Perfect Developer: what it is and what it does

by David Crocker and Judith Carlton, Escher Technologies Ltd.

Perfect Developer (a.k.a. the *Escher Tool*) is a formal tool aimed at software development but with applications in the formal specification of other sorts of system. It is built around a notation for expressing state-based specifications and optionally refining them to a form resembling a program in an imperative programming language. In this sense, it is rather like the B method, or the combination of VDM-SL and VDM-IL. However, being a relatively recent entry to the field, it is designed around two technologies that matured long after VDM and B were designed:

- Object oriented (O-O) and component-based design
- Automated reasoning

We based *Perfect Developer* around object-oriented design because that is the dominant paradigm used in industry today, but we recognise that not all problems benefit from an O-O approach. Furthermore, some features of O-O design are not yet accepted as safe by the developers of safety-critical systems. So while *Perfect Developer* does require use of two of the foundations of O-O design (abstraction and encapsulation), use of polymorphism and dynamic binding is discretionary and the use of objects obeying reference semantics is discouraged.

Thus, *Perfect Developer* is based on the paradigm of classes that encapsulate data and methods that operate on that data, in the same way that B is based on the paradigm of abstract machines.

Another advantage of using the O-O paradigm is that *Perfect Developer* can import UML models to generate skeleton specifications, on which detailed semantics can be hung. It can also generate ready-to-compile code in C++ or Java, which can be interfaced to graphical user interfaces or to other components written in those languages.

In general, just the process of writing a formal specification is likely to improve the quality of a program written from it. Clearly, actually proving that the specification is consistent, and that it's correctly implemented, is of greater value. From the commercial point of view, though, producing proofs by hand is far too time-consuming. Even if an interactive theorem prover assists the user in constructing the proofs, the process is nowhere near fast enough for widespread commercial use.

The second major technology underlying *Perfect Developer* is automated reasoning, and this helps solve the difficulties with commercial productivity.

Automated reasoning technology has advanced in leaps and bounds during the last decade – particularly in the field of first-order theorem proving. We therefore designed the notation of *Perfect Developer* to give rise to verification conditions (or *proof obligations*) that are overwhelmingly first-order. Then we built a theorem prover, optimising it to handle real-life verification conditions, rather than the abstract mathematical theorems for which academic provers are designed. The result is that the tool is able to discharge more than 95% of valid verification conditions without user intervention in typical commercial applications - one real-life system has recently reached 99.89%.

Listing 1: Specification of a bounded queue

```
final class Queue of X ^=
abstract
  var b: seq of X,       // the queue data
      maxLen: nat > 0;  // maximum items in the queue

  invariant #b <= maxLen;

interface
  function empty: bool   // test if the queue is empty
    ^= #b = 0;

  schema !add(x: X)      // add an element to the end of the queue
    pre ~full
    post b!= b.append(x);

  function full: bool     // test if the queue is full
    ^= #b = maxLen;

  schema !remove(x!: out X)  // remove the head element
    pre ~empty
    post x! = b.head,
         b! = b.tail;

  build{!maxLen: nat, dummy: X}  // build an empty queue
    pre maxLen ~= 0
    post b! = seq of X{};

  ghost operator =(arg);  // we do not evaluate equality at run-time

  // Verify that after adding an element, a queue is not empty
  property (x: X)
    pre ~full
    assert ~(self after it!add(x)).empty;

  // Verify that if we add an element to an empty queue,
  // the next element we remove will be the one we added
  ghost schema !addToEmptyThenRemove(e: X, r!: out X)
    pre empty
    post !add(e) then !remove(r!)
    assert r' = e;

end;
```

We tried to make the notation of *Perfect Developer* easy for software developers to learn, including those unused to formal methods or mathematical notation. Users who are already familiar with VDM or B should find it even easier. Those who are more used to Z need to get used to separating pre- and post-conditions.

Further information about *Perfect Developer* can be found in [1], [2], [3] and [4].

An example: specifying and refining a queue

Listing 1 shows a small example in which a bounded queue is specified as a *Perfect* class called *Queue* **of** *X*. The generic parameter *X* represents the type of element that will be stored. The **abstract** section of the class declares the abstract model of the data held by the queue, which in this case comprises a sequence of elements *b* and a

fixed bound *maxlen*. The number of elements in *b* at any time cannot exceed *maxlen* and we declare this property as an **invariant** of the class (the unary # operator applied to a sequence yields its length).

The interface section contains the declarations of operations available to users of a *Queue*. In this example we declare query functions *empty* and *full*, together with operations *add* and *remove*, and a constructor **build** for creating an empty queue. The symbol ^= used in the function declarations means "is defined as". The keyword **pre** introduces a precondition, while **post** declares a schema postcondition. In *Perfect*, a postcondition either implicitly or explicitly includes a frame, thereby defining not only how the final values of changed variables relate to the initial conditions, but also requiring that other variables remain unchanged. For example, the assignment-like postcondition *b! = b.append(x)* is actually short for **change** *b* **satisfy** *b' = b.append(x)* which states that the only variable affected is *b* and that its final value *b'* must be equal to *b.append(x)*. The *append* function is a predefined method of class **seq of** *X* and yields a new sequence comprising the original with the parameter appended.

In order to improve confidence in the specification, we can also declare behavioural properties that we expect to hold. In this example we have declared some expected behaviour by declaring a **property** and a **ghost schema**. The property declaration asserts that immediately after calling the *add* method of a queue, the empty function should return *false*. The ghost schema describes the scenario of adding an element to an empty queue and then removing an element, and asserts that the element removed should be equal to the element added.

When asked to verify this specification, *Perfect Developer* generates and proves 16 verification conditions, assuring us that the specification is well-formed and consistent and that it exhibits the expected behaviour.

Although the specification in Listing 1 can be used to generate code directly, in practice it is more efficient to implement a bounded queue using a ring buffer. Listing 2 shows the same specification with refinement from the abstract model to an array *ring* together with head and tail indices *hd* and *tl*. The data refinement is declared in the *internal* section, together with the invariants that the sequence *ring* has fixed length and the two index variables are in range. By redeclaring the original abstract sequence *b* as a retrieve function, we indicate that it is not a stored variable in the implementation and we describe the value of *b* that is represented by any combination of values of *ring*, *hd* and *tl* that satisfy the invariant. In defining the retrieve function, we use a conditional expression, which has the form *([guard1]: expression1, [guard2]: expression2)* and has the meaning "if *guard1* then *expression1* else if *guard2* then *expression2*". The member function *take(n)* of class **seq of** *X* returns the first *n* elements of the sequence, while *drop(n)* returns all but the first *n* elements. The operator ++ applied to sequences denotes concatenation.

Alongside this data refinement, the specifications of the public operations are refined to implementations in the **via**…**end** blocks. The implementations declare how the corresponding specifications should be implemented as operations on the ring buffer and associated head and tail variables.

The refinement of the specification to a ring buffer implementation causes *Perfect Developer* to generate and prove an additional 34 verification conditions, which taken together show that the implementation is well-formed and faithfully implements the original specification.

Listing 2: Implementation of the queue using a ring buffer

```
final class Queue of X ^=
abstract
  var b: seq of X,       // the queue data
      maxLen: nat > 0;   // maximum items in the queue

  invariant #b <= maxLen;

internal
  var ring: seq of X,    // implement internally as a ring buffer
      hd, tl: nat;       // indices of the first and last elements

  invariant #ring = maxLen + 1,
        hd < #ring,
        tl < #ring;

  function b ^=          // retrieve function for variable 'b'
    (  [tl >= hd]:  ring.take(tl).drop(hd),
       [tl < hd]:   ring.drop(hd) ++ ring.take(tl)
    );
interface
  function empty: bool  // test if the queue is empty
    ^= #b = 0
    via
      value hd = tl
    end;

  schema !add(x: X)     // add an element to the end of the queue
    pre ~full
    post b! = b.append(x)
    via
      ring[tl]! = x,  tl! = (tl + 1)%(#ring)
    end;

  function full: bool   // test if the queue is full
    ^= #b = maxLen
    via
      value (tl + 1)%(#ring) = hd
    end;

  schema !remove(x!: out X)  // remove the head element
    pre ~empty
    post x! = b.head, b! = b.tail
    via
      x! = ring[hd],  hd! = (hd + 1)%(#ring)
    end;

  build{!maxLen: nat, dummy: X}  // build an empty queue
    pre maxLen ~= 0
    post b! = seq of X{}
    via
      ring! = seq of X{dummy}.rep(maxLen + 1),
      hd! = 0, tl! = 0
    end;

  // Include property and ghost schemas here as before…

end;
```

Verifying security properties of the Mondex Abstract World

At the recent Refinement Workshop, proof of the Z specification of the Mondex electronic purse [5] was discussed. As an exercise, a reformulation of the top level of this specification provided by Jim Woodcock was translated into *Perfect* and proven automatically.

Listing 3 shows a revised version of this translation in which we have tried to mirror the Z original more closely. We declare classes to represent the contents of a purse, the details of a transfer, and the abstract world itself. As in the Z version, the collection of authorised purses is represented as a mapping from the names of purses to their contents. We have declared separate schemas *AbTransferOkay*, *AbIgnore* and *AbTransferLost* in the abstract world to represent each of the three possible outcomes of attempting a transfer between purses (i.e. the transfer may succeed, or be ignored, or the amount may be lost).

A transfer attempt is represented by schema *AbTransfer* and its outcome is a nondeterministic choice between the other three schemas. The Z specification uses the schema disjunction operator to express this choice, but since in *Perfect* it is necessary to respect the schema preconditions, we use a conditional postcondition to select which of the three schemas may be invoked. The fact that *AbTransfer* is intentionally nondeterministic is flagged by declaring it **opaque**, and we again use the keyword **opaque** within the conditional postcondition, to indicate nondeterministic choice between those schemas whose guards are true, rather than deterministically choosing the first one whose guard is satisfied. The two security properties are expressed as post-assertions attached to schema *AbTransfer*.

Of the 30 verification conditions generated and proved by *Perfect Developer* for this example, two represent the security properties; the remainder are precondition checks and domain checks.

Listing 3: Abstract specification of Mondex electronic purse

```
// Declare a type for identifying purses
class NAME ^= tag;    // this creates a new abstract type called NAME

// Class to represent a Mondex purse
class AbPurse ^=
abstract
  var balance, lost: nat;
interface
  function balance, lost;    // this makes 'balance' and 'lost'
                             // readable from outside the class

  // Schema to represent an amount being lost from the purse
  schema !lose(amt: nat)
    pre amt <= balance
    post balance!- amt, lost!+ amt;

  // Schema to represent an amount being removed from the balance
  schema !remove(amt: nat)
    pre amt <= balance
    post balance!- amt;
```

Listing 3 (continued)

```
// Schema to represent an amount being added to the balance
  schema !add(amt: nat)
    post balance!+ amt;

  // Constructor
  build{}
    post balance! = 0, lost! = 0;
end;

// Class to represent details of a proposed transfer between purses
class TransferDetails ^=
abstract
  var frm, to: NAME,          // the 'from' and 'to' purses
      val: nat;               // the amount of the transfer
interface
  function frm, to, val;

  // Constructor
  build{!frm, !to: NAME, !val: nat};
end;

// Class to represent the abstract Mondex world
class AbWorld ^=
abstract
  var AbAuthPurse: map of (NAME -> AbPurse); // the authorised purses

  // Get the total balance of all authorised purses
  function totalAbBalance: int
    ^= + over (for x::AbAuthPurse.ranb yield x.balance);

  // Get the total lost from all authorised purses
  function totalAbLost: int
    ^= + over (for x::AbAuthPurse.ranb yield x.lost);

  // Determine whether a purse name is authentic
  function Authentic(id: NAME): bool
    ^= id in AbAuthPurse;

  // Determine whether the 'from' purse in a proposed transfer
  // has sufficient funds
  function SufficientFundsProperty(details: TransferDetails): bool
    pre Authentic(details.frm)
        ^= details.val <= AbAuthPurse[details.frm].balance;

  // Schema to represent a transfer attempt that is ignored
  schema !AbIgnore(details: TransferDetails)
    post pass;

  // Schema to represent a transfer attempt that results
  // in the amount being lost
  schema !AbTransferLost(details: TransferDetails)
    pre Authentic(details.frm), Authentic(details.to),
      details.frm ~= details.to,
      SufficientFundsProperty(details)
    post change AbAuthPurse
      satisfy AbAuthPurse' =
        AbAuthPurse.replace(details.frm -> AbAuthPurse[details.frm]
                                          after it!lose(details.val));
```

Listing 3 (continued)

```
// Schema to represent a successful transfer
  schema !AbTransferOkay(details: TransferDetails)
    pre Authentic(details.frm),
      Authentic(details.to),
      details.frm ~= details.to,
      SufficientFundsProperty(details)
    post change AbAuthPurse
      satisfy AbAuthPurse' =
        AbAuthPurse.replace(details.frm -> AbAuthPurse[details.frm]
                                        after it!remove(details.val))
                   .replace(details.to -> AbAuthPurse[details.to]
                                        after it!add(details.val));
interface

  // Schema to represent a transfer attempt that may be successful,
  // ignored, or result in the amount concerned being lost
  opaque schema !AbTransfer(details: TransferDetails)
    post
    ( opaque  // use nondeterministic guarded choice here to mimic
              // the Z schema disjunction operator
      [  Authentic(details.frm)
       & Authentic(details.to)
       & details.frm ~= details.to
       & SufficientFundsProperty(details)
      ]:
        !AbTransferLost(details),
      [  Authentic(details.frm)
       & Authentic(details.to)
       & details.frm ~= details.to
       & SufficientFundsProperty(details)
      ]:
        !AbTransferOkay(details),
      [true]:
        !AbIgnore(details)
    )
    // declare security properties
    assert self'.totalAbBalance <= totalAbBalance,
                  // no value created (Z: NoValueCreation)
           self'.totalAbBalance + self'.totalAbLost
               = totalAbBalance + totalAbLost;
                  // all value accounted (Z: AllValueAccounted)

  // Constructor
  build{!AbAuthPurse: map of (NAME -> AbPurse)};

end;
```

Obtaining Perfect Developer

Perfect Developer is free to evaluate, including use in small-scale student projects. Over 20 universities are using the tool in one way or another and six have purchased licences for classroom teaching or for research. For more information, please email info@eschertech.com.

References

1. *Perfect Developer Language Reference Manual*. Available at http://www.eschertech.com/product_documentation/Language%20Reference/language_reference.pdf, or online in HTML format via the *Support* section of www.eschertech.com.
2. *Proof Obligations Generated by Perfect Developer*. Available at http://www.eschertech.com/product_documentation/ProofObligations.pdf.
3. D. Crocker, *Safe Object-Oriented Software: the Verified Design-by-Contract paradigm*. Proceedings of the Twelfth Safety-Critical Systems Symposium (ed. F.Redmill & T.Anderson) 19-41, Springer-Verlag, London, 2004.
4. D. Crocker and J. Carlton, *A High Productivity Tool for Formally Verified Software Development*. To be published in the International Journal on Software Tools for Technology Transfer (Special Section on FME2003).
5. S. Stepney, D. Cooper and J. Woodcock, *An Electronic Purse*. Technical Monograph PRG-126, Oxford University Computing Laboratory (July 2000).

End