Developing Reliable Software using Object-Oriented Formal Specification and Refinement [Extended abstract prepared 24 March 2003]

Dr. David Crocker

Escher Technologies Ltd., Mallard House, Hillside Road,
Ash Vale, Aldershot GU12 5BJ, United Kingdom
dcrocker@eschertech.com
http://www.eschertech.com

Abstract. It is our view that reliability cannot be guaranteed in large, complex software systems unless formal methods are used. The challenge is to bring formal methods up to date with modern object-oriented techniques and make its use as productive as traditional methods. We believe that such a challenge can be met and we have developed the Escher Tool to demonstrate this. This paper describes some of the issues involved in marrying formal methods with an object oriented approach, design decisions we took in developing a language for object-oriented specification and refinement, and our results in applying the tool to small and large projects.

1 The challenge of developing reliable large-scale software systems

The availability of ever more powerful processors at low prices has prompted organizations to attempt the construction of large, complex software systems. The larger the software system, the less effective testing is as a means of ensuring the absence of faults. Formal methods have for many years offered a logical solution to the problem of software reliability but have generally come at a high cost, demanding developers with considerable mathematical skill and costing many additional hours of developer time to assist with discharging proof obligations.

Our goal in developing the Escher Tool was to bring formal methods to modern object-oriented and component-based approaches to software development while at the same time achieving developer productivity no worse than standard informal techniques. We chose to base the system on refinement for two reasons: firstly because of the considerable difficulties in formally analyzing programs hand-written in conventional programming languages, and secondly because of the promise of increased productivity provided by automating much of the refinement and all of the code generation.

2 Formal methods and object technology: a marriage made in heaven?

Opinions differ as to the essential features of object-oriented languages, but elements often cited include: encapsulation, inheritance, polymorphism with dynamic binding, object identity, and the Liskov type substitution principle. We will consider the implications each of these has for formal methods in turn.

2.1 Encapsulation

Encapsulation is very helpful because to a large extent it permits formal verification to be done component-by-component. Simultaneous formal analysis of both the system and all its components need only be carried out to verify a few properties (e.g. absence of unbounded indirect recursion). Furthermore, encapsulation permits refinement of a component's abstract data into efficient implementation data structures without affecting the development or verification of the rest of the system.

2.2 Inheritance

Inheritance is easily handled formally by copying the inherited elements from the definition of the parent class into the derived class definition.

2.3 Polymorphism and dynamic binding

These present serious challenges. When a derived class method overrides an inherited method, the traditional approach (e.g. in the Eiffel language) is to require the overriding postcondition to be a strengthening of the inherited postcondition. However, when such an approach is used it is frequently not possible to express in full the necessary and sufficient state changes in the postcondition, so postconditions are often incompletely specified. This may be acceptable to static checking tools where method body code is written by the developer, but is unacceptable in a refinement context where the method body will usually be generated automatically from the postcondition.

Our solution is to use a two-part postcondition. One part (which we refer to as the *postcondition*) describes the complete state change required (including the frame) and is not inherited. The second part (which we call a *postassertion*) is a predicate that must be a consequence of the first part and is subject to the usual inheritance and overriding rules.

2.4 Object identity

Object identity is a source of serious problems due to potential aliasing between objects. For the verifier, the possibility of aliasing makes it hard to reason about components that deal with several objects of similar type and modify one or more of those objects, unless it can be guaranteed that all the objects are distinct. For the developer, the need to consider when to use shallow equality vs. deep equality, or assignment vs. cloning, is a rich source of errors.

We have observed that there are many situations in which object identity is both unnatural and undesirable (the classic example being a *String* class). By implementing value semantics by default, the problems of aliasing are avoided. Reference semantics are available on demand where the developer has a genuine need for object identity.

A further advantage of using value semantics by default is that there is no need for the artificial distinction between primitive types and class types that is present in traditional object-oriented programming languages, so that types such as **int** and **bool** behave like (and are defined as) "final" classes.

The use of value semantics does impose an additional execution-time overhead due to the need to copy objects (or, more typically, parts of objects) at times. Copying can largely be avoided by using shared objects in the generated code and a copy-on-write mechanism; consequently we have not found the over-

head to be a serious problem in commercial applications. Where speed is critical, the user has the option of specifying reference semantics for selected objects.

2.5 Liskov type substitution principle

This principle states that wherever an object of some class is expected, an object of a derived type can be substituted. We take issue with the safety of this principle where reliability is paramount, because it requires the class hierarchy to be extremely carefully constructed if a method that expects its parameters are of particular types is in all cases to behave correctly when it is given parameters of alternative types that were not envisaged (any may not even have existed) when the method was written. If the declared parameter type is a deferred class, the method author might reasonably be expected to cater for as yet unknown conforming types, but what if the type is a concrete type that has been subsequently been built on? Although formal specification and verification can be used to ensure correctness in these cases, specification in the presence of polymorphism is more complex than specification when types are known exactly and it seems unreasonable to force developers to spend additional effort in order to cater for what may only be a distant possibility of derived classes being substituted in the future.

One solution to this issue is to enforce the practice of requiring all non-leaf classes in a class hierarchy to be deferred (i.e. non-instantiable) so that a variable is explicitly either of a single fixed type or belongs to a hierarchy. However, our approach is instead to distinguish between exact types and unions. In the Escher Tool, the type "**from** *T*" is defined as the union of all non-deferred types in the set of *T* and its (direct and indirect) descendents, whereas *T* alone means precisely the type specified in *T*'s class declaration. We note that Ada 95 takes a similar approach, distinguishing between *T* and *T*'**class**.

When the user does choose to allow polymorphism (by declaring a variable or parameter of type **from** *T* for some type *T*), it is necessary to ensure that the Liskov type substitution principle holds. We therefore generate proof obligations requiring that when a method of class *T* is overridden in a class derived from *T*, the precondition is not strengthened and the post-assertion is not weakened. This ensures that provided the method caller satisfies the contract of the method as declared in class *T*, it will also satisfy the contract for the corresponding method in any class derived from *T*.

3 Building safety and verifiability into an object-oriented language

From the preceding section, it is evident that traditional object-oriented programming languages are not suited to complete formal verification (notwithstanding the achievements of advanced static analyzers such as ESC-Java [1]). Furthermore, our goal was to combine formal specification, refinement and programming in a single language, in order to avoid any need to switch between different syntax, semantics and even underlying logic when moving from specification to implementation.

Our research resulted in an object-oriented specification/refinement language described in [2]. The most commonly used object-oriented languages are full of traps for the unwary (many being present due to an excessive desire to maintain compatibility with older languages). We were determined to produce a safe but powerful language, resulting in the following design decisions.

3.1 No side effects in expressions

Functions, operators and other expression constructs have no side effects. This not only makes evaluation order immaterial, it simplifies formal analysis.

3.2 Overloading of operators and other methods

Used correctly, overloading is a powerful tool, but it interacts disastrously with automatic type conversion and default parameters. Therefore we do not provide automatic type conversions (save for the widening of one type to a union that includes that type) or default parameters. Furthermore, we do not permit the declaration of a set of overloaded declarations such that it is possible to construct a parameter list that matches more than one of them.

3.3 Casts

Casting constructs are essential in a system using class hierarchies. In addition to type comparison operators, we provide two type-casting operators. The "**as**" operator widens a type to a union that includes the original type. The "**is**" casting operator provides a type narrowing conversion (asserting that the actual type of the expression concerned conforms to the specified type); naturally a proof obligation is generated every time it is used.

3.3 Unions

We have already described our definition of the construct "**from** *T*" in terms of a union of classes. We also provide a type union operator, allowing variables of united types to be declared. The use of such unions is completely safe because their use is formally verified (values of united types are extracted using an **is**-cast); however we find they are rarely needed (save for the very common case of uniting **void** with another type) because a class hierarchy normally provides a better solution.

4 Improving the Productivity of Formal Methods

Two of the barriers to the widespread adoption of formal methods have been the low productivity they are perceived to offer and the mathematical skills they demand of their users. Software development organizations have been reluctant to bear these costs except in safety-critical areas where the benefits of improved reliability provide an adequate payback.

We believe it is possible in principle for formal methods to provide substantially greater productivity than non-formal development. Our approach to raising productivity is to automate as much of the implementation and verification processes as possible. A combination of recent advances in automated reasoning technology, cheap processor power and careful language design has allowed us to come near to our target of 100% automated proofs. Automated refinement is also offered by the Escher Tool so that many components can be generated directly from their specifications.

By avoiding the need for developers to assist in discharging proof obligations, the requirement for users to have substantial mathematical skills is eliminated. We therefore use syntax more reminiscent of

programming than mathematics to describe specifications so as to increase the accessibility of the system to ordinary software developers.

Loops are a particular problem because of the need for knowledge of a loop invariant when performing verification. Although there has been some research on automated determination of loop invariants [3], it is impossible to avoid the need for users to declare loop invariants in more complex cases. Fortunately, the presence in the language of elements such as quantifiers and comprehension operators, together with automated refinement, means that it is rarely necessary to code loops explicitly.

5 Results and experience

Our research and development led to the Escher Tool being previewed in September 1999 at the World Congress of Formal Methods and commercially released in September 2001 under the product name *Perfect Developer*.

Aside from academic examples, the Escher Tool has been used to implement real-world systems including a terminal emulator and the Escher Tool itself. Here are some statistics on these two, very different, projects:

	Terminal Emulator	**Escher Tool**
Lines of specification and refinement[1]	3 148	129 000
Lines of generated C++ [2]	6 928	225 000
Number of proof obligations generated	1 456	125 924
% automatically discharged	96.8%	90.3%
Time to process obligations[3]	48 minutes	229 hours
Loop multiplication factor[4]	15	13

The Escher tool is under continuous development and is a moving target for verification, which is why the percentage of unproven obligations remaining is substantially greater than for the terminal emulator. Experience in other projects has shown us that in order to achieve a very high degree of automated validation, it is necessary to freeze the development for a time and concentrate on dealing with validation failures. Analysis of a sample of validation failures from the Escher Tool revealed that about 60% are genuinely unprovable due to incomplete preconditions or class invariants, but it has been the case that as fast as we fix them, new ones are introduced as we add new functionality but fail to get some of the preconditions exactly right first time.

In both projects, proof failures have highlighted significant problems. For the terminal emulator, an inconsistency in the 5-year old protocol specification document was revealed; while a proof failure for the Escher Tool uncovered a language design flaw that could have resulted in the generated C++ failing to compile.

Although our automated refinement technology is in its early stages of development, we have found that manual refinement is only needed for a small proportion of the classes in a system. For example, the

[1] Including comments

[2] Not commented except for heading; no formatting of long expressions/statements apart from line wrap at 120 characters

[3] Using a PC with 1.3GHz AMD processor and 512Mb memory

[4] Number of loops in the generated C++ divided by number of explicit loops in the source, to nearest integer

Escher Tool contains a compiler element (which includes the refinement and code generation subsystems) and a verification element. Despite the fact that the compiler is constructed almost entirely using classes and methods for which no refinement has been manually written, it is able to compile and generate code for the entire Escher Tool in only a few minutes (compared with the several hours needed to compile the resulting C++ code). This also reinforces our view that the overhead associated with using value semantics in place of reference semantics is not severe. The verifier is more heavily refined due to the processor-intensive nature of automated theorem proving; nevertheless the majority of class methods involved are not manually refined.

Conclusions and further work

We have shown that it is possible to develop a large, complex program in reasonable time using formal methods merged with object technology, and that automated reasoning can be used to successfully discharge a very high proportion of the generated proof obligations on a modern PC. The commercial product *Perfect Developer* is now in its second major version, featuring an improved automated theorem prover using backtrack-free splitting [4] and a facility to generate skeleton specifications from imported UML models.

Code generation is currently restricted to C++ and Java. Ada 95 code generation is partially implemented, however the infamous Ada "with-ing" problem [5] prevents the completion of this work pending a revision to the Ada language to address this limitation.

Our main focus is now directed towards further automating refinement, our goal being to fully automate the refinement of class methods in almost all cases, leaving data refinement as the main developer contribution to implementation efficiency.

Another important area of research is to analyze verification failures and attempt to suggest fixes to the source. We note that some work has already been done in this field [6] in relation to ESC/Java.

The construction of programs that are proven to conform to formal specifications is of little value if the specifications do not fulfill user requirements. Some requirements are easy to express as expected behaviors of the system and these can be formally verified, but other categories of requirements prove more elusive. We are working in partnership with a company specializing in formal requirements to address this part of the development process.

References

1. ESC/Java User's Manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000, K. Rustan M. Leino, Greg Nelson and James B. Saxe. See http://research.compaq.com/SRC/esc/
2. Perfect Developer Language Reference Manual. July 2002, Escher Technologies Ltd. Available at: http://www.eschertech.com/product_documentation/LanguageReferenceManual.htm
3. Invariant Discovery via Failed Proof Attempts. 1998, Jamie Stark and Andrew Ireland LNCS 1559, p. 271 ff.
4. Splitting without Backtracking, A. Riazanov, A. Voronkov, University of Manchester, CSPP-10.
5. John Volan's answers to Frequently Asked Questions about the Ada "with-ing" Problem, June 1997. Available at http://www.eschertech.com/WithingProblem.htm
6. Houdini, an annotation assistant for ESC/Java. SRC Technical Note 2000-003, Cormac Flanagan and K. Rustan M. Leino. See http://research.compaq.com/SRC/esc/relatedTools.html