Development of Formally Verified Object-Oriented Systems with *Perfect Developer*

David Crocker

Escher Technologies Ltd.
3 Archipelago Business Park, Lyon Way, Frimley, Camberley GU16 5ER, UK
Web: `www.eschertech.com` Email: `dcrocker@eschertech.com`

Abstract. *Perfect Developer* (formerly known as the Escher Tool) is a highly productive system for developing reliable software systems using object-oriented methods. Dynamic binding and aliasing are carefully controlled to make the verification problem tractable.

1 Background

Formal methods have been used for many years in the development of safety-critical software but have yet to make it to mainstream software development. Barriers to wider use of formal methods include the requirement for users of formal methods tools to have extensive mathematical knowledge, the labour associated with assisting tools in discharging proof obligations, and the lack of support in most tools for object-oriented methods. These issues are addressed by *Perfect Developer*.

2 Limitations of existing O-O languages

The use of existing object-oriented program languages to develop formally verified software runs into both technical and practical difficulties.

The primary technical difficulties we have identified are:

- Unconstrained polymorphism. Variables, parameters and return values with non-primitive types are typically all polymorphic (i.e. an object of any class derived from the declared type is acceptable). This greatly increases the potential for dynamic binding and makes specifications and code very hard to reason about.
- Default reference semantics. The use of reference semantics as the default or only semantics for assignment and parameter passing of class variables greatly increases the potential for aliasing. When combined with polymorphism and dynamic binding, this leads to many situations in which it is impossible to prove that parameters to a class method (including the 'self' or 'this' parameter) are all distinct, which typically makes it impossible to prove correctness of methods that change their parameters or 'self'. We have observed that unintentional aliasing also makes a significant contribution towards software errors.

The practical difficulties are:

- Syntax for preconditions, postconditions, invariants and many other constructs needed for specification are not provided in the language. The usual solution is to insert these constructs as specially-formatted comments. This apparent demoting of specifications conveys the wrong message to developers.

– To make formal methods of software development acceptable to the mass market, the additional time spent writing specifications must be balanced by time savings elsewhere. The obvious solution is to generate code automatically from specifications; however, this also requires the language to be extended (by providing a syntax for an omitted code body).
– Programming languages do not support data refinement, which is a key tool in object-oriented software development using formal methods.

These difficulties mean that the most that can be achieved using existing programming languages is extended static checking (e.g. ESC/Java [1]). Such tools may be a useful bridge between existing software development practice and true verified software development; but even if the user can be persuaded to annotate the program with preconditions, invariants etc., complete formal verification is far from possible.

3 Perfect Developer

Perfect Developer (a development system for producing perfect software) takes the approach that the notation must resemble typical programming notations (i.e. avoid mathematical notation) but that code must take second place to specifications. Accordingly, it uses its own notation [2]. Concepts that can be expressed in the *Perfect* language include:

– Classes with single inheritance and dynamically bound methods
– Parametric polymorphism
– Class invariants and type constraints
– Method preconditions, postconditions, variants and post-assertions
– Quantification over sets, bags and sequences
– Expected behaviour of the system as a whole and of subsystems, including 'what if' scenarios

To avoid excessive use of unconstrained polymorphism, we distinguish between the type **T** and the type **from T**, where **T** is any non-final class (Ada 95 makes a similar distinction). Thus the user indicates explicitly where polymorphism is required.

In order to reduce the aliasing problem, *Perfect* uses value semantics by default for assignment and parameter passing. To avoid excessive copying, value semantics are simulated using reference semantics; copying is avoided wherever possible and where copying is necessary, typically only some part or parts of an object need to be copied. Reference types are provided for those situations in which intentional aliasing is required.

To develop a software system or component with *Perfect Developer*, a set of required properties of the system or component is specified. Hardware devices and other subsystems with which the software will interact may also be described and relevant behaviour specified. An assembly of classes is then designed to model and encapsulate the stored data and perform the required operations. Contracts are written for the class methods. *Perfect Developer* will attempt to generate code to satisfy the contracts where none has been provided. The developer may refine both class data and method code in order to meet performance targets.

Proof obligations are generated asserting that all contracts are honoured by both parties, that required properties will be observed, that all constructs will terminate and that refinements are valid. *Perfect Developer* attempts to discharge them using a fully automatic theorem prover, on the grounds that most software developers have neither the skill nor the time to assist in discharging proof obligations.

Final code is generated in C++ (code generation in Java and Ada 95 is under development).

4 Handling inheritance and dynamic binding

Dynamic binding continues to be a potential problem in that calls to dynamically bound methods cannot be expanded during verification.

If the software system is to be validated without regard to extensibility, it would be possible to enumerate the set of possible types of each polymorphic variable and perform validation with respect to all possible types of all polymorphic variables. We do not currently do this but may offer it as an option in the future.

A better solution is to recognise that in any family of classes with a common ancestor, for any method declared in the common ancestor and defined or redefined in each class, all the method declarations implement some common purpose. In some cases the method represents the definition of some property of the class; in other cases, the method modifies the class so as to make it satisfy some property. In the latter case, a non-compiled or 'ghost' dynamically bound function can be defined to express the property and the original method specified in terms of this function. Although this places a greater burden on the user, it helps greatly in clarifying the specification of dynamically bound methods as well as making them amenable to validation.

We note in passing that most class methods must be validated separately in each non-abstract class into which they are inherited without being overridden, since it is frequently the case that a method definition is valid in the class in which it is defined but is invalid in the context of a derived class (e.g. because it does not take account of additional variables in the derived class).

5 A large case study: *Perfect Developer* itself

Perfect Developer is itself implemented in *Perfect* apart from the user interface functions in the IDE module.

At the time of writing (May 2001), the source for the compiler/verifier comprises 105000 lines of *Perfect* (including comments) from which 176000 lines of C++ (without comments) are generated.

When validation of the entire system is performed, 115000 proof obligations are generated. Using default settings, the theorem prover discharges 87.6% of these in under 5 days. The success rate is currently increasing by between 1 and 2% per month as we improve the prover and eliminate specification and coding errors, while at the same time the average time spent on each obligation has decreased in the last few months from 5 seconds down to 3.5.

Analysis of unproven obligations indicates that about half are the result of incompletely specified contracts (due in part to the bootstrap process used to develop *Perfect Developer*) and most of the rest are provable in principle but beyond the capability of our present prover within a reasonable time limit. However, failed proof obligations do occasionally reveal incorrect coding or an inconsistent specification and did in one case reveal an error in the *Perfect* language definition itself.

6 Conclusions and future work

We have shown that formal methods can be used to develop a large and complex application in an object-oriented style with high productivity. Despite the relative immaturity of our prover, we have achieved a substantial degree of automated validation.

Perfect Developer is currently available in a teaching and evaluation edition. Commercial release is due in September, by which time we expect to support exceptions and multithreading in the language. Work continues on the theorem prover and we expect that the use of term indexing techniques and better unification algorithms will significantly improve the speed and success rate of validation.

References

1. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. Research Report 159, Compaq Systems Research Center, December 1998. Available at http://research.compaq.com/SRC/esc/.
2. *Perfect* Language Reference Manual. Escher Technologies Ltd., March 2001.