A High Productivity Tool for Formally Verified Software Development

David Crocker and Judith Carlton

Escher Technologies Limited, Mallard House, Hillside Road, Ash Vale, Aldershot GU12 5BJ, United Kingdom.
e-mail: `dcrocker@eschertech.com, jcarlton@eschertech.com`

Revised version: 20 September 2004

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 techniques up to date with modern object-oriented approaches to software design and to make their use as productive as informal 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 Introduction

The availability of ever more powerful processors has encouraged 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 proving verification conditions.

A further barrier to the widespread adoption of formal methods stems from the fact that industry has largely moved to object-oriented (O-O) software technology. Although object-oriented extensions of Z and VDM have been described [1,2], neither has been embraced by industry to the same extent as more established formal methods such as Z [3], VDM [4], the B-method [5], or SPARK Ada [6]. Furthermore, we doubt the wisdom of extending non-O-O languages to include O-O constructs. The O-O paradigm is very different from previous approaches. Although O-O languages need much of the same small-scale structure as their procedural counterparts (for example, a rich expression syntax), the higher-level structuring mechanisms are quite different. Adding O-O features to older languages leads to the kind of anomalies visible in the C++ language and in parts of Ada 95.

Our goal in developing the Escher Tool (now released commercially as *Perfect Developer*) was to combine formal methods with 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 specification and refinement for two reasons: firstly because of the considerable difficulties in formally analysing 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 notations, but elements often cited include abstraction and encapsulation, inheritance, polymorphism with dynamic binding, object identity, and parametric polymorphism[1]. We will consider the implications that each of these has for formal analysis.

[1] This is not meant to imply that these features are specific to object-oriented languages; some of them are present in other languages too.

2.1 Abstraction and Encapsulation

Abstraction and encapsulation are very helpful to the formalist, because to a large extent they permit 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, abstraction and encapsulation permit refinement of a component's abstract data into efficient implementation data structures, without affecting the behaviour of the component apart from the resources it uses.

2.2 Inheritance

On its own, inheritance is easily handled formally by copying the inherited elements from the definition of the parent class into the definition of the derived class. It is when inheritance is used to support polymorphism that complications arise.

2.3 Polymorphism with dynamic binding

Most object-oriented languages provide polymorphism by default; that is, wherever a variable, formal parameter or return value of type T is declared, a value of any type derived from T may be provided instead, under the assumption that such a derived type is a behavioural subtype of T.

We take issue with the safety of this approach where reliability is paramount because it requires the class hierarchy to be constructed extremely carefully. Otherwise, a method that expects its parameters to be of particular types may not behave correctly when it is given parameters of alternative types that were not envisaged (and may not even have existed) when the method was written. If the declared parameter type is a **deferred**[2] class, the method author might reasonably be expected to cater for as yet unknown conforming types; but what if the type is a concrete class, which is only later used as a base for other classes? A developer who extends a class and another developer who uses the same class may have different views on what are the essential behaviours, so behavioural subtyping is at least partly in the eye of the beholder.

Although formal specification and verification can generally be used to ensure correctness in these cases (as will be discussed shortly), specification in the presence of polymorphism is more complex than specification when types are known exactly. 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 declaring all non-leaf classes in a class hierarchy as deferred (i.e. non-instantiable) classes, 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 *Perfect Developer* notation, the type **from** T is defined as the union of all non-deferred types in the set of T and its direct and indirect descendants; 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**.

Where the user does choose to allow polymorphism (by declaring a variable or parameter of type **from** T for some type T), safe use of dynamic binding requires that an object of a derived class may be substituted for an object of its parent class. The conditions necessary to achieve this have been explored by Liskov and Wing [7], popularised by Meyer in the Design-by-Contract approach [8] and are discussed further in [9].

The essence of Design-by-Contract is that when a derived class method overrides an inherited method, the precondition of the overriding method must be satisfied whenever the precondition of the overridden method would be; that is, the overriding methods may weaken the precondition but not strengthen it. Similarly, the postcondition of an overriding method may strengthen the postcondition of the inherited method. This approach works well provided that postconditions are not expected to describe the total state change, which is the case when postconditions are used to annotate program code (for example, in the Eiffel language and in annotated extensions of Java). In particular, the postconditions do not generally include a frame; that is, they place no restrictions on which variables can be changed (except where the initial and final value of a variable are expressly equated in the postcondition). Such postconditions are insufficient in a system that is required to generate complete code from specifications.

Our solution is to use a two-part postcondition when declaring class methods. The main part (which we refer to as the *postcondition*) describes the complete state change required including the frame. It is not inherited by an overriding method declaration, although it can be referred to in such a declaration using the usual *super* notation. It is this part that is refined to code. The second part (which we call a *postassertion*) is a predicate that is required to be a logical consequence of the first part and is subject to the usual design-by-contract rules. The postassertion does not include a frame, because the set of attributes available to be modified in a derived class is typically greater than the corresponding set in the base class, since the derived class may declare additional attributes.

[2] A *deferred* class is a class that cannot be instantiated, but serves only as a base from which to derive other classes. Also called an *abstract base class* in some languages.

In consequence, where a method call is statically bound to a method, both the postcondition and the postassertion can be assumed by the caller when the call returns. However, where the binding is dynamic, the caller may only assume that the postassertion has been satisfied.

It is important to realize that where a method is inherited by a derived class without being overridden, then even if all its verification conditions were satisfied in the context of the class in which it was originally declared, the same may not be true in the derived class. This is because the method may directly or indirectly make a call that is dynamically bound, leading to different behaviour. In general, a method must be verified in every non-deferred class in which it is declared or into which it is inherited[3]. An exception can be made if the method is declared as having no dependence on dynamic binding (which property must of course be verified). We use the keyword **early** to denote such a method, because such methods are useful in other contexts (for example, they can safely be called in contexts where dynamic binding is undesirable, such as within constructors of deferred classes, or within class invariant declarations).

2.4 Object identity

Most object-oriented programming languages implement reference semantics for variables and parameters of all types other than primitive types such as integers and characters. This means that a variable or parameter of non-primitive type holds a pointer or reference to an object stored in a heap area of memory, rather than holding the value directly[4]. The use of reference semantics gives rise to the notion of object identity.

The use of reference semantics is a source of serious problems, due to aliasing between objects. From a formal perspective, 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 choose whether 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 is a *String* class, which is invariably implemented as a library class obeying reference semantics, even though this is highly unnatural to the user[5].

Perfect Developer implements value semantics by default, avoiding the problems of aliasing. 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**[6] 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 overhead 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 Parametric polymorphism

The provision of parametric polymorphism (also called generics or templates) in an object-oriented programming language facilitates the development of reusable classes (especially collection classes). However, the designer of a templated class may need to assume that the classes with which its parameters are instantiated obey certain properties (for example, that they declare a '<' operator that defines a total ordering between objects of the class). Unless the language provides an adequate mechanism for expressing these assumptions, there is a risk that the template will be instantiated with unsuitable parameters.

One way to avoid this problem is not to attempt verification of the template declaration itself but instead to verify each of its instantiations. However, this replicates the verification effort needed and prevents the development of pre-verified generic components.

Our solution is to declare instantiation preconditions for templates instead. The designer of a generic class may specify that the classes with which its parameters are instantiated must be derived from some parent class (similar to the "constrained genericity" facility provided by some programming languages), and/or that those classes must have specific methods that conform to particular precondition/postassertion semantics. This allows a template declaration to be verified independently of its instantiation context. At each point of template instantiation, it is only necessary to verify that the corresponding instantiation preconditions hold.

[3] Provided that all derived classes are re-verified when a new version of a base class is introduced into a project, this also catches errors that occur when a base class is changed in such a way that assumptions made by the developer of a derived class no longer hold. This scenario is one of a number of *fragile base class* problems.

[4] The C++ language does allow the user the choice of reference or value semantics, but dynamic binding can only be used in conjunction with reference semantics.

[5] Some languages attempt to mitigate the effects of reference semantics on strings by providing both mutable and immutable string classes (e.g. *StringBuffer* and *String* in Java). The use of reference semantics poses no problem for users of the immutable version, but the problem remains when the mutable version is used.

[6] A *final* class is a class from which no further classes may be derived; that is, it cannot be inherited by another class declaration.

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 well suited to complete formal verification (notwithstanding the considerable achievements of advanced static analysers such as ESC/Java [10]). Furthermore, our goal was to combine formal specification and refinement to programming-like constructs in a single language, in order to avoid any need to switch between different syntax, semantics or underlying logics when moving from specification to implementation. We therefore created the notation described in [11].

The most commonly used object-oriented languages are full of traps for the unwary - many of which are the result of efforts to maintain compatibility with older languages. We were determined to produce a notation that was both powerful and suitable for safety-critical applications, 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

Overloading allows the user to declare multiple methods or operators with the same name but differing in the number and/or types of parameters taken (just as the '+' symbol is generally used to stand for both integer and floating-point addition, and the '−' symbol represents both negation and difference).

Used sensibly, overloading is a powerful tool; but it interacts disastrously with other features of some programming languages - in particular automatic type conversion and default parameters - resulting in ambiguous method calls.

The notation of *Perfect Developer* provides neither default parameters nor automatic type conversions, save for the widening of one type to a union that includes that type. Furthermore, we forbid the declaration of any set of identically named methods for which it is possible to construct a parameter list that matches more than one of them. In consequence, the issue of resolving ambiguous bindings does not arise.

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 (for example, converting a value of type *Derived* to type **from** *Base*, where class *Derived* inherits class *Base*). The '**is**' casting operator provides the converse type-narrowing conversion (for example, converting a value of type **from** *Base* to *Derived*). It asserts that the actual type of the expression concerned conforms to the narrower type at run-time; naturally, a verification condition is generated every time it is used.

3.4 United types

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. Values of united types are extracted using an '**is**' cast.

In practice, in most situations where unions might be used, it is better to declare a class hierarchy and use polymorphism instead; so the main use of the type union operator is to unite **void** (a class which has the single value **null**) with another type. This form of type union is particularly convenient when declaring recursive data structures. For example, a class *Branch* used for building trees might have members *left* and *right* whose type is the union of *Branch* with **void**. We might instead have allowed any variable x which is declared as having some class type T to be given the value **null**, just as Java and most other languages using reference semantics allow any variable of a non-primitive type to be assigned a null reference. However, if x is permitted to take the value **null**, then the action of calling a member method of T on x or accessing a member variable of x introduces a verification condition that x is not **null** at that point. For verification to succeed, this typically requires $x \neq$ **null** to be stated as an additional precondition, class invariant or postassertion. It is far better to require the developer to specify explicitly where null values are allowed.

3.5 Method overriding

In most object-oriented languages, a method declaration that overrides an inherited method is not syntactically distinguished from a virgin method declaration. This creates the possibility of a user accidentally overriding an inherited method of which he is not aware, usually with serious consequences[7].

Perfect Developer therefore requires every overriding method declaration to be introduced with the keyword **define** if it is overriding the declaration of a deferred method, or **redefine** if it is overriding a non-deferred method, thereby preventing accidental overriding. We note that the C# language likewise provides an **override** keyword.

[7] Even if the developer of the derived class is aware of all the methods of the base class, a developer may later add a method to the base class that clashes with a method in some derived class. This is another *fragile base class* problem.

4 Syntax issues

Most formal notations rely heavily on notation borrowed from mathematics. For example, conjunction and disjunction are usually represented by the symbols $\wedge$ and $\vee$.

We consider that the use of such symbols in a notation targeted at software engineers is a mistake. Many software developers have only a basic grounding in mathematics and are not familiar with these symbols. The characters are not available on standard keyboards and are not easy to find in the most popular word processing programs.

For the sake of ease of learning and developer productivity, we felt it preferable to borrow from programming language syntax instead - for example, by using the symbols & and | instead of $\wedge$ and $\vee$. Likewise, we use keywords rather than symbols to represent quantifiers and to provide the syntactic skeletons for new forms of expression such as set comprehension.

It was tempting to go further and borrow the syntax of constructs wholesale from a popular programming language, much as the designers of Java borrowed most of the syntax from C++. However, it would have been difficult to do so without importing some of the well-known safety problems present in these languages; so we contented ourselves with using snippets of syntax from a number of different programming and specification languages.

The presentation of quantified expressions deserves special mention. We find that the most common use of quantified expressions in software specifications is to state that all elements of a particular collection have some property, or that at least one element of some particular collection has a property. Furthermore, some software developers find the concept of quantification over all elements of a collection easier to grasp than quantification over an infinite type. We therefore provide quantification over both types and over the elements of collections. When teaching new students, we introduce quantification over collections early in the course; quantification over types comes much later.

5 Improving the Productivity of Formal Methods

Barriers to the widespread adoption of formal methods include the low productivity they are perceived to offer and the mathematical skills they demand of their users. Software development organizations are reluctant to bear these costs - even (in many cases) when developing critical systems.

We believe that formal methods have the potential to provide greater productivity than traditional software development processes. In this section we discuss how we have attempted to improve the productivity of formally verified software development.

5.1 Automated verification

Perfect Developer generates 50 different kinds of verification condition [12] in order to express a range of correctness properties including the following:

- The requirements are well-formed;
- The specification is well-formed;
- The specification behaves in accordance with the requirements;
- The design-by-contract rules for behavioural subtyping are obeyed;
- Type constraints are honoured;
- Variables are initialised before use;
- Each class invariant is established by each constructor of the class and is preserved by each member schema;
- Assertions and postassertions are satisfied;
- User-defined operators have the appropriate associativity, commutativity, symmetry and transitivity properties where the context so requires;
- Manual refinements are well-formed and are true refinements of the corresponding specifications;
- Loops are terminating;
- Directly recursive definitions are terminating (a thorough treatment of indirect recursion is not yet implemented).

By “well-formed” we mean that all expressions can be evaluated without violating preconditions. This encompasses not only operator preconditions (e.g. “array index within bounds” and “no division by zero”) but also covers the correctness of other constructs (e.g. a narrowing type cast has a precondition that the dynamic type of the expression is a subtype of the type to which it is cast).

In order to ensure correctness, the verification conditions must be shown to be true theorems. *Perfect Developer* includes a theorem prover for this purpose. Commercial software developers rarely have the skills needed to assist a theorem prover in developing proofs, and those that do have the skills cannot afford the time; so we chose to provide a non-interactive prover rather than an interactive one.

A combination of recent advances in automated reasoning technology, increasing processor power and careful language design has allowed us to approach our target of 100% automated proofs. The prover uses a combination of term rewriting and a superposition-based first-order theorem prover; the latest version uses backtrack-free splitting [13] to improve performance. Although the underlying logic of the notation is a many-sorted logic of partial functions, we are able to use conventional two-valued logic in most of the prover, because any verification condition that relies on partial functions is accompanied by another condition that ensures that the associated preconditions are satisfied.

5.2 Avoiding loops

Loops pose a particular problem for formal software verification, because neither a loop nor the code that follows it can be verified without knowledge of an adequate loop invariant. Although there has been some research on automated determination of loop invariants [14], it is at present impossible to avoid the need for users to declare loop invariants in more complex cases. The manual construction of correct loop invariants is difficult and time-consuming in all but the simplest cases.

Specifications in *Perfect Developer* are state-based and so do not contain loops; but loop statements can be introduced explicitly by the user within method refinements. The notation requires the user to declare a loop invariant. Fortunately, the provision of automated refinement by the tool and the presence in the notation of elements such as quantifiers and comprehension operators means that it is rarely necessary to write explicit loops. In practice, we find that of all the loops in the generated code, only about 7% correspond to loops explicitly introduced by the developer.

5.3 Automated refinement

Perfect Developer offers automated refinement, so that the code for many components can be generated directly from their state-based specifications. Manual refinement (with automated verification) is needed only where the automatic refinement fails to deliver an implementation that is efficient enough, or where the specification is written in an implicit style and *Perfect Developer* is unable to refine it to a more explicit form.

Automated refinement is currently implemented using a fixed set of rules together with a library of commonly-used classes and methods. In the future we intend to try a more intelligent approach that uses feedback from the verifier, similar to the proof planning mechanisms used in some theorem provers [15].

The tool generates code in two stages. First, the result of manual or automatic refinement is further refined to an internal notation. This notation is based on the standard *Perfect Developer* implementation notation, but removes constructs that are not readily translatable (e.g. quantifiers) while adding low-level constructs such as assignments. The precise subset of this notation that is used depends on the output language and compiler selected, reflecting the fact that some languages offer more facilities than others and some compilers have been found to generate incorrect object code for some constructions. We do not generate verification conditions for this refinement step, although it would be easy enough to do if this were required for a high-integrity application.

The output from this last refinement is translated into code. We rely on using simple rules for code generation rather than attempting to verify that the code is correct. Indeed, verification of the generated code is somewhat pointless in the absence of verified compilers, particularly as we have been informed by our contacts in safety-critical software development organisations that even "certified" compilers quite often exhibit code generation faults. We have given some thought to verifying the object code produced by the compiler directly against the final refinement.

6 An Example: Binary Search

Listing 1 illustrates the specification of a function to search an ordered table and its refinement to a classic binary search algorithm, in the notation of *Perfect Developer*. The **require** clause constrains the types with which the generic parameter X can be instantiated to those types having a total ordering operator '~~' (from which *Perfect Developer* automatically defines other comparison operators including '<' and '<='). The precondition (introduced by the **pre** keyword) states that *table* must be in nondecreasing order, using member function *isndec* of class **seq of** X to express this. The required function result is specified implicitly using the **satisfy** clause. The unary '#' operator denotes the length of its operand while the construct "**forall** *x*::*collection* :- *p(x)*" expresses universal quantification over the elements of *collection*. The binary operator '..' constructs a sequence of values in increasing order from its first to its second operand inclusive. Sequence indices range from zero to one less than the length of the sequence.

The specification is explicitly refined to an implementation, comprising three statements enclosed by **via** and **end**. The first statement declares and initialises a variable *low* that will in time contain the result. This is followed by a loop statement that includes a declaration of local variable *high*, three loop invariants, the termination condition, a loop variant (so that termination may be verified), and finally a loop body. The *let* statement in the body creates a local constant *mid*. The assertion that follows not only documents the developer's expectations but also introduces a lemma in the verification process, thereby easing the automated proof of some of the verification conditions. The keywords **if** and **fi** enclose guarded statement lists, with the empty guard '[ ]:' meaning "else". Finally, the **value** statement after the end of the loop defines the return value.

For this example, *Perfect Developer* generates and proves 27 verification conditions, providing assurance that the specification is well-formed and the implementation is correct.

```
// Find the index of the first element in "table" that is greater than or equal to "x"
function search(table: seq of class X, x: X): nat
  require X has total operator ~~(arg) end
  pre table.isndec
  satisfy result <= #table,
      forall z::0..(result - 1) :- x > table[z],
      forall z::result..(#table - 1) :- x <= table[z]
  via
    // Implement with a binary search
    var low: nat! = 0;
    loop
      var high: int! = #table;
      change low                          // loop frame
      keep                                // loop invariants...
        0 <= low' <= high' <= #table,
        forall z::0..(low'- 1) :- x > table[z],
        forall z::high'..(#table - 1) :- x <= table[z]
      until low' = high'                  // termination condition
      decrease high' - low';              // loop variant

      // Start of loop body
      let mid ^= (low + high)/2;
      assert low <= mid < high;
      if [x > table[mid]]:
        low! = mid + 1;
      []:
        high! = mid;
      fi
    end;
    value low;
  end;
```

Listing 1. Implementation of a binary search algorithm

7 Modelling at multiple levels

Software systems need to be modelled at various levels during development. In this section, we discuss some of these levels and how they are represented in *Perfect Developer*. Most of these sorts of model can be used to describe either a complete software system or a component of a larger system. We will illustrate these various models with a small example. Suppose that a spell-checking function is required for some application program, and that we are required to construct a class to fulfil the role of a dictionary of known words. We will assume the required operations are:

- Build a new, empty dictionary;
- *add* a new word to the dictionary;
- *remove* an existing word from the dictionary;
- *check* whether a proposed word is in the dictionary, returning **true** if it is and **false** otherwise.

We will impose the following requirements on these operations:

- By an empty dictionary, we mean a dictionary for which the *check* operation will return **false** for every word;
- The *add* operation causes a subsequent *check* for the word added to return **true**;
- The *remove* operation cause a subsequent *check* for the word removed to return **false**;
- Neither *add* nor *remove* affects the result of calling *check* for a word that is different from the one added or removed;
- If we use *add* to add a new word and then we immediately call *remove* with the same word, the dictionary is unchanged as far as the client is concerned.

The last of these requirements is redundant, but is a property that the client would expect to hold. Verification of redundant requirements increases confidence that the remaining requirements have been correctly expressed. To make the specification a little more interesting, we will also constrain the client to only invoke *add* with a word for which *check* returns **false**. Similarly, *remove* may only be called with a word for which *check* returns **true**.

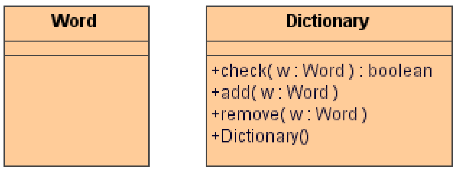

Fig. 1. UML class diagram for Dictionary

7.1 Structural model

This describes the overall structure of the software, the way it interacts with the outside world, and the interfaces to its operations.

The Unified Modeling Language (UML) does an acceptable job of representing this sort of model and we saw no reason to develop an alternative notation within *Perfect Developer*. Instead, we provide a mechanism for importing UML models, providing a structural skeleton in *Perfect Developer* notation on which detailed semantics can be hung.

A UML class diagram for our dictionary component might look like Fig. 1. If this is imported into *Perfect Developer*, the skeleton shown in Listing 2 is constructed, containing two class declarations. The Dictionary class

```
class Word ^= ?;

class Dictionary ^=
abstract
  ?
interface
  function check(w: Word): bool
    ^= ?;

  schema !add(w: Word)
    post ?;

  schema !remove(w: Word)
    post ?;

  build{}
    post ?;
end;
```

Listing 2. Structural model

declaration includes skeletal method declarations. The keyword **schema** introduces the declaration of a method that changes the state, while the keyword **build** introduces the declaration of a constructor for objects of the class. The '?' symbol indicates places where semantic detail needs to be added.

7.2 Behavioural model

This level corresponds to externally visible behaviour that the software is expected to exhibit, including safety properties and any conceivable tests that might be performed together with their expected outcomes.

Perfect Developer provides three main mechanisms for representing these behaviours. A **property** declaration is a parameterised theorem that expresses an expected truth. A *postassertion* is a convenient way of expressing a required behaviour when that behaviour relates mainly to a single method call. A *ghost schema* expresses a usage scenario (typically involving a sequence of method calls) and its postassertion expresses truths that should be established in the final state.

Listing 3 shows how the skeleton created from the UML model could be built upon to express some expected behaviours. The three postassertions correspond to expectations we have about the effect of calling the methods to which they are attached. These are: after adding a word using the *add* method, a call to *check* will assert that the word we added is now in the dictionary; similarly, after calling the *remove* method, a call to *check* reveals that the word removed is not in the dictionary; and the constructor does indeed build an empty dictionary in that for any word we choose, *check* does not report that word as being present. The use of the prime character in a postassertion indicates that we are referring to the final value of the preceding variable.

We have added a ghost schema that states that if a word not already present is added to the dictionary and then removed from it, then to the outside observer the dictionary is unchanged from its initial state. We have also included a property declaration expressing the notion that adding a new word to the dictionary does not affect the result of a query involving a different word.

7.3 State-based model

This is a more detailed model which provides an abstract data model for each class and specifications of the methods in terms of this data. If we choose to use a set of words as the abstract data model in our example, our state-based model might look like Listing 4.

We have defined the *Word* class as a constrained **string** (which is itself synonymous with the type **seq of char**). The postconditions of the *add* and *remove* schemas indicate the frame and the required state change.

At this point, the verifier can be used to check that the state-based model conforms to the behavioural model. *Perfect Developer* generates 7 verification conditions for this example; the verifier proves them all in less than a second.

```
class Word ^= ?;

class Dictionary ^=
abstract
  ?
interface
  // Check if a word is in the dictionary
  function check(w: Word): bool
    ^= ?;

  // Build an empty dictionary
  build{}
    post ?
    assert forall w:Word :-~self'.check(w);
        // the dictionary really is empty

  // Add a word to the dictionary
  schema !add(w: Word)
    pre ~check(w)
    post ?
    assert self'.check(w);
        // w is now in the dictionary

  // Remove a word from the dictionary
  schema !remove(w: Word)
    pre check(w)
    post ?
    assert ~self'.check(w);
        // w is no longer included

  // If we add a word and then remove it,
  // we get the dictionary we started with
  ghost schema addThenRemove(w: Word)
    pre ~check(w)
    post !add(w) then !remove(w)
    assert self' = self;

  // Adding a word does not affect whether
  // a different word is in the dictionary
  property(w1, w2: Word)
    pre ~check(w1), w1 ~= w2
    assert (self after it!add(w1))
                .check(w2) = check(w2);
end;
```

Listing 3. Behavioural model

```
class Word ^= those x: string :- ~x.empty;

class Dictionary ^=
abstract
  var words: set of Word;
interface
  // Check if a word is in the dictionary
  function check(w: Word): bool
    ^= w in words;

  // Build an empty dictionary
  build{}
    post words! = set of Word{}
    assert forall w:Word :-~self'.check(w);
        // the dictionary really is empty

  // Add a word to the dictionary
  schema !add(w: Word)
    pre ~check(w)
    post change words
        satisfy words'= words.append(w)
    assert self'.check(w);
        // w is now in the dictionary

  // Remove a word form the dictionary
  schema !remove(w: Word)
    pre check(w)
    post change words
        satisfy words'= words.remove(w)
    assert ~self'.check(w);
        // w is no longer included

  // ghost schemas and properties as before
  ...
end;
```

Listing 4. State-based model

7.4 Refinement

In many cases, *Perfect Developer* can generate ready-to-compile C++ or Java code of acceptable quality from the state-based model by automatic refinement. However, manual refinement is sometimes needed, typically in order to effect a more efficient representation of the data.

We will illustrate this by modifying our example to use a more compact representation for the dictionary. Observing that in the English language, the plural form of most nouns can be obtained by appending the letter 's' to the singular form, it seems pointless to represent a pair of words like "computer" and "computers" as two separate words. More efficient use of memory is achieved by storing just "computer" along with an indication that appending the letter 's' yields another valid word.

To distinguish between words for which it is known that appending 's' yields another word, and words for which this is not known, we will store the former in a set called *specialWords* and the latter in another set *plainWords*. Of course, from the point of view of the user of the dictionary, "computer" and "computers" are two separate words and will need to be added independently.

We can indicate a data refinement by adding an **internal** section to the class (Listing 5). The '++' operator indicates union when its operands are sets and concatenation when its operands are sequences The '**' operator indicates set intersection and the "**for**...**yield**" construct is a set comprehension. The construct "x ++ "s" **is** Word" is parsed as "(x ++ "s") **is** Word" and narrows the result of the '++' operation (which is of type

```
class Dictionary ^=
abstract
  var words: set of Word;
internal
  var plainWords,specialWords: set of Word;

  // Retrieve function
  function words ^=
    plainWords
     ++ specialWords
     ++ (for x::specialWords
           yield x ++ "s" is Word);

  invariant
   // No word is represented more than once
   plainWords ** specialWords
        = set of Word{},
   forall x::specialWords
        :- x ++ "s" ~in plainWords,
   forall x::specialWords
        :- x ++ "s" ~in specialWords,

   // A word and its plural should not both
   // be in plainWords, because storing the
   // word in specialWords would save space
   forall x::plainWords
        :- x ++ "s" ~in plainWords;

  interface
    ...
```

Listing 5. Data refinement

string) to class *Word* (which we declared as a subtype of **string**), so that the enclosing **for**...**yield** construct yields "**set of** *Word*" rather than "**set of string**". By redeclaring the abstract variable *words* as a retrieve function (also called an *abstraction function*), we indicate that the internal variables replace the abstract variable rather than supplement it.

Having refined the data, we also need to refine the class methods to use the new data. As in the previous example, this is accomplished by adding statement lists enclosed by **via**...**end**. The *check* function and the constructor are easily refined (Listing 6). Note that the state-based specification remains exactly as before; the **via**...**end** blocks do not change the specifications, but indicates how they are to be implemented. The *front* member function of the **seq of** ... class yields the sequence less the last element, while the *last* member function yields its final element.

Suitable refinements of the *add* and *remove* schemas are shown in Listing 7. We use a conditional statement to handle three separate cases that may arise. The semantics of this statement is "if..then...elseif...then...else...". The syntax "v! = e" is a shorthand for "**change** *v* **satisfy** *v'*= *e*" so that it has the effect of assignment. Although this is a postcondition in the notation of *Perfect*

```
  function check(w: Word): bool
    ^= w in words
    via
      value w in plainWords
          | w in specialWords
          |   #w >= 2
            & w.last = `s`
            & w.front in specialWords
    end;

  build{}
    post words! = set of Word{}
    via
      plainWords! = set of Word{},
      specialWords! = set of Word{}
    end
  assert forall x: Word :- ~self'.check(x);
```

Listing 6. Refinement of method *check* and constructor

Developer, the language syntax allows a postcondition to be used where a statement is expected, with the obvious meaning "satisfy this postcondition at this point!".

Having refined the data and the methods that operate on it, the verifier can be asked to prove that the refinements satisfy the specifications and preserve the invariants. For this example, *Perfect Developer* is able to prove all 64 verification conditions in less than one minute on a modern personal computer.

8 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 the commercial release of *Perfect Developer* in July 2002.

Aside from academic examples, *Perfect Developer* has been used to implement real-world systems including a terminal emulator, part of an IT system for a government department, and the *Perfect Developer* compiler/verifier itself. Some statistics from these projects are shown in Table 1. When comparing the number of lines of specification and refinement with the amount of generated code, bear in mind that the lines of specification and refinement include comment lines, whereas the generated C++ is uncommented except for a header comment. Also, long expressions and statements in the generated code are unformatted apart from line wrap at 120 characters. The uncertainty in the figures for the percentage of true theorems proven automatically for two of the projects reflects the fact that we have not manually investigated each verification failue to establish whether the condition concerned is a true theorem. The figures for time per verification condition attempted were obtained on a PC with AMD 2800+ processor and 512Mb memory.

```
schema !add(w: Word)
  pre ~check(w)
  post change words satisfy words'= words.append(w)
  via
    if [#w >= 2 & w.last = `s` & w.front in plainWords]:
      // word ends in 's' and root is in dictionary
      plainWords! = plainWords.remove(w.front),
      specialWords! = specialWords.append(w.front);

    [w ++ "s" in plainWords]:
      // the "plural" form of the word is already in the dictionary
      plainWords! = plainWords.remove(w ++ "s"),
      specialWords! = specialWords.append(w);

    []:
      // no related word is already in the dictionary
      plainWords! = plainWords.append(w);
    fi
  end
  assert self'.check(w);        // w is now in the dictionary

schema !remove(w: Word)
  pre check(w)
  post words! = words.remove(w)
  via
    if [w in plainWords]:
       // the simple case
      plainWords! = plainWords.remove(w);
    [w in specialWords]:
      // the "plural" form of w is also in the dictionary
      specialWords! = specialWords.remove(w);  // remove the word and its "plural"
      !add(w ++ "s");                          // add back the "plural"
    [#w >= 2 & w.last = `s` & w.front in specialWords]:
      // word ends in `s` and its "singlular" form is also in the dictionary
      specialWords! = specialWords.remove(w.front);
      !add(w.front);                           // add back the "singular"
    fi;
  end
  assert ~self'.check(w);       // w is no longer included
```

Listing 7. Refinement of methods *add* and *remove*

Project	Terminal emulator	Part of IT system	Compiler/verifier
Lines of specification and refinement	3192	13777	114720
Lines of generated C++	6752	38858	229367
Number of verification conditions	1349	2645	131441
% valid verification conditions proved	$\geq$ 98.6%	99.89%	$\approx$ 96%
Average time per verification condition	2.4 sec	3.7 sec	4.5 sec

Table 1. Three projects implemented in *Perfect Developer*

In all three 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 compiler/verifier uncovered a language design flaw that, for some inputs, would 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, *Perfect Developer* contains a compiler (which includes the refinement and code generation subsystems) and a verifier. 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 an hour or more 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 reasoning; nevertheless, the majority of class methods involved are not manually refined.

9 Limitations

One of the reasons behind the decision to implement *Perfect Developer* in its own technology was to ensure that the technology was scalable to large projects. In consequence, we revised the notation several times during development in response to issues that arose. We are now confident that the notation is sufficiently expressive to specify and implement a wide range of single-threaded software systems.

Although we have constructed multithreaded programs using *Perfect Developer*, the notation is currently unable to model concurrency, so desirable properties such as absence of deadlock cannot be verified or even expressed. Our main research is now directed towards adding concurrency to the notation and extending the verifier to prove concurrency properties.

Full verification of programs that use reference variables explicitly continues to be difficult because of the potential for aliasing. The problem is compounded when inheritance and dynamic binding are used because it becomes necessary to extend the notion of behavioural subtyping to include anti-aliasing invariants.

Code generation is currently restricted to C++ and Java. Ada 95 code generation is partially implemented, however the Ada "with-ing" problem [16] prevents the completion of this work pending a revision to the Ada language to address this limitation.

When the verifier fails to produce a proof of a verification condition, *Perfect Developer* analyses the failure and attempts to provide the user with helpful information, to assist in tracking down the likely error. Although the information produced is sometimes quite helpful, on other occasions it is of little use. It might be more helpful if the tool were to suggest fixes to the source. We note that some work has already been done in this field [17] in relation to ESC/Java.

10 Conclusions

We have shown that it is possible to develop a large, complex program in reasonable time using an object-oriented formal method; and that automated reasoning can now be used to successfully prove a very high proportion of the generated verification conditions.

The construction of programs that are proven to conform to formal specifications is of little value if the specifications do not fulfil user requirements. Some requirements are easy to capture and express as expected behaviours of the system and these can be formally verified, but other categories of requirements prove more elusive. We are working with others who specialize in formal requirements [18] to address this part of the development process.

References

1. G. Smith, *The Object-Z Specification Language* (Advances in Formal Methods Series, Kluwer Academic Publishers, 2000).
2. E.H. Durr and N. Plat (editor), *Afrodite (ESPRIT-III project number 6500) document AFRO/CG/ED/LR-M/V10* (Cap Volmac, 1995, in Dutch).
3. J. Spivey, *The Z Notation: a Reference Manual* (Prentice Hall, 1992).
4. C. B. Jones, *Systematic Software Development Using VDM* (Prentice Hall, Englewood Cliffs, NJ, USA, 1990).
5. J-R. Abrial, *The B-Book: Assigning Programs to Meanings* (Cambridge University Press, 1996).
6. J. Barnes, *High Integrity Ada: The SPARK Approach* (Addison-Wesley, Harlow, England 1997).
7. Barbara Liskov and Jeannette Wing, *A behavioral notion of subtyping*, ACM TOPLAS **16(6)** (1994) 1811-1841.
8. Bertrand Meyer, *Object-Oriented Software Construction* (Prentice Hall, Englewood Cliffs, NJ, USA, 1988).
9. D. Crocker, *Safe Object-Oriented Software: The Verified Design-by-Contract Paradigm.* In F. Redmill and T. Anderson, *Practical Elements of Safety: Proceedings of the Twelfth Safety-Critical Systems Symposium* (Springer-Verlag, London, 2003) 19-41.
10. K. Rustan, M. Leino, Greg Nelson and James B. Saxe, *ESC/Java User's Manual* (Technical Note 2000-002, Compaq Systems Research Center, 2000). Available via `http://research.compaq.com/SRC/esc/`.
11. *Perfect Developer Language Reference Manual* (Escher Technologies Limited, 2002). Available at `http://www.eschertech.com/product_documentation/LanguageReferenceManual.htm`.

12. N. Evans and D. Crocker, *Proof Obligations Generated by Perfect Developer 2.10* (Escher Technologies Limited, 2003). Available at `http://www.eschertech.com/product_documentation/ProofObligations.pdf`.
13. A. Riazanov and A. Voronkov, *Splitting without Backtracking*, Preprint CSPP-10, University of Manchester (2000).
14. J. Stark and A. Ireland, *Invariant Discovery via Failed Proof Attempts.* In *Proceedings of the 8th International Workshop on Logic Programming Synthesis and Transformation* (LNCS 1559, Springer-Verlag, 1998) 271-288.
15. A. Bundy, *The use of explicit plans to guide proofs.* In *Proceedings of CADE-9* (LNCS 310, Springer-Verlag 1998) 111-120.
16. J. Volan, *John Volan's answers to Frequently Asked Questions about the Ada "with-ing" Problem* (1997). Available at `http://www.eschertech.com/WithingProblem.htm`.
17. Cormac Flanagan, K. Rustan and M. Leino, *Houdini, an annotation assistant for ESC/Java* (Technical Note 2000-003, Compaq Systems Research Center, 2000). See `http://research.compaq.com/SRC/esc/relatedTools.html`.
18. J. Warren and R. Oldman, *A Rigorous Specification Technique for High Quality Software.* In F. Redmill and T. Anderson, *Practical Elements of Safety: Proceedings of the Twelfth Safety-Critical Systems Symposium* (Springer-Verlag, London 2003) 43-65.