# Software Refinement with Perfect Developer

Gareth Carter
Department of Computer Science,
National University of Ireland, Maynooth
gcarter@cs.nuim.ie

Rosemary Monahan
Department of Computer Science,
National University of Ireland, Maynooth
rosemary.monahan@nuim.ie

Joseph M. Morris
School of Computer Applications,
Dublin City University
Joseph.Morris@computing.dcu.ie

[1]

## Abstract

*Perfect Developer is a software tool that supports the formal development of object-oriented programs by refinement, including formal verification of code. It is built around a single language that supports both specification and implementation. We critically examine how Perfect Developer supports programming by refinement, focusing on three refinement techniques: algorithm refinement, data refinement and delta refinement. In particular we examine the extent to which Perfect Developer provides formal verification for these techniques. We assess it as a tool for software construction and compare it with related tools.*

## 1. Introduction

Perfect Developer [8] is a software tool that supports the formal development of object-oriented programs by refinement [5]. By *refinement* is meant a style of development that leads from a formal specification to a working implementation in a sequence of correctness-preserving steps. Among the many tools that support formal programming, Perfect Developer is interesting because it aims to offer a unique combination of attributes: object-orientation, full verification, and support for refinement. In this paper we critically evaluate the extent to which Perfect Developer achieves these goals, concentrating on refinement. We focus on three refinement techniques: *algorithm refinement*, *data refinement* and *delta refinement*. The first two of these are well described in the literature [18] [9]; the third will be explained later.

To keep the paper self-contained, we will begin with a brief overview of Perfect Developer and its associated specification/programming language. At the end, we will compare it with similar software tools, and draw some conclusions. All the specifications and examples used in the paper, along with other supporting material, are available on an accompanying web site; see `http://www.cs.nuim.ie/toolap/pd/`.

## 2. Perfect Developer

Perfect Developer (PD) was designed to be a high-productivity tool for developing formal specifications and refining them to code [8]. It is available from Escher Technologies in three versions, an education, a professional and a safety-critical version, all running under Windows XP, Windows 2000, Windows NT4 or Linux.

Our experience of PD is primarily experimental and educational, using version 2.10 running under Windows 2000. Educationally we have used it on degree-level courses in software engineering at two universities (National University of Ireland Maynooth and Griffith College Dublin). Escher Technologies report an industrial use on a project of several hundred thousand lines of source code [1].

For our purposes, PD consists of three components: the Perfect Language (PL), the verifier, and the compiler. PL is an object-oriented specification/programming language. More precisely, it is a specification language with an implementable subset identified as its programming language. The verifier is a custom-built theorem prover that collects and attempts to discharge proof obligations for the software it is presented with. The compiler accepts code written in the programming language and compiles it into equivalent Java, C++ or Ada95 code. Third party editors and UML modeling tools can be integrated into PD.

[1]preprint of paper to appear in Proc. SEFM 2005, ©IEEE.

The verifier can be used to test run specifications, even those that are not programs. The trick here is to accompany the specification $S$ with an assertion $P$ where $P$ in effect states that $S$ delivers a certain result for a certain input. If $S$ and $P$ are together presented to the verifier, the verification of $P$ in effect is a test run of $S$. It is not always possible to play this trick — the verifier will sometimes find it too much effort — but when it works it is of some value in testing specifications before committing to code.

In classical refinement theory [5, 19] there are typically many steps on the path from original specification to final implementation. Perfect Developer, however, does not support small refinement steps. In particular, data refinement of a class must proceed in a single step from an abstract specification to an executable concrete implementation.

## 3. Perfect Language

PL encourages a Z-like model-oriented specification style. Like Z [23], it provides a library of useful collection and structure types such as sets, bags, sequences, and maps. Like Java, it supports encapsulation via classes, message passing, and inheritance. Methods are specified in the style of *design by contract* [17]. We illustrate its main features with the following very trivial example.

```
class A ^=          // Empty Superclass
interface
// Constructor
 build{}
end;

class B ^= inherits A
abstract            // Specification
// Variables
 var x:int;

// Restrictions
 invariant 0<=x<10;

internal            // Implementation
 var y:int;
 invariant 0<=y<10;

// Retrieve function
 function x
   ^= y;

interface           // Public methods

// Constructor
 build{} inherits A{}
   post x!=0
   via y!=0
   end;

// Equality definition
 operator =(arg);

// Getter Function
 function x;

// Evaluation Function
 function q(r:int): bool
   ^= r<10;

// A Function with DbC specs
 function g: int
  pre x<9
   ^= x+1
   via value y+1;
   end
  assert q(result);

// An implemented Predicate Function
 function h: int
   satisfy result>=x
   via value 2*y;
   end;

// A schema
 schema !k
  pre x<9
   post x!=g
   via y!=g;
   end
  assert  q(x');
end;
```

In the example, class `B` inherits all the features of class `A`. Inheritance is introduced by the keyword `inherits`. The keyword must be used again in the constructor of class `B` to invoke a call to the constructor of class `A`. Constructors are introduced by the keyword `build`.

As illustrated by class `B`, classes typically have three main sections: `abstract`, `internal` and `interface`. The `abstract` section introduces the abstract (or model) data by which the specification will be described. The `internal` section introduces the concrete data that will be used by the executable code. It also describes the relation between the abstract and concrete data by means of a *retrieve function* from concrete to abstract values (PL does not provide the option of a function from abstract to concrete values). The `interface` section defines the methods of the class.

The abstract data of class `B` is a single variable `x`, introduced by the keyword `var`. The value of `x` is restricted

to the range 0 to 9, inclusive, as indicated by the boolean term following the keyword `invariant`. The boolean must evaluate to `true` whenever an object of class `B` is constructed, and whenever a method of the class is called or terminates. The concrete data is in this case (very trivially) another integer variable, `y`.

There are three kinds of methods: *constructors*, *side-effect free methods*, and *state-changing methods*. Constructors instantiate objects of the class and initialize the instance variables. Side-effect free methods are functions that return a result without changing the global state. The state-changing commands, also called *schemas*, change the state without returning a result. Exclamation marks are used liberally in PL to draw attention to state changes. For example, they occur in the name of the schema (`!k`) and in the references to `x` and `y` in the `post` clause. All methods may be specified using design by contract: the keyword `pre` introduces the pre-condition of the method and the keyword `assert` introduces the post-assertion (more traditionally referred to as the "post-condition"'). The interface of class `B` also includes an equality operator; this applies to concrete data and must be included when refinement occurs.

There are in turn three varieties of functions in PL: *getter functions*, *evaluation functions*, and *predicate functions*. Getter functions return the value of a class attribute; they have the same name as the attribute and have no body. Evaluation functions are like functions found in functional programming languages: the body consists of a term (introduced by `^=`, read "is-defined-as") whose evaluation yields the result. A predicate function defines its result as a value that satisfies a given boolean term. In the example, the result of invoking function `h` is defined to be some number greater than `x` (note that the result need not be uniquely specified). Specifications may include primed versions of variables; the primed version denotes the value of the variable after the action being specified has completed.

Many methods are not executable, including all predicate functions and all functions specified using abstract data. Before compilation is possible, functions must be made executable by writing code in the body (introduced by the keyword `via`). Executable code may be expressed using concrete data only; no reference to abstract data is allowed. Note that specification and implementation are kept together as a single entity.

Loops in PL have a complex syntax. The general form is as follows:

```
loop
 var        // local loop variables
 change     // other variables
            // changed by loop
 keep       // loop invariant
 until      // end condition
 decrease   // loop variant
            // the loop body
 end;
```

We will see a concrete example shortly. Conditionals in PL come in two forms. The conditional statement "if condition C1 then statement S1 otherwise S2" is written as `if [C1]:S1 []:S2 fi`, while the conditional term "if condition C1 then term t1 otherwise t2" is written as `[C1]:t1, []:t2`. A more complete description of the language can be found in the Perfect Reference Manual [3].

## 4. Algorithm Refinement

Algorithm refinement as provided by PD allows either an entire method or a fragment within it to be refined to an appropriate implementation. The following sort program illustrates this.

```
function sort(ip:seq of int):seq of int
  satisfy isPermutation(result, ip),
          isSorted(result);
```

(The comma in the `satisfy` clause denotes logical conjunction). The function `isPermutation` is defined as

```
function isPermutation(a:seq of int,
                       b:seq of int):bool
   ^= a.ranb=b.ranb;
```

(`a.ranb` denotes the bag of elements in sequence `a`). Function `isSorted` is defined as

```
function isSorted(a:seq of int):bool
 decrease #a
  ^=([a.empty]:
      true,
     []:
      (forall e::a.tail :- a.head >= e)
         & isSorted(a.tail)
     );
```

(`head` and `tail` have their usual meanings: the head of a sequence is the first element of the sequence, and the tail is the sequence with the head omitted. Verification of a recursive function requires the inclusion of a variant term introduced by the keyword `decrease`. `#a` denotes the number of elements in sequence `a`.) The `sort` specification may be refined into an efficient *insertion sort* implementation, as follows:

```
function sort(ip:seq of int):seq of int
  satisfy isPermutation(result, ip),
          isSorted(result)
  via
   var op:seq of int!=seq of int{};
```

```
  loop
   change op

   keep 0<=#op'<=#ip,
     isPermutation(op',ip.take(#op')),
     isSorted(op')
   until #op'=#ip
   decrease #ip-#op';
    op!=addInPlace(op,ip[#op]);
  end;
  value op;
 end;
```

The helper function `addInPlace` inserts an element into a sequence at an indicated position:

```
function addInPlace(a:seq of int,
                    b:int):seq of int
 pre isSorted(a)
 decrease #a
  ^=([a.empty]:
  // Insert item if an empty sequence
     a.prepend(b),
    [b > a.head]:
  // Recurse if not in correct position
     addInPlace(a.tail,b)
                     .prepend(a.head),
    []:
  // Insert in the correct position
     a.prepend(b)
   )
 assert isSorted(result),
    isPermutation(result,a.prepend(b));
```

(The function `a.prepend(b)` prepends item `b` to sequence `a`.) When we first refined `sort`, we omitted the post-assertion in `addInPlace` (introduced by the keyword `assert`). It plays no role in the implementation, and there is no a priori reason to believe it could play a role in the verification. However, it had to be included to assist verification.

PD's support for algorithm refinement is not total. The step from specification to implementation is often too great a jump for the verifier, and the developer has to provide hints in the form of additional assertions as in the preceding example. Discovering the appropriate hints is a trial-and-error process that is pretty time-consuming. The effort required is exacerbated by the fact that it is not possible to work on selected pieces of the code in relative isolation, because the verifier treats the specification as a monolithic entity. On the other hand, PL provides a rich assertion language in which to write hints. We found that most of the examples we explored could eventually be verified, although we sometimes had to re-work the code in addition to providing hints.

# 5. Data Refinement

Data refinement arises when we decide on a change of data representation, and want to derive the new version of the specification/program from the old one. The change of data representation can arise for a variety of reasons, and we focus on just two, each illustrated with an example. In the first example, we illustrate the introduction of extra variables to improve efficiency (*attribute introduction*). The second example illustrates the classical case of employing one kind of data to facilitate the specification process, and replacing it with a more concrete kind to allow implementation (*type transformation*).

## 5.1 Attribute Introduction

Attribute introduction pertains to the introduction of auxiliary data to reduce computational cost, for example by introducing a variable to store the result of an evaluation of a term that would otherwise be evaluated several times. Consider a class that maintains a collection of numbers, and provides a method (among others) that returns the sum of the elements in the collection:

```
class ListOfNumbers ^=
abstract
 var list: seq of int;

nonmember function
                  sum(s:seq of int):int
  decrease #s
   ^= ([s.empty]: 0,
       []: s.head + sum(s.tail)
      );

interface
 build{}
   post list!= seq of int{};

 schema !addNumber(n: int)
   post list!= list.prepend(n);

 function getSum: int
   ^= sum(list);
end;
```

(A `nonmember` function is a static or class function.) Calls to function `getSum` have time complexity O(n). We can do much better by introducing a `sumOfList` attribute to keep track of the sum of the elements in the list:

```
internal
 var sumOfList: int;
 invariant sumOfList = sum(list);
```

The methods of the class are refined accordingly:

```
interface
 build{}
   post list!= seq of int{}
   via
    list!= seq of int{};
    sumOfList!= 0;
   end;

 schema !addNumber(n: int)
   post list!= list.prepend(n)
   via
    list!= list.prepend(n);
    sumOfList!= n+sumOfList;
   end;

 function getSum: int
   ^= sum(list)
   via
    value sumOfList;
   end;
end;
```

There are many alternatives to the above coding, some of which we tried and not all of which were successful. For example, the collection might have been modelled as a bag rather than a sequence (assuming none of the omitted operations relied on any positioning of numbers in the collection). The `addNumber` schema might `append` elements to the sequence rather than `prepend` them. Or the `sum` function might be discarded altogether by employing PL's built-in mechanism ("+ `over`") for summing the elements of a list. We tried all of these, but the verifier failed in each case, and we were led to the solution presented above.

Suppose in the above example that there were no methods other than those explicitly written. In that case the `list` attribute is redundant and we might feel that we may as well remove it. Unfortunately, PD insists that it be retained if the verification is to succeed.

PL has a rich language that would appear to support various specification styles. For example, it is possible to employ a more algebraic style of specifying (making correspondingly less use of the model-oriented style). However, we found that the verifier is at its best when the coding style is not too distant from the specification style. We did not manage to codify a universal style preferred by the verifier. Rather, it seemed that each new problem requires some experimentation to find a good pairing of specification and implementation. However, one can build up experience in each problem area. When verification is unsuccessful it is usually unclear whether the fault lies with the developer or the theorem prover. As a consequence we abandoned some refinements whose correctness we were confident of (we reasoneed very carefully about them) because the verifier could not deal with them.

### 5.2. Type Transformation

As an example of type transformation, we define a rather simple set data type with operations `insert` and `remove`, and implement it using PL's `seq` data structure.

```
class SetSeq ^=
abstract
 var setofNums: set of int;

internal
 var seqOfNums:seq of int;
 function setofNums
     ^= seqOfNums.ran;
```

(`seqOfNums.ran` in the retrieve function yields the set of elements in sequence `seqOfNums`.) The example is trivial in that it uses the `set` type provided by PL, but it nevertheless suffices to illustrate our points. The interface of the class is specified and implemented as follows:

```
interface

// Equality definition
 operator =(arg);

 build{}
   post setofNums!=set of int{}
   via  seqOfNums!=seq of int{}
   end;

 schema !insert(a:int)
   post setofNums!=setofNums.append(a)
   via  seqOfNums!=seqOfNums.append(a)
   end;

 schema !remove(a:int)
   post setofNums!=setofNums.remove(a)
   via
    loop
     var i:nat!=0;
     change   seqOfNums

     keep i'<=#seqOfNums',
        a ~in seqOfNums'.take(i'),
        seqOfNums'.ran<<=seqOfNums.ran,
        seqOfNums.ran=seqOfNums'.ran |
        seqOfNums.ran =
               seqOfNums'.ran.append(a)
     until    i'=#seqOfNums'
     decrease #seqOfNums' - i';
```

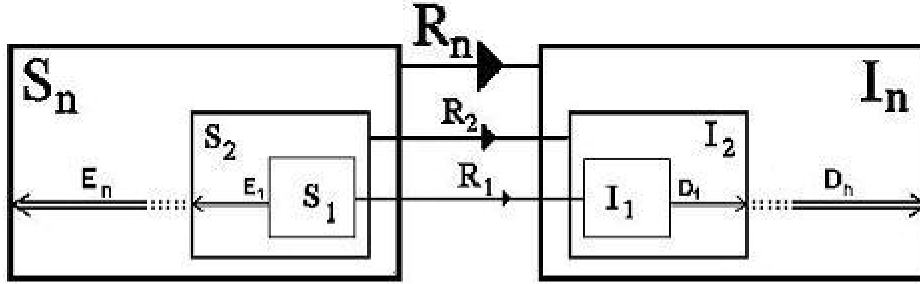


**Figure 1. Refinement of S to I**

```
      if
       [a=seqOfNums[i]]:
        seqOfNums!=seqOfNums.remove(i);
       []:
        i!=i+1;
      fi;
    end;
   end;
```

(~ and | denote logical negation and disjunction, respectively, and <<= denotes the subset relation. `s.append(a)` inserts item `a` to set `s`, or adds item `a` to the end of a sequence `s` depending on the type of `s`.) The `remove` schema is complex because it is necessary to remove *all* occurrences of the item being deleted, and that requires an iteration over the sequence. We also made attempts to maintain an injective (i.e. duplicate-free) version of `seqOfNums` but we were let down by the verifier. The loop invariant (following the keyword `keep`) is almost a word-for-word description of set removal. Although formally a weaker predicate suffices to verify the loop, the verifier needs the version presented. As mentioned earlier, the relation between abstract and concrete data must be written as a function from concrete to abstract data; it is not possible to use a relation, or a function from abstract to concrete data.

## 6. Delta Refinement

Sometimes after specifying and refining a component we may find we have use for an enhanced version. For example, in object-oriented programming a component might be described by a class which we might subsequently enhance by inheritance (which can entail some combination of extension, overriding and descendant hiding). It is clearly desirable that we should not have to refine the enhanced specification from scratch, but rather should be able to derive the new implementation by an appropriately modest re-working of the original. We refer to this as *delta refinement*.

In Figure 1, an initial specification $S_1$ is refined by technique $R_1$ into an implementation $I_1$. Suppose now that specification $S_1$ is enhanced by some procedure $E_1$ resulting in specification $S_2$. $I_2$ represents a candidate implementation of $S_2$ obtained by the incremental refinement $D_1$ corresponding to the enhancement $E_1$. Let $R_2$ denote the total refinement represented by $R_1$ plus $D_1$. If $R_2$ does indeed represent a refinement of $S_2$ into $I_2$ – as we would wish – then we refer to $D_1$ as a delta refinement. Delta refinement may be used repeatedly to enhance a component over and over as Figure 1 indicates.

We present two examples of delta refinement. The first enhances a specification by inheritance (*class extension*) and the second enhances the specification of an algorithm (*specification adaptation*).

### 6.1. Class Extension

Class extension allows specifications to include extra features through inheritance. For our purposes we require that descendant classes include all the features of their parents, although this is not always the case in practice due to descendant hiding [17]. Consider a class that defines a point in the 2-dimensional integer plane, including a function `distance` that calculates the distance between two points:

```
class Point ^=
abstract
 var x:int, y:int;

interface

 build{a,b:int}
   post x!=a, y!=b;

 function x;
 function y;

 function distance(p:from Point):int
   ^= (let a^=x-p.x;
       let b^=y-p.y;
       let rootVal^=(a^2) + (b^2);
```

```
      that i::0..rootVal :-
        (i*i <=rootVal < ((i+1)*(i+1)))
      )
  via
    value ? is int;
    end;
end;
```

(Keyword `that` introduces a variable that is assigned the unique element in a collection that satisfies the accompanying predicate. PL allows a question mark following `via` to indicate that code is omitted; we are omitting the code here because it contributes nothing to the example.) Note that parameter `p` of function `distance` has type `from Point`; this indicates that `p` may be bound to any object of class `Point` or a descendant of `Point`. The function `distance` employs an integer approximation to the square root of an integer. Let us suppose that the implementation is animated and evaluated by the client, and that he/she then requests that `Point` be enhanced to include colour information. The new class is constructed using inheritance as follows:

```
class ColourPoint ^= inherits Point
abstract
 var colour:int;

interface
 build{a,b,c:int} inherits Point{a,b}
   post colour!=c;
 function colour;
 function same(c:from ColourPoint):bool
   ^= colour=c.colour;
end;
```

Observe that the implementation code of `Point` is retained in `ColourPoint`.

Inheritance polymorphism does not occur in PL by default, but must be explicitly indicated using the `from` keyword. On the face of it this seems like good practice, but in fact the verifier is not at all comfortable with it because it can no longer predict attribute types at run time. We found ourselves avoiding the use of `from` where possible. Notwithstanding this, class extension seems to be quite well supported by PD.

### 6.2. Specification Adaptation

By *specification adaptation* we refer to the enhancing of a method either by weakening the assumptions on its inputs and/or by strengthening the requirements on its outputs. This is what happens when methods are overridden according to the rules of *behavioural subtyping* [16]. It may also arise when we discover an implementation of a method that delivers more than the specification actually asked for. In this case, it is sometimes appropriate to enhance the specification to reflect this.

Consider a class that defines some abstract model and set of interfaces for a suite of mathematical functions. Suppose one of the functions calculates the square root of a nonnegative real number to some given degree of accuracy:

```
class MathFunctions  ^=
interface
 function sqrRt(val:real,eps:real):real
  pre val>=0.0, eps>0.02
   ^= ?
  assert result^2<=val<(result+eps)^2;
end;
```

Suppose now that a prototype of the system is generated and animated, and that during this phase we discover and decide to retain an implementation of `sqrRt` that works for smaller values of `eps`. PL allows us to upgrade the specification appropriately, by defining it as an adaptation of the original. The new version has both a weaker precondition and a stronger post-assertion:

```
class BetterMathFunctions ^=
      inherits MathFunctions
interface
 redefine function
          sqrRt(val:real,eps:real):real
  pre val>=0.0, eps>0.00002
   ^= ?
  assert result^2<=val<(result^2)+eps;
end;
```

(Keyword `redefine` introduces a function that is being overridden.) Specification adaptation is useful because it provides formal support for method overriding in inheritance. We note, however, that it does not sit comfortably with design by *specialization* of classes [22]. On the other hand, PL supports specialization by providing so-called *functional contracts*. By this is meant that instead of hard-coding a specification into a `pre` clause, we can use class functions to define the contract. The advantage is that we may subsequently redefine the functions at different levels of the inheritance hierarchy. This gives the contracts a dynamic form that allows specialization and behavioural subtyping abstractly. Functional contracts as provided by PD provide strong support for specification adaptation.

## 7. Larger Experiments with PD

We briefly report on two larger examples that we worked on, namely a library management system and a resource

manager. More details can be found on the web site accompanying the paper. The web site also records our experiments with other language features such as as higher-order functions and generics.

The library management system is essentially that formulated by Kemmerer [12]. The software has to manage a collection of library users consisting of both registered borrowers and staff members, and a collection of borrowable library items. Our specification made use of some primitive data types in PL that provide good functionality but are computationally expensive. They are therefore good candidates for refinement.

The library catalog was specified as a `set` of items, and we began by seeking to data refine this into a permanent file data structure. We discovered that file handling in PL is quite primitive and our effort failed because of this. Our next approach was to refine the set into a custom-built hash table implemented as an array of hash buckets. Specifying and verifying the hash table per se proceeded without difficulty, but the verifier struggled to prove that the hash table data refined our original set. Some progress was made by specifying the hash table differently, this time in a style similar to that of the `set` class. However, we never achieved full verification and indeed we were led to discover some underlying difficulties with type transformation as discussed in Section 5.2.

The purpose of a resource manager is to manage a pool of resources that are shared out among competing processes. The manager must allocate the resources dynamically on demand, while seeking to ensure high levels of use, and dealing sensibly with any deadlock situations that might arise. This proved particularly difficult as PL does not support any kind of concurrency. We tried to simulate multi-threading and deadlock but with limited success and concluded that it was not possible to use PD to reason about properties of concurrent systems.

Further analysis of these experiences can be found in [6].

## 8. Comparisons with other tools

The tools most closely related to PD include the B-method, ProofPower, and JML. The B-method [4], supported by both Atelier-B [2] and the B-Toolkit [14], is one of the best known methods for formal software construction. In place of PD's single step refinement, B supports an iterative refinement process, ultimately leading to a C implementation. B does not support object-oriented programming. B supports refinement relations between abstract and concrete data. Unfortunately, B contains two languages for specification alone, both of which require mathematical expertise and are challenging to learn and use. Unlike PD, the theorem prover is interactive, and requires developers to have a deep understanding of the logic of B. Work is ongoing with B to make it more user-friendly with tools such as U2B [20].

ProofPower [21] is a tool that supports the refinement of Z specifications into Ada programs. It differs from PD in that it supports iterative refinements during software development. Each refinement step generates proof obligations that are discharged through application of the interactive theorem prover. The theorem prover is based on HOL [10] and requires an understanding of the associated logic. ProofPower offers a mathematically richer refinement process than PD. However, it relies on multiple languages and technologies that must be understood before development with ProofPower can commence.

The Java Modeling Language (JML) [15] is essentially a specification language for Java, employing Java syntax and semantics as much as possible. It is one of the few object-oriented systems that supports some form of refinement. JML is supported by a tool called ESC/Java2 [7]. This is not a verifier but an "extended static type checker" — C programmers may think of it as a very sophisticated version of `lint`. JML supports specification through design by contract. Behavioural subtyping is the mechanism by which refinement is supported. ESC/Java2 is built around the *Simplify* theorem prover which is neither sound nor complete [13], though research to replace it is underway. The LOOP tool [11] is similar to ESC/Java2, but it attempts full verification. LOOP is currently under development.

We summarize our comparisons in Table 1. In comparison with other tools, PD offers a software oriented approach to refinement rather than a brutally mathematical one. This may reduce the specification and refinement options for the developer, but it increases the audience that might use such a tool. PD supports specification and implementation in a relatively simple language, so its learning curve is quite gentle for practicing software engineers.

## 9. Conclusion

PD purports to do what is still an ambition for other software development tools: to support the construction of object-oriented software using refinement as a key technique while providing complete formal verification. The tool is not difficult to use and its associated specification/programming language is expressive, providing multiple specification styles for software development.

Unfortunately, the verifier is not strong enough to prove a number of refinement techniques correct without excessive additional work. It often requires the developer to re-work code and add assertions that to the human appear needlessly strong. The verifier was custom built for PD, and no supporting documentation on its architecture is provided. Its output is often difficult to understand, and it provides poor guidance as to the causes of failure. The Perfect Language

| | **B/B4Free** | **ProofPower** | **ESC/Java2** | **PD** |
|---|---|---|---|---|
| *Refinement Support* | Alg, Data | Alg, Data | Delta | Alg, Data, Delta |
| *Refinement process* | Iterative | Iterative | Iterative | Single Step |
| *Verification* | Interactive | Interactive | Auto | Auto |
| *OO* | No | No | Yes | Yes |
| *Language(s)* | AMN, GSL, C | Z, Ada, HOL | JML, Java | PL |
| *IDE* | Command Line | Command Line | Basic GUI | Basic GUI |
| *Learning Curve* | Steep | Steep | Slight | Gentle |

**Table 1. Summary of Refinement Tools**

provides fairly good specification and implementation elements, but the verifier limits the ability to use all the features as one would wish. Indeed, one is inevitably led to adopt a style of specification that best suits the verifier, rather than one that suits the problem.

Despite all this, PD has attractive features. It is relatively easy for software engineers to learn, even if they are not mathematically inclined. Once learned, it rewards the user with some surprisingly good verifications, and often enough uncovers unexpected errors in code. It is particularly rewarding when it successfully test runs specifications. For these reasons, we have found it to be a good tool to use in the classroom. It can enthuse students and software engineers with what is possible in formal programming, and what advantages it might bring. The technology has some way to go before it is industrial strength, but it might well be effectively used in designing and verifying small parts of critical code.

## 10. Acknowledgements

We would like to thank the anonymous referees for their constructive and helpful comments. Many thanks to David Crocker of Escher Technologies for his help and support with Perfect Developer. Enterprise Ireland provided funding under the Basic Research Grant SC/03/278.

## References

[1] Escher Technologies website: `http://www.eschertech.com/index.php`.

[2] Atelier B website: `http://www.atelierb.societe.com/index_uk.htm`.

[3] *The Perfect Developer Language Reference Manual, Version 3.0 : `http://www.eschertech.com/product_documentation/LanguageReference/language_reference.pdf`*, 2004.

[4] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[5] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[6] G. Carter. Automating formal software development. Master's thesis, (in Preparation), Dept. of Computer Science, National University of Maynooth, 2005.

[7] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. Technical Report R0413, Security of Systems Group, Nijmegen Institute for Computing and Information Science, 2004.

[8] D. Crocker. Perfect Developer: A tool for object-oriented formal specification and refinement. In FME 2003, Tools Exhibition Notes, `http://www.eschertech.com/papers/fme_2003_tools_paper.pdf`, 2003.

[9] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[10] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[11] B. Jacobs and E. Poll. Java program verification at Nijmegen: developments and perspectives. *Proc. Software Security - Theories and Systems*, LNCS Vol. 3233:134–153, Kokichi Futatsugi, Fumio Mizoguchi, Naoki Yonezaki (editors), Springer-Verlag, 2004.

[12] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11(1):32–42, 1985.

[13] J. Kiniry and E. Poll. Design by contract and automatic verification for Java with JML and ESC/Java2. *Proc. ECOOP 2004, Object Oriented Programming*, LNCS Vol. 3344, Martin Odersky (editor), Springer-Verlag, 2004.

[14] K. Lano and H. Haughton. *Specification in B: An Introduction Using the B Toolkit*. Imperial College Press, 1996.

[15] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. Technical Report 98-06q, Department of Computer Science, Iowa State University, 2001.

[16] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[17] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[18] C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[19] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1987.

[20] C. Snook and M. Butler. Verifying dynamic properties of UML models by translation to the B language and toolkit. *Proc. UML 2000: Advancing the standard*, LNCS Vol. 1939, A. Evans, S. Kent, B. Selic (editors), Springer-Verlag, 2000.

[21] P. Steggles and J. Hulance. *Z Tools Survey*, 1993. `ftp://ftp.ist.co.uk/pub/doc/zola/ztool-survey.ps`.

[22] M. Torgersen. Inheritance is specialization. *Proc. Object Oriented Technology. ECOOP 2002 Workshop and Posters, LNCS vol. 2548*, Juan Hernandez, Ana M.D. Moreira (editors), Springer-Verlag, 2002.

[23] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.