Perfect Developer: A tool for Object-Oriented Formal Specification and Refinement

David Crocker

Escher Technologies Ltd., Mallard House, Hillside Road,
Ash Vale, Aldershot GU12 5BJ, UK,
`dcrocker@eschertech.com`,
WWW home page: `http://www.eschertech.com`

Abstract. *Perfect Developer* is a formal methods tool for developing specifications and refining them to code. High productivity is achieved through the use of a push-button theorem prover using advanced automated reasoning technology. The tool can import UML models and generates final code in Java or C++. It is being used both commercially and for teaching formal methods in universities.

1 Introduction

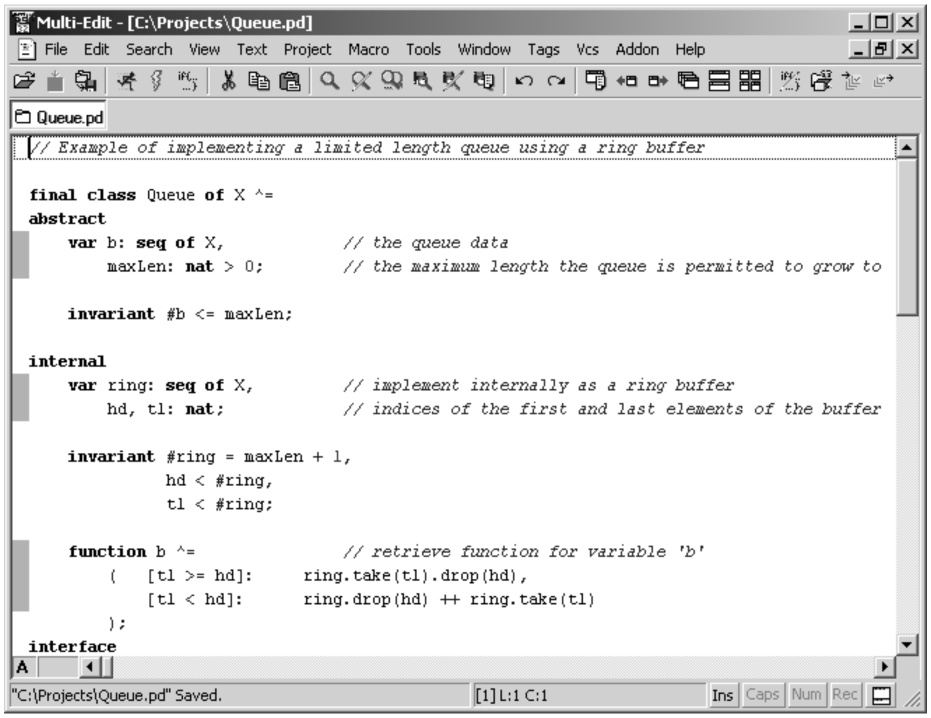
Perfect Developer is a high-productivity software development tool for developing formal specifications and refining them to code. The tool uses advanced automated reasoning to discharge almost all proof obligations without user intervention. Advanced mathematical knowledge is not a pre-requisite, which means that any developer fluent in an object-oriented language such as Java or C++ should be able to learn the notation.

A single notation is used to express functional requirements, specifications, and code. Specifications can be verified against the requirements. They can then be refined into code either manually (with verification) or, in many cases, automatically. The final code is automatically translated into ready-to-compile Java or C++.

Perfect Developer is of commercial interest primarily for safety-critical or mission-critical applications, but its high productivity makes it suitable for non-critical applications too. It is also used by several universities for teaching formal methods to undergraduates and for research.

2 Outline of the tool and the language

The *Perfect* language has been designed to be accessible to software developers with limited mathematical knowledge. It has the look and feel of an object-oriented programming language but the power of a specification language (Fig. 1). Type-safety is a major consideration when developing critical systems and the safety of the language is comparable with Ada in this respect.


Multi-Edit - [C:\Projects\Queue.pd]
File Edit Search View Text Project Macro Tools Window Tags Vcs Addon Help
Queue.pd

```
// Example of implementing a limited length queue using a ring buffer

final class Queue of X ^=
abstract
    var b: seq of X,            // the queue data
        maxLen: nat > 0;        // the maximum length the queue is permitted to grow to

    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 of the buffer

    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
```

"C:\Projects\Queue.pd" Saved. [1] L:1 C:1 Ins Caps Num Rec

Fig. 1. Source text

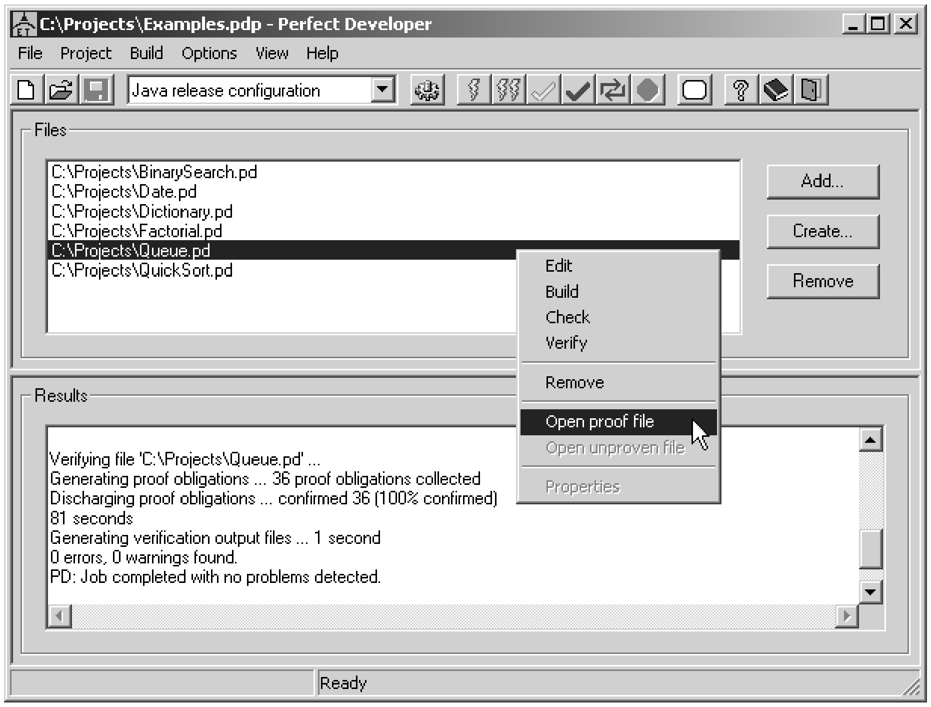
Each class is defined with an abstract data model and the class methods are specified in terms of this model. If the abstract data model of a class is refined to a more efficient implementation model, then in accordance with the principle of encapsulation, the specification remains unaltered and the refinement is invisible to clients of the class. Method specifications can be individually refined towards code where necessary. Further information about the language can be found in [1] and an introductory tutorial is available at [2].

The tool generates the proof obligations needed to establish that the requirements are well-formed, that the specifications are well-formed and satisfy the requirements, and that the code is terminating and behaves according to the specification.

The prover runs without user intervention. It attempts each proof obligations and delivers either a proof (which can be output in either HTML or Tex format), or a warning that a proof has not been found together with information to assist in finding the likely error. For correct programs, typical percentages of proof obligations successfully discharged are in the high nineties.

3 Using *Perfect Developer*

Perfect Developer runs on standard PCs running Windows or Linux. The interface to the toolset is the Project Manager (Fig. 2). This module allows users



Java release configuration

```
C:\Projects\BinarySearch.pd
C:\Projects\Date.pd
C:\Projects\Dictionary.pd
C:\Projects\Factorial.pd
C:\Projects\Queue.pd
C:\Projects\QuickSort.pd
```

```
Verifying file 'C:\Projects\Queue.pd' ...
Generating proof obligations ... 36 proof obligations collected
Discharging proof obligations ... confirmed 36 (100% confirmed)
81 seconds
Generating verification output files ... 1 second
0 errors, 0 warnings found.
PD: Job completed with no problems detected.
```

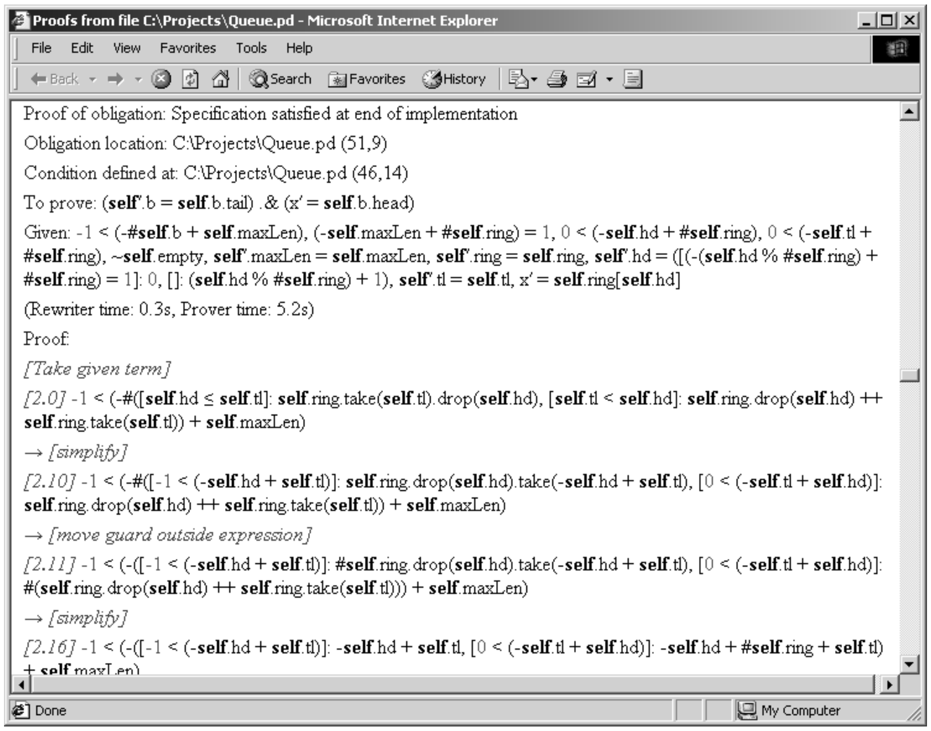
Fig. 2. The Project Manager

to create projects, add files, edit files, import UML models, verify or build individual files or the whole project, view proofs, and generate a cross-reference. The build process can be configured to compile the resulting Java or C++ code automatically.

Proofs may be generated in HTML format for on-screen viewing or electronic publishing (Fig. 3), or in Tex format where hard copy is needed.

4 Theoretical foundations

Perfect Developer was inspired by Floyd-Hoare logic and Dijkstra's weakest-precondition calculus [3]. It uses the design-by-contract approach to specification of class methods, and the refinement approach to implementation. The language has the power of first-order predicate calculus, plus a few higher-order constructs. The mechanisms of pre-conditions, post-conditions, post-assertions,


Proofs from file C:\Projects\Queue.pd - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Done
My Computer

```
Proof of obligation: Specification satisfied at end of implementation

Obligation location: C:\Projects\Queue.pd (51,9)

Condition defined at: C:\Projects\Queue.pd (46,14)

To prove: (self'.b = self.b.tail) .& (x' = self.b.head)

Given: -1 < (-#self.b + self.maxLen), (-self.maxLen + #self.ring) = 1, 0 < (-self.hd + #self.ring), 0 < (-self.tl +
#self.ring), ~self.empty, self'.maxLen = self.maxLen, self'.ring = self.ring, self'.hd = ([(-(self.hd % #self.ring) +
#self.ring) = 1]: 0, []: (self.hd % #self.ring) + 1), self'.tl = self.tl, x' = self.ring[self.hd]

(Rewriter time: 0.3s, Prover time: 5.2s)

Proof:

[Take given term]

[2.0] -1 < (-#([self.hd ≤ self.tl]: self.ring.take(self.tl).drop(self.hd), [self.tl < self.hd]: self.ring.drop(self.hd) ++
self.ring.take(self.tl)) + self.maxLen)

→ [simplify]

[2.10] -1 < (-#([-1 < (-self.hd + self.tl)]: self.ring.drop(self.hd).take(-self.hd + self.tl), [0 < (-self.tl + self.hd)]:
self.ring.drop(self.hd) ++ self.ring.take(self.tl)) + self.maxLen)

→ [move guard outside expression]

[2.11] -1 < (-([-1 < (-self.hd + self.tl)]: #self.ring.drop(self.hd).take(-self.hd + self.tl), [0 < (-self.tl + self.hd)]:
#(self.ring.drop(self.hd) ++ self.ring.take(self.tl))) + self.maxLen)

→ [simplify]

[2.16] -1 < (-([-1 < (-self.hd + self.tl)]: -self.hd + self.tl, [0 < (-self.tl + self.hd)]: -self.hd + #self.ring + self.tl)
+ self.maxLen)
```

Fig. 3. Proof output

class invariants, recursion invariants, loop variants and invariants are all present. Object-oriented features such as inheritance, polymorphism, and dynamic binding (dynamic dispatch) are handled by appropriate extensions to these mechanisms, in conformance with the Liskov Substitution Principle.

The logic used is a many-sorted logic of partial functions, which in the prover mostly reduces to a classical two-valued logic. There are additional inference rules to handle polymorphism, dynamic binding, and other constructs that do not fall within the framework of first order logic.

5 Applications

5.1 Teaching formal methods

Many universities today teach software development using Java and modeling using UML. *Perfect Developer* fits well with such courses, since it has an object-oriented basis, can import UML models, and generates code in Java (or C++).

Because it uses a fully automatic theorem prover, knowledge of proof techniques is not required, so that *Perfect Developer* is accessible to students with limited mathematical knowledge.

By generating ready-to-compile code, *Perfect Developer* provides a quick route from writing the specification to running the program. This gives students fast feedback on their work, helping to maintain student motivation.

5.2 Commercial software development

Unlike most other formal methods tools, *Perfect Developer* offers high productivity, thanks to its use of advanced automated reasoning in the prover. Many specifications can be automatically refined to code, so that the amount of code that has to be written is substantially reduced.

Correct use of *Perfect Developer* will eliminate debugging and re-testing activity, leading to lower costs and shorter time-to-market. Unlike most other formal methods tools, *Perfect Developer* does not require all its users to have advanced mathematical skills. It supports and encourages (but does not mandate) object-oriented development.

It can be used to produce a functioning prototype quickly, which means that a fully-functional model of the system can be rapidly constructed. This can be demonstrated to users, to check that their requirements have been correctly captured before large development costs have been incurred. Final code can be generated in Java or C++, allowing easy integration with other components written in those languages.

For aviation software development, *Perfect Developer* offers some additional benefits. The NASA/FAA working party on Object Oriented Technology in Aviation is producing a handbook which identifies and seeks to resolve safety and DO-178B certification issues when using object oriented technology. *Perfect Developer* operates very substantially in accordance with the guidance in the draft handbook. In particular, dynamic binding is safely handled in accordance with the Liskov Substitution Principle, using the Design By Contract approach. Traceability from requirements to code is a major concern with applying the object-oriented paradigm to aviation software development, but *Perfect Developer*'s proof output contains all the information needed to establish traceability from requirements to code.

References

1. *Perfect Developer* Language Reference Manual.
   http://www.eschertech.com/product_documentation/lrm.htm
2. A *Perfect Developer* Tutorial.
   http://www.eschertech.com/tutorial/contents.htm
3. A Discipline of Programming: Edsger W. Dijkstra. ISBN 013215871X