Safe Object-Oriented Software: The Verified Design-By-Contract Paradigm

David Crocker

Escher Technologies Ltd.
Aldershot, United Kingdom
dcrocker@eschertech.com

Abstract. In recent years, large sectors of the software development industry have moved from the procedural style of software development to an object-oriented style. Safety-critical software developers have largely resisted this trend because of concerns about verifiability of object-oriented systems. This paper outlines the benefits offered by object technology and considers the key features of the object-oriented approach from a user's perspective. We review the main issues affecting safety and propose a paradigm – Verified Design-by-Contract – that uses formal methods to facilitate the safe use of inheritance, polymorphism, dynamic binding and other features of the object-oriented approach. An outline of *Perfect Developer* – a tool supporting the Verified Design-by-Contract paradigm – is included.

1 Introduction

In recent years there has been a substantial move from procedural to object-oriented approaches in many sectors of the software development industry. The principal advantage of the object-oriented approach is the ease with which reusable components and application frameworks can be created.

Although the benefits of object technology have been oversold in some quarters, most studies indicate that software development companies moving to object technology have at worst maintained their previous productivity (Potok et al 1999) and at best increased it by several times (Port & McArthur 1999, Mamrak & Sinha 1999). The greatest productivity gains come from re-using components or frameworks from one project to another. This suggests that even if a company switching to object technology sees little saving in the short term, it will gain in the longer term as the opportunity for re-use arises. Our own experience is that in some application areas at least, an object-oriented design is significantly simpler and faster to implement than a procedural design, regardless of re-use; provided that

the development staff are already experienced in object technology. We have also found object-oriented designs easier to extend to meet new requirements.

Despite the potential benefits, safety-critical software developers have largely avoided object-oriented methods, preferring instead to use procedural or modular approaches. However, a number of companies in safety-critical sectors are now moving to object-oriented software development, or planning such a move. This is particularly evident in the North American aerospace community. In recognition of this trend, a number of interested parties including the Federal Aviation Administration and NASA have set up the Object Oriented Technology in Aviation (OOTiA) programme to address safety and certification issues when object-oriented software is used in airborne applications.

This paper considers the reasons behind the slow uptake of object technology by the safety-critical software development community. We describe how the coupling of an existing design technique with formal verification allows the most powerful features of object technology to be safely used, even in critical applications. We have developed a toolset that employs modern automated reasoning technology to obtain a very high degree of automated proof, in order to make formal verification economic even for less critical software. The use of formal specifications also makes automatic code generation possible, eliminating coding errors and providing greater overall productivity than a non-formal approach in many cases. All of this makes it easier for developers to create safe software.

2 Features of Object-Oriented Technology

There is general agreement that the essential attributes of the object-oriented approach to software development include the following:

- Encapsulation: the process of encapsulating data and the operations pertaining to that data in a single entity such that the data cannot be publicly manipulated other than via the published operations. The template describing such an entity is called a *class* and plays the role of a type in procedural languages. Instances of a class are called *objects*.
- Abstraction: the process of hiding the unimportant details of an object from its users so that only the essential features that characterize it remain. The process of abstraction is greatly helped by encapsulation, since the details of how the data is represented inside an object can be hidden from its users.
- Inheritance: the principle of defining new classes by inheriting existing classes, adding new data and/or operations and possibly redefining existing operations. Some languages support single inheritance, while others support multiple inheritance (i.e. a class declaration may inherit more than one other class).

- Polymorphism: the principle that where some variable or similar entity is declared as being an instance of some class, then at run-time it may be permissible to substitute an instance of a different class derived from the original by inheritance.
- Dynamic binding (also known as *dynamic dispatch*): the principle that when a variable or similar entity is polymorphic and is passed as a parameter in a call to a function or procedure, the exact function or procedure called may not be statically determined but may depend at run-time on the class of which the variable is an instance. Most object-oriented languages support *single dynamic dispatch* (i.e. the choice of function or procedure called depends on at most one parameter, which is typically distinguished syntactically from the other parameters); a few support *multiple dynamic dispatch*.

Abstraction and encapsulation are highly beneficial features to have in a programming or modelling language and are likely to enhance safety. Indeed, the widely used *modular programming* approach captures both these features. Abstraction and encapsulation also facilitate formal analysis, since when analysing code that uses a class, the abstract specification of the class is sufficient to capture its significant behaviour, so that the detailed implementation of the class can be disregarded. Separately, the class can be analysed to ensure that its detail conforms to its abstract specification.

Inheritance does not in itself cause any particular difficulty for program analysis, since a class derived by inheritance could be expanded by substituting the member declarations of the inherited class(es) into the definition of the derived class. However, the combination of inheritance, polymorphism and dynamic binding is not directly amenable to traditional static analysis, which typically requires that the target of each procedure call is statically known.

One solution for engineers of safety-critical software who wish to adopt object technology is to eschew dynamic binding. This may be a reasonable way of getting started with object technology. However, dynamic binding has been found to be such a powerful and useful feature that this is surely not the best long-term approach. Better instead to seek new verification techniques that can ensure dependability even in the presence of dynamic binding.

3 An Object-Oriented Example

We have heard it claimed that object technology (and dynamic binding in particular) is not useful in most safety-critical software. While there may be some systems for which object technology has little to offer, there are many others that clearly could benefit from object technology provided that safety concerns can be addressed.

As a working example for the purposes of this paper, consider a glass-cockpit flight instrument display with the following requirements:

- A number of flight instruments (e.g. airspeed indicator, altimeter, horizontal situation indicator) are to be displayed on a single screen.
- The details of which instrument is displayed in which position should not be fixed in the software but should be easy to change. This will allow a family of systems to use the same software and also provide for a limited amount of installation-time or in-flight customisation (e.g. a choice of presentation, or a choice between two different instruments that convey the same information).
- There may be other information (textual information, alarms etc.) to be displayed on the screen.

This design outline was inspired by an example given in the OOTiA draft handbook (OOTiA 2003). To the object-oriented design engineer, there is a natural inheritance hierarchy in this description. At the root is an abstract class representing any self-contained displayed entity, which we will name *DisplayedElement*. Inheriting from this we might have a concrete class *TextElement* and another class *FlightInstrument*. Concrete classes such as *AirspeedIndicator* will be derived from *FlightInstrument*.

The complete glass-cockpit display can be represented by another class *Display* whose data comprises a collection of objects derived from *DisplayedElement*, each associated with the corresponding screen coordinates. To initialise the screen, we provide a method *drawAll* that iterates through the collection, drawing each object. This suggests a dynamically bound call to a *draw* method[1] that is separately defined for each concrete class derived from *DisplayedElement*. Note that the *draw* method in class *DisplayedElement* is declared but not defined, making it an ***abstract*** method (sometimes referred to as a ***pure virtual*** method). Likewise, the class DisplayedElement is an abstract class (meaning that it cannot be instantiated but serves only as a base for other classes to inherit). Classes derived from *DisplayedElement* will provide their own definitions of the *draw* method.

The architecture described above provides the flexibility we need in that any sort of *DisplayedElement* can occur at any position in the collection. Furthermore, if we wish to add a completely new type of flight instrument to this design, we need only define a corresponding class derived from *FlightInstrument* and provide a means to store an instance of this class in the collection.

Without dynamic binding, we would have to use a "switch" statement or similar in place of each call to *draw* so as to select the correct procedure according to the actual type of the element we wish to display. The same goes for any other operation that depends on the element type. If we add a new type of flight instrument, we need to update every one of these switch statements to handle the new type, thereby spreading the modification throughout the code instead of concentrating it in one place. Thus, the object-oriented approach makes it easier to

[1] Functions and procedures are referred to as *methods* in object-oriented software development

extend the system in a safe manner that leaves almost all of the existing system unaltered.

This example provides several opportunities for re-use. The application framework (comprising the class representing the collection of instruments and associated scheduling of *draw* operations) can be re-used for different displays with widely differing selections of flight instruments. The flight instrument classes could likewise be re-used with a different framework. Several flight instruments may share some common presentation details (e.g. style of frame and caption), so these features may be implemented in a common parent class (e.g. *SquareFlightInstrument*).

There is a standard language – the Unified Modeling Language (UML) – for describing graphically the relationships between classes (and many other aspects of object-oriented systems). A UML class diagram of the system described above is shown in Fig. 1.

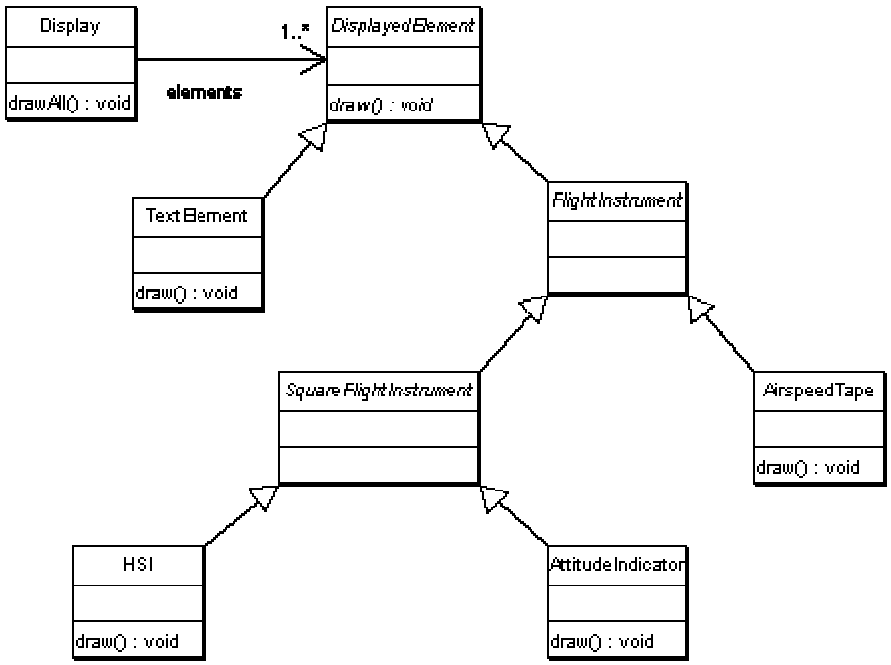

Figure 1: Class diagram for flight instrument display system

4 The Design-By-Contract Paradigm

4.1 Basic principles of Design-by-Contract

The term "Design By Contract" (DBC) appears to have been conceived by Bertrand Meyer of Interactive Software Engineering (Meyer 1988) and the term is claimed by that company as a trademark. However, the principles of DBC go back to Floyd-Hoare Logic (Hoare 1969), the essence of which can be summarised as follows.

A program statement ***S*** exists to achieve some desired *postcondition* ***R*** after its execution (where ***R*** is a predicate over the program state; in other words, ***R*** is a mathematical description of the state that the programmer intends after ***S*** is executed). Typically, the statement ***S*** will only accomplish the state ***R*** if some *precondition* ***P*** is satisfied before ***S*** is executed (***P*** is another predicate over the program state). Given ***P***, ***R*** and ***S***, then in order to be certain that ***R*** will be satisfied after executing ***S***, we need to be sure of two things:

1. Provided ***P*** is initially satisfied, executing the program fragment ***S*** will terminate in a state satisfying ***R***; and
2. ***P*** is always satisfied immediately before ***S*** is executed.

In Design By Contract, this principle is applied to the case where ***S*** is a call to a method. For each method, its precondition ***P*** and its postcondition ***R*** are documented. The correctness requirements are expressed in the form of a contract between the method and its callers, like this:

- The method promises that provided it is called in a state satisfying its precondition ***P***, then it will return in a state satisfying its postcondition ***R***.
- All callers of the method promise to satisfy ***P*** at the point of call; in return, they are entitled to assume that the call will complete and that ***R*** is satisfied on return.

Returning to our example, let's look at how DBC may be applied to the *draw* method of class *DisplayedElement* and its descendants. We assume that the parameters to *draw* include the coordinates of a rectangular portion of the screen in which the element is to be displayed. The contract might look like this:

- The caller of *draw* promises that the size of the rectangle passed to *draw* is at least the minimum needed to display the element;
- The implementation of *draw* promises that on return, the element is displayed in that rectangle and the remainder of the screen is unchanged.

It is frequently the case that the instance variables of a class should satisfy some property at all times. In our example, we might wish that in the *Display* class, the rectangles associated with the elements to be included on the display never overlap and are always wholly contained within the visible area. Such a property would logically be part of the postcondition of every constructor[2] for the class and part of both the precondition and the postcondition of every method of the class. Rather than explicitly state the property in all these preconditions and postconditions, it is simpler and clearer to state it as a *class invariant* instead.

4.2 Design-by-Contract with Dynamic Binding

The Design-by Contract paradigm can also be applied to dynamically bound method calls. We will confine ourselves to the case of single dynamic dispatch.

The general situation is as follows. The program code contains a call to a ***nominal target*** (e.g. method *draw* of the abstract class *DisplayedElement*); but this is interpreted at run-time as a call to some ***actual target*** that depends on the run-time type of the object concerned (e.g. if the object concerned has run-time type *AttitudeIndicator*, the actual target will be the version of *draw* defined in that class). Recall that the correctness of a program segment involving a method call depends on the following conditions:

1. The caller satisfies the precondition of the called method.
2. The method guarantees to satisfy its declared postcondition, provided that its precondition was satisfied.
3. On return, the caller may assume that the postcondition of the method holds.

If the method is dynamically bound, we have a potential difficulty with conditions 1 and 3 because the actual target method is not statically known (and hence we cannot determine its precondition and postcondition). Indeed, one of the features of object-oriented development is that we can add new classes (such as new classes derived from *FlightInstrument*) and that the old client code (for example the *drawAll* method of class *Display*) will work with them unaltered.

The solution is for the caller to refer to the contract of the nominal target instead of the (unknown) actual target. Our correctness conditions become:

1. The caller satisfies the precondition of the nominal target.
2. The actual target guarantees to satisfy its declared postcondition, provided that its precondition was satisfied.
3. On return, the caller may assume that the postcondition of the nominal target holds.

---

[2] A procedure whose purpose is to create and initialise objects of a class is called a *constructor*

To these we need to add:

4. Satisfaction of the precondition of the nominal target is sufficient to ensure satisfaction of the precondition of the actual target.
5. Satisfaction of the postcondition of the actual target is sufficient to ensure satisfaction of the postcondition of the nominal target.

We have added conditions 4 and 5 to link the contract that is actually satisfied with the contract that the caller assumes. We now consider how these conditions may be guaranteed.

The simple case is for each actual target to have the same contract as the corresponding nominal target. For example, suppose the definition of *draw* in class *AttitudeIndicator* inherits the contract given at the declaration of *draw* in its ancestor *DisplayedElement*. Any call whose nominal target is *draw* in class *DisplayedElement*, and which is correct with respect to the contract declared for *draw* in that class, will be correct if the actual target is *draw* in *AttitudeIndicator*.

However, it is also permissible to define a new contract for *draw* in class *AttitudeIndicator*, provided that the new contract conforms to the original. The conformance required is that the new contract may assume no more than the old, and it must promise no less. In other words:

- The overriding method (e.g. *draw* in *AttitudeIndicator*) may have a weaker precondition than the overridden one (*draw* in *DisplayedElement*), but not a stronger one (i.e. the original precondition implies the new one);
- The overriding method may have a stronger postcondition than the overridden one, but not a weaker one (i.e. the new postcondition implies the original one).

In summary, an overriding method may weaken the precondition and/or strengthen the postcondition of the overridden method.

5 Informal and Semi-Formal Use of Design-By-Contract

Design-By-Contract can be implemented in various ways, ranging from informal methods that rely on the developer to ensure correctness, to rigorous ways that are amenable to automated analysis.

5.1 Implementing DBC with Comments

The contract of a method can be documented in the form of comments. This approach may be used with any programming language. For example, a C++ declaration of the *draw* method might appear like this:

```
// File DisplayedElement.hpp
#include "Screen.hpp"

class DisplayedElement {
public:
  // Display the element in the given rectangle
  virtual void draw(Screen &s, Rectangle r)
  //pre r.height >= minHeight,
  //    r.width >= minWidth;
  //post instrument is displayed in the rectangle,
  //     rest of the display is unchanged;
    = 0;
}
```

Although contracts expressed as comments are better than nothing, it is difficult to ensure that the contracts expressed are complete and are satisfied by both parties. Design-by-Contract is much more powerful if contracts can be verified in some way.

5.2 Annotated Development with Run-Time Checks

A few programming languages and tools support notations for expressing contracts. Examples of such notations include Eiffel (Meyer 1992) (in which contracts are part of the language itself) and iContract (Kramer 1998) (an extension of Java in which specially-formed comments are used to express contracts). Typically, the tools for such languages provide the facility for generating run-time checks of preconditions, postconditions and class invariants. Nevertheless, it is still up to the software developer to ensure that the contracts are completely expressed and that the callers of a method do not rely on behaviour that is not expressed in the contract. Likewise, the developer must ensure that where preconditions and/or postconditions are redefined in an overriding method definition, the new contract conforms to the inherited contract.

Furthermore, even if run-time checks are enabled during testing, there remains the possibility that testing has not exercised all possible targets at every point of call, or that broken contracts occur in rare cases that have been missed due to insufficient coverage. Some contracts (e.g. those involving quantification over all possible values of a type) are either impossible or too expensive to check at run-time.

5.3 Annotated Development with Extended Static Analysis

Another way of using contract annotations is to perform extended static analysis (often involving term rewriting or theorem proving techniques) in order to attempt to prove that the code satisfies the specifications. The most commercially successful example of this approach is Spark Ada (Barnes 1997). Its success results

from starting with a subset of Ada that avoids hard-to-verify elements such as pointers.

When this approach is applied to object-oriented languages such as C++ and Java, the following difficulties have to be confronted:

- The widespread use of pointer and reference types makes it impossible in practice to perform full static analysis except on small snippets of code because of the potential for aliasing. It is not possible to avoid reference types in Java, and it is only possible to avoid pointers and references in C++ if polymorphism is not used. In order to perform useful analysis of larger sections of code, it is necessary to make sweeping assumptions to limit the extent of aliasing. While these assumptions may frequently hold, this approach cannot be justified in safety-critical work.
- Traditional programming languages are not designed to be verifiable and have features that make verification difficult unless additional information is provided. For example, object-oriented languages typically allow a variable or parameter of any non-primitive type to have a null value. When contracts are added, it becomes necessary to include a great many preconditions, postconditions and class invariants stating that certain entities are not null. If the developer forgets to add these, the correctness conditions become unprovable.
- Complex data structures are often used to store data that is conceptually simple. For example, a tree structure may be used to store a set of records, and additional index structures may be added to enable fast searching on multiple keys. To the clients of the class that maintains this data, the internal structure is irrelevant and the operations are much better specified in terms of a simpler abstract model. Therefore the programming language needs to be supplemented not only by a means of expressing contracts but also a means of declaring an abstract data model and its relationship to implementation data.
- Programming languages do not have sufficiently powerful expression syntax to express many contracts. In particular, quantification and associated concepts from first-order predicate calculus need to be expressible; so the expression sub-language needs to be extended. This may lead to confusion in the mind of the user, because there are different expression sub-languages depending on whether the context is specification or code.
- Integer arithmetic in C++ and Java is subject to wrap-around when the result is too large to represent. This is incompatible with the interpretation that is generally required in specifications (Chalin 2003). Annotated development systems either ignore this problem or provide a context-sensitive interpretation of expressions.

The state-of-the-art in object-oriented annotated development is represented by tools such as ESC/Java (Flanagan et al 2002) which is based around annotated Java. We note that the documentation for ESC/Java states that the system is deliberately unsound in some respects – because the price for soundness in the

context of Java would be a great reduction in the practical usefulness of the tool. Nevertheless, these tools represent a substantial achievement and are capable of detecting many programming errors. Where there is a requirement to assess the correctness of Java code in a safety-related project, we consider that the use of such tools would be very worthwhile as they are likely to discover many of the coding errors. What they cannot do (except in simple cases) is guarantee to find *all* of the design and coding errors (or prove the absence of any), or prove more generally that the program fulfils the requirements.

A further difficulty (affecting all forms of static analysis) is that the degree to which programs containing loops can be verified is severely restricted unless precise and complete loop invariants are available (since the program state following a loop cannot be computed without one). However, determining loop invariants is tedious and often very difficult. In order to achieve the twin goals of verifiability and productivity, the number of hand-written loops needs to be drastically reduced.

We also consider that a code-centric notation with optional annotations is far from ideal. Developers will be tempted to write the code first and add the specification annotations later. This is likely to lead to incomplete specifications and hard-to-verify code. Instead, specifications should be compulsory and central to the notation; code should be optional and subservient to specifications.

6 Verified Design-By-Contract

The limitations of informal and semi-formal implementations of Design-by-Contract are avoided if all contracts are formally verified without making assumptions that cannot be justified or sacrificing soundness in other ways. We refer to this approach as Verified Design-by-Contract.

The “Escher” project was conceived with the goal of developing a toolset to support Verified DBC with close to 100% automated verification. The system is intended for use in applications at all safety integrity levels. We now present an outline of the toolset.

6.1 Principles and Notation

In pursuit of our goal we adopted the following principles:

- Code should serve only to implement a corresponding specification;
- The notation should support specifications based on abstract data models with refinement to implementation models;
- The notation should be designed to facilitate automated verification, avoiding the problems of notations based on programming languages.

We also felt that the notation should avoid mathematical symbols that are not familiar to ordinary software developers, since many developers are put off by the highly mathematical notations of some formal languages.

These principles were embodied in the *Escher Tool*, which has been commercially released as the product *Perfect Developer*. The tool is based around a notation designed for the expression of functional requirements, specifications (of which contracts are a part) and implementation code.

Returning to our example, a declaration of the *display* method in the notation of *Perfect Developer* might look like this:

```
// File DisplayedElement.pd
import "Screen.pd";

class DisplayedElement ^=
interface

  // Display the element in the given rectangle
  deferred schema draw(s!: Screen, r: Rectangle)
    pre r.height >= minHeight,
        r.width >= minWidth
    assert isDisplayedOn(s', r),
        s'.isSameOutsideRectangle(s, r);

  deferred ghost function
              isDisplayedOn(s: Screen, r: Rectangle): bool;
end;
```

We refer to an inherited postcondition as a *postassertion* because it is necessarily incomplete; hence the 'postcondition' part of the contract of *display* is introduced by the keyword **assert**.

A 'ghost' function *isDisplayedOn* has been declared in order to properly define the first part of postassertion (i.e. that on return, the element is displayed on the screen in the specified window). This function will be defined in derived classes such that it returns **true** if the window contains exactly the displayed element (but without being concerned with the details of how it is drawn, as such detail belongs in the *draw* method). Since it has been declared **ghost**, no code will be generated for it; its declaration exists solely to facilitate specification and verification.

The second part of the postassertion expresses the requirement that calling *draw* changes no part of the screen outside the given rectangle. So we are able to specify (and verify formally) not only that *draw* correctly draws the element in the rectangle, but also that it does not corrupt any other part of the screen. The method *isSameOutsideRectangle* of class *Screen* will be another ghost function.

6.2 Verification Conditions

In common with most other formal method tools for software development, *Perfect Developer* performs type checking on the input text and generates *verification conditions* (also known as *proof obligations*). Each verification condition is a mathematical statement, and for a correct program, all the verification conditions will be true theorems. The tool is designed to ensure that, apart from a small number of documented limitations, the converse is also true: that is, if a program's verification conditions are all true theorems, the program correctly implements its specification (subject, of course, to the availability of sufficient resources and to the correct behaviour of the hardware on which it is run, the compiler and linker used to process the generated code, and the tool itself).

Verification conditions are generated to express 47 separate aspects of correctness, including the following:

- Every method precondition is satisfied at each point of call;
- Every constructor and procedure satisfies its postcondition and postassertions;
- Every function delivers its declared result value;
- When one method overrides another and declares a new contract, the new contract respects the old;
- Class invariants are established by all constructors and preserved by all methods;
- Loop invariants are established and preserved;
- Loops terminate after a finite number of iterations;
- Assertions embedded within an implementation are satisfied;
- Behavioural properties specified by the user are satisfied;
- Explicit type conversions always succeed.

By providing a mechanism to express expected behaviour, we make it possible to prove that the program satisfies safety properties and other functional requirements.

When generating the verification conditions for code, the tool computes the program state forwards from the start of each method. Initially, the known program state comprises the method precondition, the class invariant, and any declared type constraints. At any point where a verification condition is required (e.g. a method call, an assertion, or the end of the method), it generates the theorem:

current state ==> required condition

where *current state* is the accumulated program state and *required condition* is the expression that should hold at that point (e.g. the precondition of a called method, or the expression asserted, or the postassertion if we are at the end of a method body).

6.3 Proving the Verification Conditions

In order to maintain high productivity, we use a fully automatic (i.e. non-interactive) theorem prover to process the verification conditions. We decided on an automatic prover because commercial software development organizations typically have neither the time nor the skilled staff needed to develop mathematical proofs, even with computer assistance.

The prover uses a combination of conditional term rewriting and a first-order theorem prover based on a modified Rasiowa-Sikorski deduction system. This combination was chosen because first-order reasoning is easier to automate than higher-order reasoning. Although some features of the notation (such as dynamic binding) cannot be expressed in first-order logic, the instances where higher-order reasoning is needed are infrequent and conform to standard patterns, so they can be handled by term rewriting.

The logic underlying the verification conditions is a logic of partial functions (because of the presence of functions with preconditions). However, it is possible to use a classical 2-valued logic in most parts of the prover, by ensuring that for any term involving partial functions, either the preconditions have been shown to hold, or there is another verification condition stating that they do so.

6.4 Reporting Successful and Failed Proofs

Automated theorem provers typically generate proofs that are hard for humans to follow. Therefore, *Perfect Developer* transforms successful proofs into a hierarchical format designed for human consumption, allowing them to be inspected or checked if required.

Failed proofs typically indicate errors. Our goal is to provide the developer with sufficient information to identify the cause of the error. This has proved to be a difficult task; nevertheless we have been moderately successful.

6.5 Developing Code from Specifications

While it is certainly possible to use Verified Design-by-Contract during the specification and design phases only, productivity can be increased by using automatic or semi-automatic code generation. Also, we have already mentioned that the correct design of loop invariants is a difficult task. This burden on the developer can be reduced if most loops can be generated automatically from specifications.

Our toolset therefore supports refinement of specifications to code (still within the same notation) not just manually but also (in many cases) automatically. Verification conditions are generated to ensure that manual refinements precisely conform to the specification. The code is then automatically refined to a slightly lower-level notation internally before being translated to a standard programming language.

6.6 Results

Perfect Developer has been used by our own organization and by others for a variety of applications. Metrics relating to three very different applications are given in Table 1. The applications illustrated are the *Perfect Developer* compiler/verifier itself, a terminal emulator, and a substantial subsystem of government information system that was originally specified using the CREATIV toolset (Warren & Oldman 2003).

	Compiler/ verifier	**Terminal emulator**	**Government IT system**
Perfect source lines[3]	114720	3192	13486
Generated C++ lines[4]	229367	6752	-
Verification conditions	13144	1349	2631
Prover success rate[5]	≥ 96%	≥ 98.0%	≥ 99.6%
Seconds/verification condition[6]	4.5	2.4	3.8

In both projects where C++ code was generated, the number of lines of generated C++ is about twice as great as the number of lines of specification and explicit refinement. This is notwithstanding that the *Perfect* source contains comments and some specification elements (e.g. preconditions and behavioural properties) that have no counterpart in the C++. We estimate that an equivalent handwritten C++ program would contain 1.5 to 2 times the number of lines of generated C++, so the developer writes only one-third to one-quarter the amount of *Perfect* text that he/she would in C++, further reducing the opportunity to introduce errors.

The number of loops appearing in the generated C++ outnumbers loops (provided by way of explicit refinements) in the source text by a factor of thirteen to one. Thus we have succeeded in relieving the developer of much of the chore of designing loop invariants. Nevertheless, we feel that further improvement is possible in this area, since many of the remaining loops conform to a common pattern.

The lower bound of the prover success rate varies from 96% to 99.6%. In the case of the compiler/verifier, the figure is nearly two years old because we have not investigated a sufficiently large sample of failed proofs for some time.

---

[3] Including comments and extra line breaks within complex expressions to enhance readability

[4] Total of header and code files; no comments; no line breaks within complex expressions except at right margin

[5] Percentage of verification conditions that we believe to be provable for which the prover produced a proof without the need for additional proof hints

[6] Average per verification condition attempted, including unsuccessful proof attempts

Significant improvements have been made to the prover since the figure of 96% was obtained and we believe that the true figure is nearer 98% now.

The use of the tool has resulted in the detection of a number of significant bugs. For example, in the compiler/verifier, a proof failure highlighted a condition in which invalid C++ could have been generated. In the case of the terminal emulator, the protocol specification was found to contain an ambiguity, despite having been in use for five years.

7 Other Issues with Object-Oriented Development

Although the safety of polymorphism with dynamic binding is usually regarded as the most serious issue arising from the use of object-oriented technology in safety-critical systems, a number of other concerns have been raised. We comment briefly on some of these here and, where applicable, the solutions we adopted in the design of *Perfect Developer*.

7.1 Traceability

Standards such as RTCA DO-178B include the recommendation (depending on criticality level) to trace all code to requirements. In the absence of dynamic binding, this is relatively straightforward to achieve. Each procedure at the outermost layer of the software is typically present to support directly a stated functional requirement. Static analysis of the program can identify all lower-level procedures that are directly or indirectly called from the outermost procedures. Thus a lattice can be generated in which every procedure is directly or indirectly linked to one or more functional requirements.

Problems arise if a particular branch of a conditional (e.g. if- or switch-statement) is never executed because its condition can never be satisfied. Such situations can be hard to identify unless formal analysis is used. The associated code will appear to be linked to a requirement but is in reality dead or deactivated.

Dynamic binding complicates the situation because when dynamic binding is present, it is generally not possible to determine statically what method is called. However, we can take an alternative approach, based on treating method postconditions as low-level requirements. In our example:

- We define a low level requirement: “Every displayable element can be displayed in a rectangle within the screen by calling its *draw* method”.
- For every class in the *DisplayedElement* hierarchy, the *draw* method is implemented so as to satisfy this requirement. The details of the implementation will vary from one instrument class to another.
- At various points in the application, in support of higher-level requirements, a flight instrument will need to be displayed in a rectangle. The programmer inserts a call to *draw* at each such point, knowing that the need coincides with the low-level requirement that *draw* satisfies.

Thus we trace the high-level requirements of the application to the low-level requirements associated with called methods such as *draw* (possibly going through some statically-bound method calls on the way). Separately, for each class derived from *DisplayedElement*, we trace the implementation of *draw* to the low-level requirements defined for that method. If each low-level requirement can be traced to some high-level requirement in this way, and every piece of code can be traced to some high-level or low-level requirement, we have achieved traceability even in the presence of dynamic binding.

Problems arise if a method is never called for some class(es); for example, we might define a type of *DisplayedElement* that is never displayed. Again, this situation can only be found in the general case by formal analysis.

We note than when formal verification is performed, the proofs that requirements are met contain all the information needed to trace the requirements to the code that implements them. It is our intention to extend *Perfect Developer* to generate a trace lattice automatically from the proofs.

7.2 Worst Case Execution Timing

In real-time systems it is required that certain program segments complete within defined deadlines. Where the program includes method calls that are subject to dynamic dispatch, the execution time will depend on the methods actually called and it is therefore difficult to determine statically.

A solution is to divide up the maximum allowable execution time of a program segment into a budget for each individual method call and a remainder for other statements. This can be done in such a way that the total execution time (taking account of any loops involved) will meet the deadline, as long as no individual method call exceeds its budgeted time.

To ensure by design that each method call completes within its budget, we can include the time budget in the contract of the nominal target. The method's side of the contract now reads:

- Provided my precondition ***P*** is satisfied on entry, I promise to return within time ***T*** in a state satisfying my postcondition ***R***.

When one method declaration overrides another, the time budget of the overridden declaration is inherited by the overriding declaration by default, just like the precondition and postcondition.

We saw previously that when one method overrides another, instead of inheriting the contract as-is, it may improve on it (i.e. require less and/or deliver more). Provided we are only interested in maximum execution times (and not also in minimum execution times), the contract of an overriding method might improve on the overridden contract by promising to complete in a shorter time.

7.3 Dynamic Memory Allocation

Dynamic memory allocation is generally avoided in safety-critical software. This policy is typically justified on the grounds that memory allocation operations may fail due to insufficient memory or excessive fragmentation, and that the time taken to perform them will vary depending on the history of calls to allocate and release memory.

Object-oriented programming languages typically rely on dynamic memory allocation to allocate all objects of non-primitive types. Therefore, if safety-critical systems are constructed using object technology, it is necessary to establish policies for the safe use of dynamic memory allocation. We suggest here two such policies.

The first policy is to use dynamic memory allocation during the initialisation phase only. In our flight instrument example, we would expect that the set of all elements that might need to be displayed is known at the start. We can therefore create them all during initialisation, even if not all of them need to be displayed immediately. Other objects (such as values of type *Rectangle* in our example) can be implemented as value types, avoiding the need for dynamic memory allocation when creating them.

This policy is comparable to allocating all data statically when a procedural approach is used. It is likely to be adequate for many safety-critical systems. However, it would not suit a system that handles a varying number of objects, such as an air-traffic control system handing a varying number of aircraft.

Our second policy covers this situation by maintaining a free-list for each class that has a varying number of instances. In order to run in bounded memory, there must be a known upper bound on the number of instances of each class. We can initialise each free-list with the corresponding number of instances. Provided the upper bounds are respected (which will be typically be enforced by class invariants), there will never be a need for dynamic memory allocation other than from the free-lists. Allocating from a free-list in bounded time can be easily implemented. The free-list mechanism can be provided either by the supplier of the compiler and associated libraries (as we do with *Perfect Developer*), or (in some languages) by declaring a custom allocator for the classes concerned.

This policy is, in essence, similar to declaring a static array of objects tagged by ‘in use’ flags, allocating and releasing slots in the array as objects are created and destroyed.

If we have control over the standard memory allocator, we can choose not to populate the free list in advance. When allocating an object, if the corresponding free list is empty then we use the standard memory allocation mechanism instead. Provided that no memory has ever been released via the standard mechanism, allocation will not involve searching multiple free blocks and can therefore be performed quickly. We still need an upper bound on the number of objects of each type so that we can compute the maximum amount of memory needed and ensure that this amount is available.

7.4 Overloading

Object-oriented languages typically provide a mechanism for declaring multiple methods with the same name, distinguished only by the numbers and/or types of parameters. This mechanism is known as *overloading*. The compiler decides which declaration is the intended target of a method call by choosing the one whose formal parameter list best matches the actual parameters, according to some set of criteria.

Overloading can be very useful and, by itself, is not dangerous. The symbol "+" has long been used to stand not only for the addition of integers but also for the addition of real numbers. Similarly, it is quite natural to use the call "print(expression)" where we are happy to accept the default printing format, and "print(expression, format)" where we wish to be more specific.

However, we consider that the combination of overloading with automatic type conversion is dangerous because it brings the possibility that more than one method declaration may match a particular call. The choice made by the compiler may not correspond to the intention of the user, who may not have realized that the ambiguity existed. The situation is even worse if the language also allows trailing parameters to be omitted in actual parameter lists by providing default values, as in C++.

The solution we adopted is not to perform implicit type conversions (these are, in any case, undesirable in languages used for safety-critical software development). We make one exception to this rule to allow the type of a value to be automatically converted to a supertype (otherwise the notation becomes very clumsy to use). This exception raises the possibility of ambiguity, so we explicitly forbid any instance of overloading for which it is possible to construct an ambiguous call. We have not found this restriction to be onerous in practice; indeed, we find that the corresponding error message is only triggered where a mistake has been made, or the same name has been used for two unrelated operations.

7.5 Template Instantiation

Templates (known as *generics* in Ada) are a feature of many object-oriented languages. They have been found to be very useful in developing re-usable components, especially for representing collections of objects.

However, it is possible for template declarations to make assumptions about the types with which they are instantiated. This carries the risk that a template may be instantiated with types that violates these assumptions.

One way of avoiding this danger is to perform formal verification of each template separately for each type with which it is instantiated. Although simple in concept, this has the drawback that the verification process is substantially lengthened if there are many different instantiations.

The solution we adopted is to make the assumptions explicit by providing syntax for *instantiation preconditions* in the specification notation. We create a contract between a template declaration and the code that instantiates it, similar to the contract between a method and its callers. The template declaration is formally verified just once and the instantiation preconditions are assumed to hold during this process.

7.6 Reference Semantics and Aliasing

In most object-oriented languages, objects are assigned and copied by reference: that is, a pointer to the object is copied rather than the object itself. Some languages (e.g. C++) also support assignment and parameter passing by value, although polymorphism is typically not available then.

It is well known that the presence (or even the mere possibility) of multiple pointers or references to a common object causes substantial problems for static analysis and formal verification. The provision of reference semantics by default is also a source of program errors, such as the use of a normal assignment or equality operation where cloning or deep equality is needed to achieve the desired result. A classic example is where a class is declared to represent a text string. It is natural for strings to have value semantics; yet in Java and some other languages, the *String* class has reference semantics. The best that the designers of Java were able to do to ameliorate this situation was to provide two classes: an immutable *String* class (for which reference semantics are safe since no modification of the object is possible) and a mutable *StringBuffer* class.

In our example, reference semantics are unlikely to be a problem for objects of classes derived from *DisplayedElement* since we are unlikely to store references to them other than at a single place in class *Display*. However, if objects of class *Rectangle* have reference semantics, this may well be troublesome because we are likely to refer to values of type *Rectangle* in many different places.

The solution we adopted is to specify that in our notation, objects of all classes and types shall obey value semantics. Where aliasing is required, variables of reference type may be declared. We have found that in practice, reference variables are rarely needed.

7.7 Inlining

The C++ language supports the **inline** keyword and provides default inlining of methods whose definitions are included within their declarations.

Inlining makes it more difficult to verify that the object code conforms to the source code. However, should this prove to be a problem, most compilers allow inlining to be disabled.

7.8 Suitability of Mainstream Object-Oriented Programming Languages

Safety-critical developers rightly complain about the unsuitability of mainstream object-oriented programming languages for critical systems. The most widely used object-oriented programming language is C++, which inherits nearly all the problems of C and adds a few new ones such as ambiguous method calls.

We agree that use of handwritten C++ code carries risk in safety-critical systems. However, we observe that C is widely used in critical systems, usually in the form of a subset such as MISRA, conformance with which can mostly be checked statically. We consider that MISRA C could be readily extended to include a subset of C++. Ambiguous method calls could be banned, while constructs such as pointer-to-member and the more esoteric features of templates could be excluded. Although the use of such a subset is not an ideal solution, we believe that it could be safer to use than plain MISRA C due to the increased encapsulation available in C++ and the availability of better alternatives to troublesome features of C.

Although it inherits much of the syntactic idiosyncrasy of C++, Java is a somewhat safer language. Unfortunately, its lack of support for user-defined types with value semantics increases the need for dynamic memory allocation. The Microsoft language C# is in many ways similar to Java but supports value types. The provision of a garbage collector in both languages is a boon to commercial software developers but is likely to be unacceptable in real-time systems. This may be less of a problem in future as generational and concurrent garbage collectors (Jones & Lins 1996) become mainstream. We note that a real-time subset of Java has been defined (RTJ 2003).

The Ada 95 language extends the Ada 83 standard by providing support for (among other things) polymorphism and dynamic binding. However, we do not regard Ada 95 as a satisfactory language for object-oriented development. Unlike other languages, it does not syntactically distinguish the parameter on which dynamic binding depends from other parameters, which we consider likely to cause confusion. The notorious “with-ing” problem makes it impossible to construct complex object-oriented systems unless an ugly workaround is used. We understand that both issues are to be addressed in the next revision of Ada.

The ideal solution is to use a language that does not have its roots in C and is designed with correctness and type-safety in mind. The Eiffel programming language is certainly much better designed than the mainstream object-oriented languages but has not been widely adopted.

Many of these concerns are of little or no importance when complete code is generated automatically from specifications expressed in a rigorous notation. The primary requirement is to ensure that the compiler implements the semantics assumed by the code generator. This can be achieved by generating code in a language subset carefully chosen to avoid areas of undefined behaviour and complex constructs that might be troublesome for the compiler. We adopted this

approach in the code generators of *Perfect Developer*. We note that when the tool is configured to generate code in C++, the generated code conforms to nearly all the MISRA C rules, even though we were unaware of MISRA when the code generators were specified. This suggests that we and the authors of MISRA C had similar ideas on which features of C should be avoided.

7.9 Unified Modeling Language (UML)

UML is the most widely used graphical notation for object-oriented analysis and design and is supported by a wide range of tools. Most tools can generate code skeletons from UML diagrams; some go further and claim to generate complete code if enough information is provided.

Despite the widespread marketing of UML tools, it appears that many object-oriented developers - perhaps the majority - manage without them. However, the use of UML may increase as more universities include UML in their computer science courses, and as open-source UML tools mature.

Concerns about UML among safety-critical software developers centre on the lack of a precise semantic definition for the language.

We consider that UML is a useful notation for displaying graphically the structure of a system and the relationships between the system, its components and its users. However, UML is not a substitute for a precise formal specification. Although UML has a formal sub-language called Object Constraint Language (OCL), it is rarely used, poorly supported by commercial tools and much less expressive than *Perfect Developer* notation.

Our toolset therefore allows UML models to be imported and will generate the corresponding skeletons; but the user must add the detailed requirements and specifications. A future version may allow the *Perfect* specifications and refinements to be embedded in the UML model itself.

8 Conclusions

Object technology undoubtedly facilitates re-use to a greater extent than previous programming paradigms. This is clear from the widespread existence and use of application frameworks and large component libraries. It is likely that without object technology, it would not have been economic to develop many of today's complex and powerful commercial applications.

Safety-critical software developers are right to be cautious in adopting new technology; but rather than dismissing object technology because it is not amenable to yesterday's verification techniques, the safety-critical community should seek new techniques to facilitate safe use of the new technology. The most important new issue arising from object technology is polymorphism with dynamic binding, which is tamed by the Design-by-Contract principle. The use of modern

formal methods technology to implement Verified Design-by-Contract provides a basis for safely harnessing the power of object technology in critical systems.

References

Barnes J (1997). *High Integrity Ada: the SPARK Approach*. Addison-Wesley, England.

Chalin P (2003). Improving JML: For a Safer and More Effective Language. *FME 2003: Formal Methods* (Springer LNCS 2805): 440.

Flanagan C, Leino K.R.M, Lillibridge M, Nelson C, Saxe J and Stata R (2002). Extended static checking for Java. *Proc. PLDI, SIGPLAN Notices* 37(5): 234-245.

Hoare C.A.R (1969). An axiomatic basis for computer programming, *Communications of the ACM* 12: 576-580.

Jones R and Lins R (1996). *Garbage Collection*. Wiley, England.

Kramer R (1998). iContract - The Java(tm) Design by Contract(tm) Tool. *Technology of Object-Oriented Languages and Systems*, August 03 – 07: 295.

Mamrak A and Sinha S (1999): A case study: productivity and quality gains using an object-oriented framework. *Software - Practice and Experience* 29(6): 501-518.

Meyer B (1988). *Object-Oriented Software Construction*. Prentice Hall, England.

Meyer B (1992). *Eiffel: The Language*. Prentice Hall.

OOTiA (2003). Handbook for Object-Oriented Technology in Aviation (draft). *OOTiA Workshop Proceedings*, March 5, 2003.

Port D and McArthur M (1999). A Study of Productivity and Efficiency for Object-Oriented Methods and Languages. *APSEC 1999* (IEEE Computer Society): 128-.

Potok T, Vouk M and Rindos A (1999). Productivity Analysis of Object-Oriented Software Developed in a Commercial Environment. *Software - Practice and Experience* 29(10): 833-847.

RTJ (2003). http://www.rtj.org (30 September 2003).

Warren J.H and Oldman R.D (2003). A Rigorous Specification Technique for High Quality Software, *Proceedings of the Twelfth Safety-Critical Systems Symposium*, Springer-Verlag (London).