. K., "Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts and Motifs in Concert", *Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l'Avancement des Sciences*, pp. 94-105, Montreal, Canada, May 1994.

- [6] Mattsson, M., *Object-Oriented Frameworks - A survey of methodological issues*, Licentiate thesis, Department of Computer Science, Lunds University, 1996.
- [7] Helm, R., Holland, I. M., Gangopadhyay, D., "Contracts: Specifying Behavioural Compositions in Object-Oriented Systems", *Proceedings of ECOOP/OOPSLA '90*, pp. 169-180, 1990.
- [8] Wilson, D. A., Wilson, S. D. "Writing Frameworks -Capturing Your Expertise About a Problem Domain", *Tutorial Notes, OOPSLA' 93*, Washington, 1993.
- [9] Meyer, B., "Applying 'Design by Contract'", *IEEE Computer*, Vol. 25, No. 10, pp. 40-51, October 1992.
- [10] McGregor, J. D., Dyer, D. M., "A Note on Inheritance and State Machines", *Software Engineering Notes*, Vol. 18, No 4, pp. 61-69, October 1993.
- [11] Aksit, M., Bergmans, L., "Obstacles in Object-Oriented Software Development", *Proceedings of OOPSLA '92 in ACM SIGPLAN Notices*, Vol. 27, pp. 341-358, New York, October 1992.
- [12] Matsuoka, S., Wakita, K., Yonezawa, A., "Synchronisation Constraints With Inheritance: What Is Not Possible - So What Is?", *Internal Report*, Tokyo University, 1992.
- [13] Molin, P., "Designing Reliable Systems from Reliable Components using the Context-Dependent Constraint Concept", *Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE'96)*, pp. 142-151, White Plains NY, USA, November 1996.
- [14] Budd, T. A., *Classic Data Structures in C++*, Addison-Wesley, 1993.