National University of Ireland, Maynooth
MAYNOOTH, Co. KILDARE, IRELAND.

DEPARTMENT OF COMPUTER SCIENCE,
TECHNICAL REPORT SERIES

Software Specification, Implementation and Execution with Perfect

Gareth Carter and Rosemary Monahan

NUIM-CS-TR-2005-07

Software Specification, Implementation and Execution with Perfect

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

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

June 17, 2005

1 Abstract

Perfect is an Object Oriented programming language that is supported by the Perfect Developer software development tool. The paper presents the techniques that Perfect supports for the specification and implementation of software. The executable code produced by Perfect is also discussed. A guideline to the techniques of software development is provided by the paper, illustrating the many software development mechanisms that are supported by Perfect and the Perfect Developer tool.

2 Introduction

Perfect Developer is a formal methods tool for developing specifications and refining them to code[8]. The tool comprises of the programming language, Perfect[13] and a theorem prover for verifying software systems. Perfect aims to support *Object Oriented* implementation features[8]. Object oriented program is concerned with the description of runtime objects. Objects that exhibit similar behaviour are described by *classes*. Perfect must be rich enough to support the development of *specifications*, the *refinement process* and the resulting *implementation language*. Code written in Perfect may be compiled into other

higher level languages. This translation will have an impact on the *runtime* uses of the software. These issues are explored in this paper.

3 The Specification Language

The specification language of Perfect defines basic data types and provides developers strategies to specify abstract data types. Design by Contract is the primary specification strategy in Perfect. Other strategies supported include aspects of Algebraic Specification, Model Oriented Specification, Functional Specification and Graphical Representation. This section details elements of the specification language of Perfect and the specification strategies that it supports.

3.1 Data types

Perfect has the following basic data types through which it can construct more complex data types.

`bool`	Boolean values of `true` and `false`
`int`	Integers
`nat`	Natural numbers, (integers greater than 0)
`real`	Real numbers
`char`	Characters
`void`	The `null` value

The data types obey the standard mathematical definitions. There exists no limit on some of the data types (e.g. integers are infinite). Perfect is also an implementation language with implementation concerns[3]. Overflow is not in the specification. The developer must consider it. Software that breaks rules of this type may be verified by the theorem prover but behave incorrectly.

Collection types are supported by Perfect. The mathematical types of sets, bags, maps and sequences are represented. The specification of these types is incomplete in the library. Some features are informally defined. The specifications of the data types obeys no common style. Properties of data types exist within the system that are not documented. The collection types are part of the implementation language of Perfect and specifications on their implementation costs should be formally present.

3.2 Assertions

An assertion is a boolean expression that when evaluated in the system, should always be equal to true. Assertions may involve *quantification* over types and *pure function* calls. Existential quantification is constructed with the `exists` keyword. Universal quantification with the `forall` keyword. Quantification can be over types for specifications. An assertion cannot change the state of

the system. The assertions may refer to any of the variables of the system. The assertions may relate a previous state of a variable to a current state.

Each assertion encoded in the system generates a *proof obligation* by the Perfect Developer tool. It must be verified for the system to be verified correct.

3.3 Design by contract

Design by Contract uses contracts in specifying the input-output relationship of features of a class. The contract has two distinct sections, the *pre-condition* and the *post-condition* (known in Perfect as the *post-assertion*). The pre-condition specifies what must be true at the outset of a call to the feature. The post-assertion specifies what is guaranteed to be true at termination of a successful execution of the feature. This strategy was first employed in the Eiffel language[10]. JML[9] offers a richer specification language.

Both pre-condition and post-assertion are assertions. The pre-condition follows the keyword `pre`. The post-assertion is stated as an assertion that follows the feature. Confusion may arise in Perfect as the `post` keyword is used to describe what Perfect term a *post-condition*. The post-condition is not an assertion. The post condition is used to change state.

Another aspect of the Design by Contract strategy is the *invariant* of the system. An invariant is an assertion that is true upon any call and at termination of a call to a feature of the system. The invariant may be broken during the execution of a call to a feature. The ability to break the invariant is an ongoing research area known as *ownership types*[7]. Perfect permits the breaking of the invariant, generating proof obligations if required.

3.4 Algebraic Properties

An algebraic specification is defined by $< \Sigma, E >$[16]. The Σ element describes the model of the software and the features that are present. E describe constraints of the model and its behaviour. It permits abstraction of the features of the model. Perfect capture E as *algebraic properties*. A `property` is an assertion that accepts parameters. Parameters may be constrained in a pre-condition. The parameters may change state. The result of the change is described in the assertion. Properties may relate to an object of the class or be quantified over the class.

In Algebraic specification languages such as CafeOBJ[12], the algebraic specification is used to construct the executable system. This is not supported by Perfect. Algebraic properties generate proof obligations. Properties are most useful for test case analysis of the system being developed.

3.5 Model Oriented

Model Oriented specifications defines a *mathematical model* of the system and the *operations* that act upon that model. The model of the system is treated as functions or relations between the data types of the model. The operations

over-ride these functions, changing its state. The specification language Z[15] is a model oriented specification style.

Perfect contains some of the mathematic structures for a model oriented approach. By developing a model of the software as a set of mapping functions, the developer can specify the meaning of the model clearly and precisely. Perfect does not contain a rich set of mathematic structures. Pure mathematical structures are not easily defined. A model oriented approach becomes cumbersome without a mathematical language.

3.6 Functional Specification

Functional Programming languages[5] enjoy mathematical properties. This allows them to be used for both specification and implementation such as Haskell[14]. Some functional language techniques are very useful for specification. One technique for specification is *Higher Order Functions*.

Higher Order Functions permit functions to be passed as parameters to other functions, or be returned as results from functions. Higher Order functions can be re-used with different low level functions that exhibit similar patterns but achieve different results. Some higher order functions have repeatedly shown to be useful (e.g.`fold`,`map`,`filter`). They exist in Perfect as the `over`, the *transform* and the `those` expressions respectively. These higher order functions are apply to the data types of the language. Perfect does not provide the means of constructing other higher order functions or using these with other data types. This restricts the richness of higher order functions for specification.

3.7 Graphical Notation

A *graphical notation* uses images to represents the behaviour or structure of a system. This enables developers to get an overview of the system. Graphical Representations are often informal. They are useful for the maintaining and understanding large software systems. The UML[1] is one of the most common graphical representation for object oriented software systems.

Perfect Developer does not employ any graphical representation as standard within its environment. All development is undertaken through source files of text. Perfect supports *object oriented* concepts and lends itself to the UML quite naturally. Currently there exists a UML importer for class diagrams. This allows developers to construct a class diagram for their software system outside of Perfect Developer and generate files for each class with the inheritances and aggregations. The system does not support reverse engineering of the code to the UML. The addition of formal specifications need not be represented in this UML model.

3.8 Conclusions

Perfect supports a variety of styles of specification. The specification language is used to describe an abstract model of the system. This model may be ani-

mated and developers can validate the specification matches the requirements. A software system constructed using this specification language will usually be inefficient.

4 The Refinement Step

Refinement is a formal process that constructs a *concrete description* of a software system from an *abstract description*[2]. This process assumes the correctness of the specification has been validated. Perfect supports a single refinement step[6]. There are several categories of refinement permitted by Perfect.

4.1 Abstract Model

The abstract model of a class is the simplest model of the system which illustrates its semantics. It should not concern the efficiency of the system. Data derivable from other parts of the model should not be included. A full specification of the model is given in the `abstract` section of a class. All access routines should be written in the `interface` section of the Perfect class.

The `abstract` section of the class can contain pure functions. These functions serve as helper functions to the specification. Functions declared in the `interface` section cannot be used in specifying invariants of the class. Elements of the abstract model may be *non-deterministic*. Is is captured by stating that a function `satisfy` some boolean expression. This expression must involve the `result` value. Non deterministic functions must be refined.

4.2 Internal Model

The internal model of a class describes how the class is implemented. It is optional if the specification is deterministic. A *retrieve function* is declared describing how to construct the abstract model from the internal model. The retrieve function is in this direction. The retrieve function shows the correctness of the refinement. Additional invariants may be needed to aid verification of the refinement. If the data model is refined, all interfaces must be refined. The refinement is a `via` expression. Statements in the refinement may only reference the internal model or the methods of the class.

4.3 Refinement Categories

Three forms of refinement are supported. They are Algorithm Refinement, Refinement by Parts and Data Refinement. Algorithm Refinement translates a specification into an implementation removing non-determinism and inefficiencies. Refinement by Parts permits iterative refinements while augmenting a specification supporting the specification development process. Data Refinement modifies the data structure used in the specification retaining correctness.

4.4 Loop Refinements

When refinement occurs, it is common to introduce loops. Loops are one of the most complex program fragments to prove correct. A loop in Perfect is structured as follows.

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

The `var` expression is where all local variables are declared. Variables are not permitted to be declared as part of the loop body. A let statement may occur within the loop body, permitting values to be stored. These values may not change within a single iteration of the loop. The `change` expression declares those variables outside the loop body that may be modified by the loop. This removes the risk of accidentally overwriting data stored outside the loop. The `keep` expression is the *loop invariant*. A loop invariant is an assertion that defines what is true at entry to the loop, at exit from the loop and at each iteration through the loop. It is permitted to be broken in the loop body, but not after an iteration. The `until` expression is the terminating condition of the loop. The `decrease` expression is the *loop variant*. The loop variant is a integer value. This value must decrease each iteration of the loop and it must never be below zero.

4.5 Conclusion

The Refinement step takes a specification of the software to an implementation. Applying some set of the refinement categories should increase the efficiency of the system without introducing errors. This implementation should be provably correct with respect to the specification. It should also be error free and robust.

5 The Implementation Language

To be theoretically sound, Perfect must restrict elements of the object oriented paradigm. Issues like difference between *type and class*, *encapsulation*, *inheritance*, *polymorphism* and dynamic binding need to be addressed. Perfect also treats *equality* of objects differently. There are other notational differences with traditional object oriented languages. These topics will be explored in this section.

5.1 Type and Class

A value in a software system may be given a *type*. This defines the set of values that the value belongs to. In object oriented programming languages, values are objects. Each object belongs to a *class*, which defines it's behaviour. Type and class are often used synonymously even if this is not justified[11]. In Perfect, type and class are synonymous and this rule is enforced by the language.

5.2 Encapsulation

Encapsulation permits data hiding within a class. Encapsulation is supported strongly in Perfect by refinement, permitting an internal model that differs from the abstract model. Perfect also permits the standard notion of data hiding by making abstract data inaccessible from outside the class. To allow abstract data be read from, it may be defined as a *function* as part of the classes interface. To allow abstract data be read from and written to, it may be defined as a *selector*.

5.3 Inheritance

Inheritance is the mechanism in object oriented programming languages for re-use of code. The re-used code comes from the *superclass* and re-used by the *subclass*. The subclass receives all the behaviour of the superclass but may be tailored to additional needs.

Design by Contract specifications decreases the flexibility of inheritance. Consider two classes in an inheritance relationship as follows:

```
class SuperClass ^=
    function someMethod
        pre  P
        ^=   A1
        assert Q
end;

class SubClass inherits SuperClass ^=
    redefine function someMethod
        pre  P'
        ^=   A2
        assert Q'
end;
```

For correct use of this inheritance relationship, it must be the case that:

$$P \Rightarrow P' \& Q' \Rightarrow Q$$

This restriction forces developers to have foresight about future use of a class.

Functional preconditions may be used to get around this restriction. A function call is used in the preconditions to a method rather than declaring it explicitly. This function can be redefined in subclasses permitting the strengthening

of preconditions. If a functional precondition is defined as false the method cannot ever execute. The method is then defined as `absurd`. This is known as descendant hiding.

5.4 Polymorphism

Polymorphism is the ability for objects of one type to masquerade as another type. In Perfect inheritance promotes polymorphism. An object of one class may masquerade its type as an ancestors type. Any `absurd` feature may break this ability because it won't have all the features of the superclass. *Binary methods* also break this ability. A binary method accepts a parameter of the same type as the current objects type. This parameter should vary depending on the type of the object. With polymorphism, the absolute type cannot be known. As inheritance cannot guarantee correct usage of polymorphism therefore inheritance is not permitted by default. To achieve polymorphism, Perfect uses the `from` keyword. This declares a type as polymorphic. The true type of the object will not be known.

5.5 Dynamic Binding

Dynamic Binding permits the execution code of a system to be determined at runtime rather than at compile time. Dynamic Binding suffers from lack of theoretical soundness. Errors may go unchecked at compile time resulting in less safe operation. Covariance, Contravariance and Invariance properties need to be determined for an object oriented language to guarantee correct behaviour. A test method for object oriented languages was developed[4]. This procedure leads to the late binding signature of the software.

Beugnard's test defines three classes called `Top`, `Middle` and `Bottom`. These inherit along the hierarchy that Top is the parent of Middle and Bottom and Middle is the parent of Bottom. These objects have no functionality but will be used as parameters to two other classes methods.

```
class Top ^= end;
class Middle ^= inherits Top end;
class Bottom ^= inherits Middle end;
```

The class `Up` contains three methods:

```
class Up ^=
    function cv(t: from Top) : string
      pre t within from Top
      ^= "up";
    function ctv(b:from Top):string
      pre b within from Bottom
      ^= "up";
    function inv(m:from Top):string
```

```
      pre m within from Middle
      ^= "up";
end;
```

And below this in the hierarchy is the class `Down`, which redefines these methods as follows:

```
class Down ^=
  inherits Up
    redefine function cv(m:from Top):string
      pre m within from Middle
      ^= "down";
    redefine function ctv(m:from Top):string
      pre m within from Middle
      ^= "down";
    redefine function inv(m:from Top):string
      pre m within from Middle
      ^= "down";
```

When tested, Perfect generated the following results.

calls	u	d	ud
cv(t)	up	error	error
cv(m)	up	down	down
cv(b)	up	down	down
ctv(t)	error	error	error
ctv(m)	error	down	down
ctv(b)	up	down	down
inv(t)	error	error	error
inv(m)	up	down	down
inv(b)	up	down	down

These results describe the signature of Perfect's dynamic bindings.

5.6 Equality

Equality between two objects is defined over equality of the type of the objects and equality of all the features of the objects. This differs from common object oriented programming languages where equality is based on the physical memory address of the objects. Perfects treatment of equality gives a truer measure of equality in a software system but may mislead programmers.

5.7 Method Segmentation

Methods in a Perfect class are divided into two categories, *functions* and *schema*. A function is a pure function. It has no side effects. Functions are useful for

returning information about the state of an object. Schema are operations that have side effects. They may change the state of the object the method is called upon, or they may change the the state of an annotated parameter to the schema. Schema may not return values except through this parameter passing mechanism. Annotations are required to show which parameters may be modified. A schema must make a state change.

5.8 Conclusion

The implementation language of Perfect has altered certain features of the object oriented paradigm to be theoretically sound. These restrictions give an added overhead, but may be beneficial.

6 The Executable Code

Perfect should be capable of producing efficient code. The language compiles into Java, C++ and Ada95. Some of key issues regarding this executable code include *code generation*, *wrapper classes*, *value semantics*, the *file and exception handling mechanisms*.

6.1 Code Generation

When the automatic code generation occurs, a *name mangling* may occur. This may involve renaming functions, or data members; changing the signature of a function; or renaming the abstract class names. Variables may be declared, given values and not referenced again, serving no viewable purpose. This happens in certain loops and concern the loop termination condition. Maintenance of the executable code is not possible. All code changes must be made at the Perfect front end to ensure correctness of the code.

6.2 Wrapper Classes

Programs require some interface to their users to execute. Usually this interface will take the form of a *graphical user interface*. Perfect does not support any mechanism for describing them. The interface must be constructed in the language the system is compiled to. This interface will have to pass values to and read values from the generated code. This interface has no verification or constraints on it.

In order to use this unverified interface an outer layer of the Perfect code must be constructed. This outer layer can have no pre-conditions. It must accept all inputs. It is called the *wrapper class* of the system. It is the obligation of the wrapper class to ensure pre-conditions are met for classes it instantiates and for methods called. This can entail a large overhead of checks depending on the interface to the user of the system. All communication to the user interface must be performed through the wrapper class.

6.3 File Handling

Most programming languages provide a simple and intuitive file handling mechanism. File Handling consist of providing an abstract data type of the file structure, providing an interface to the operating system, and handling errors when reading or writing to memory. These are tasks that Perfect is ideally suited for but the default mechanism is primitive. All messages associated with file handling are passed through an `Environment` object. This is not a global object and must be passed as a parameter to any object wishing to perform file handling operations.

The Environment object contains a few primitive schema that control file handling. Each schema has associated with it an `out` variable that provides some form of result. These result types are not uniform across the range of schema in the system. and may be united with other types. Post-conditions are written in terms of these result types leading to complex post-assertions. The `File` data structure is defined as a `seq of byte`. Reading and writing to a file will require significant *casting*. Even if the developer reads the data from a file correctly, there are not guarantees that the data will be used correctly.

6.4 Exception Handling

Exception handling is a programming language mechanism designed to handle runtime errors that are unforeseeable. The exception handling mechanism increases robustness of software systems with respect to events outside the systems knowledge. Perfect contains no exception handling language. Developers may use the design by contract strategy to generate software exceptions by including tests on the conditions of the contract. These checks will be performed at runtime to guarantee the contract is fulfilled. If a failure is encountered, the program should be immediately halted and the error printed to the screen. This mechanism sometimes fail to terminate the program, putting the system in an inconsistence state.

6.5 Value Semantics

Value semantics treats the contents of all variables as distinct objects even if equal. Most object oriented languages use *reference semantics*, where variables are holders of addresses of objects. Reference semantics makes reasoning about software difficult. This difficult is because of aliasing. In Perfect, aliasing cannot occur with most variables. Developers may create a `heap` object. Variables placed on the heap store addresses of the objects they point to. This leads to a redefinition of equality based on the absolute addresses and not the contents of the object.

Value semantics is easier to reason about. This ease comes at an execution cost. When passing variables as parameters or overwriting data, clones of the objects must be made. This ensures that the object is not referenced elsewhere.

This cloning is a deep clone as the attributes of the object must be cloned. This has huge implications for efficiency of execution and efficiency of memory management.

6.6 Conclusions

The executable code generated by Perfect is difficult to maintain and read. This is a problem as wrapper classes need to be constructed around this code. The efficiency of the code is questionable owing to value semantics. The mechanisms for file and exception handling are poor. Further to this code written in Perfect is single threaded and requires termination. This is part of the correctness of Perfect systems, but is not realistic in general purpose industrial applications.

7 Conclusions

The Perfect language is rich encompassing specification and implementation language details. The specification styles supported by Perfect provide developers with many options for defining the semantics of the system. Some of the strategies guide development of an implementation. Others are useful for testing software in a formal setting. Implementing systems in Perfect differs in several ways from traditional object oriented languages, but most of the key concepts can be found in some form. The refinement step in Perfect offers developers a good opportunity to develop code in one environment, retaining the abstract model and the internal model as separate aspects. Once code is generated from Perfect, systems may suffer owing to lack of efficiency and lack of support. Software constructed in Perfect will be more formal than most programming languages, but at a cost of flexibility and efficiency.

References

[1] Sinan Si Alhir. *UML in a Nutshell*. O'Reilly, Sebastapol, CA, 1998.

[2] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[3] Bernhard Beckert and Steffen Schlager. Software verification with integrated data type refinement for integer arithmetic. In *Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK*, LNCS. Springer, 2004. To appear.

[4] Antoine Beugnard. OO languages late-binding signature. In Martin Odersky, editor, *proceedings of the 9^{th} workshop on Foundations of Object-Oriented Languages*. ACM Press, January 2002.

[5] Richard J. Bird and Philip Wadler. *Introduction to Functional Programming*. International Series in Computer Science. Prentice-Hall, New York, NY, 1988.

[6] Gareth Carter and Rosemary Monahan. Refinement in perfect developer. Technical report, National Universtiy of Ireland Maynooth, Department of Computer Science, 2004.

[7] D. Clarke. Object ownership and containment, 2001.

[8] David Crocker. Perfect developer: A tool for object-oriented formal specification and refinement. as part of the Tools Edition notes at Formal Methods Europe 2003, 2003.

[9] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06q, Iowa State University, Department of Computer Science, December 2001. See `www.jmlspecs.org`.

[10] B. Meyer. *Eiffel: The Language & Environment*. Prentice-Hall, 1991.

[11] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 edition, 1997.

[12] Ataru T. Nakagawa, Kokichi Futatsugi, and Toshimi Sawada. *CafeOBJ User's Manual - ver.1.4*, '1.4 edition, April 08 2000.

[13] Escher Technologies. *The Perfect Developer Language Reference Manual*, 2.10 edition, September 2003.

[14] Simon Thompson. *Haskell: The Craft of Functional Programming (2nd edition)*. Addison Wesley, 1999. ISBN 0-273-03151-1.

[15] Jim Woodcock and Jim Davies. *Using Z, Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.

[16] xxxx. *Software Engineers Referece Book (ask Rosemary)*. xxxx, xxxx.