Introducing the Perfect Language

Gareth Carter

May 4, 2005

1 Abstract

Perfect Developer is an environment for developing software systems and verifying their correctness. The developer supports programming with the object oriented language Perfect. A general overview of the syntax of Perfect is provided. Interesting features of the language is highlighted and discussed. A small example is developed in Perfect illustrating many of the features of the language.

2 Introduction

The Perfect Developer environment supports development of correct software in the Perfect Language[2]. This language acts as both specification and implementation language. The specification of programs is primarily through *design by contract*. It supports *object oriented* features such as classes, data encapsulation, and *inheritance*. The language also supports single step *refinement* and the environment permits this refinement to be verified correctly.

The language is introduced demonstrating several of its key features. A general purpose class template is provided and used to document how to construct a class and perform a refinement in Perfect. Additional key components of the language are documented separately. A simple class is developed to illustrate how the language is used to create a class. The example shows a representation of numbers with strings. A simple data refinement process is performed. The documentation and illustration should provide an overview of the Perfect Language.

3 Background

3.1 Aspects of the language

Identifiers in Perfect consist of letters, underscores or digits. The first character of an identifier cannot be a digit. An identifier is case sensitive and can be of any length.

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 or equal to 0)
`real`	Real numbers
`char`	Characters
`void`	The `null` type
`set of X`	An unordered collection of unique items of type X
`bag of X`	An unordered collection of items of type X
`seq of X`	An ordered collection of items of type X
`map of (X,Y)`	A map of items of type X to items of type Y
`string`	A `seq of char`
`byte`	A `seq of bool` of length `8`

3.2 Design by Contract

Design by Contract uses contracts to specify 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 developed by Bertrand Meyer[1].

3.3 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

3.4 Refinement

Refinement[3] is a formal process that transforms specifications preserving correctness. A refinement process consists of a series of refinement steps that translate an *abstract* specification into a more *concrete* one. The abstract specification is the simplest representation of the system that defines the interface to the external world. The concrete implementation is the program that realizes this specification. In Perfect only a single refinement step is supported.

4 Class Structure

The *class* is the building block for development in object oriented languages. It defines the set of possible runtime *objects* that may be instantiated of that *type*. It defines the data model and the message passing capabilities of objects of this type. For this section keywords are lowercase and expressions are uppercase.

```
class CLASS_NAME1 ^= inherits CLASS_NAME2
abstract
  // Specification of data
internal
  // Concretization of data
confined
  // Private methods of class
interface
  // Public Methods of class
end;
```

The `CLASS_NAME1` is an Identifier that describes the class. The class may optionally inherit features from another class.

There are two levels of data model in Perfect, the *abstract* and the *internal*. The abstract model is the simplest possible data model of the software. The internal model is the concrete implementation of the model. Message passing in Perfect is performed through *method invocation*. Methods are defined in the *confined* and *interface* sections. The confined methods are private. The interface methods are public.

An abstract section solely can be written with the other sections optional. A confined section may be written solely with the interface section optional. An interface section may be written solely.

4.1 Abstract Section

The abstract section defines the *variables* and *invariants* of the model of the class.

```
abstract            // Specification of data
  var DATA1:DATA_TYPE1;          // variables
  invariant BOOLEAN_EXPRESSION1; // restrictions
```

The variables are declared with a `var` expression. The term `DATA1` is an Identifier that states the name of the variable. The `DATA_TYPE1` is an Identifier that states the type of the variable. Types can belong to the data types in Perfect or user defined data types of the system. The invariants are expressions that relate the abstract variables to each other. They define the`BOOLEAN_EXPRESION1` that always evaluates to true when an interface method is called or returned. The invariant may be false when a confined method is called or during the interface method execution.

4.2 Internal Section

The internal section defines the variables and invariants of the implementation of the class and the *retrieve function*. The retrieve function constructs the abstract data model from the internal data model.

```
internal            // Concretization of data
  var DATA2:DATA_TYPE2;          // variables
  invariant BOOLEAN_EXPRESSION2; // restrictions

  // retrieve function
  function DATA1
    ^= RETRIEVE_FUNCTION
```

The variable and invariant declarations are equivalent to those found in the abstract section. Variables may be declared in the internal model that augment the model rather than change the model. The retrieve function is only necessary when data refinement occurs. The retrieve function header is the variable name of the abstract model that has been refined. No parameters are passed to the function. The `RETRIEVE_FUNCTION` must evaluate to a value of type `DATA_TYPE1`.

4.3 Confined Section

The confined section defines the methods of the class accessible locally to the runtime object associated to them. These methods are accessible to objects of the current class or its descendants. There are three categories of methods, *constructors*, *functions* and *schemas*. Constructors define how a runtime object is built. Functions return evaluations on a runtime object and have no side-effects. Schemas change the state of the system. Functions come in two varieties, *deterministic* and *non-deterministic*. A deterministic function must always evaluate to the same value. A non-deterministic function may permit multiple correct evaluations.

```
confined           // Private methods of class
  // constructor
  build{PARAM_LIST1}
    pre BOOLEAN_EXPRESSION3      // pre-condition
    inherits CLASS_NAME2{VALUE_LIST1}
    post CONSTRUCTOR_BODY1;

  // deterministic function
  function FUNCT_NAME1(PARAM_LIST2):RETURN_TYPE1
    pre BOOLEAN_EXPRESSION4      // pre-condition
    ^= FUNCTION_BODY             // specification
    via IMPLEMENTATION_BODY1     // implementation
  assert BOOLEAN_EXPRESSION5;    // post-assertion

  // non-deterministic function
  function FUNCT_NAME2(PARAM_LIST3):RETURN_TYPE2
    satisfy FUNCTION_SPECIFICATION
    via IMPLEMENTATION_BODY2

  // static function
  nonmember function FUNCT_NAME3(PARAM_LIST4):RETURN_TYPE3
     FUNCTION1;     // Some function

  operator XX (PARAM_LIST6):RETURN_TYPE4
    OPERATOR_BODY;

  // variable access
  function DATA1;

  // variable overwrite
  selector DATA1;

  // schema
  schema SCHEMA_NAME1(PARAM_LIST5)
    post SCHEMA_BODY1;
```

Constructors are written using the keyword `build`. They may accept a parameter list. A parameter list states the identifiers of input variables and their type. Constraints on the input parameters may be encoded in the pre-condition of the build. If the class explicitly inherits from a superclass a call to the constructor of that class is performed with the `inherits` expression. The superclass constructor is passed a set of values matching its signature. Constructors must always establish the invariants of the system.

All functions consists of the `FUNCT_NAME1` identifier, the input parameter list `PARAM_LIST2` and the `RETURN_TYPE1` of the function. The `RETURN_TYPE1` is

a type. Functions may be written with the design by contract strategy. The pre-condition is a boolean expression written with the `pre` keyword. The post-assertion is a boolean expression written with the `assert` keyword at the end of the function. The post-assertion may reference the value of variables at the point of call of a function. To differentiate between the current and initial values, an apostrophe `'` follows the variable name for the current value.

Specification of a deterministic function is written with the `^=` (read "is defined as") expression. This specification alone must be executable. Specification of a non-deterministic function is written with the `satisfy` expression. This may determine a single result but is not immediately executable. Both functions may be refined to an implementation with the `via` expression. To generate an implementation non-deterministic functions must be refined.

Functions may be defined in two other formats. Functions may be static (i.e. be associated with the class and not any particular runtime object) or be defined as operators. A static function is declared `nonmember` and follows the same format as any function but may only reference data passed as a parameter. An operator allows infix, prefix or postfix formatting of the function. They may be unary or binary. Perfect has a restricted set of `XX` operator symbols.

A notational convenience of Perfect permits direct access to the data members of a class. Data members may be read only by declaring a `function` with the data member as its header. Data members may be read from and written to by declaring a `selector` with the data member as its header.

A schema must change the state of the runtime object that it is called on or the value of some parameters passed to it. If the schema changes the runtime object a `!` precedes the schema name. Within the post-condition of the schema some variables of the object must be changed. If the schema changes parameters passed to it, each parameter identifier must be followed by the `!` symbol. A schema cannot return values accept through this parameter updating mechanism.

4.4 Interface Section

The interface section defines the methods that are accessible to other runtime objects. As in the confined section, these can be constructors, functions and schemas. All methods can be constructed as found above. The schemas in the interface section must establish the invariant.

```
interface           // Public Methods of class
  // constructor
  build{PARAM_LIST6}
    pre BOOLEAN_EXPRESSION4      // pre-condition
    inherits CLASS_NAME2{VALUE_LIST2}
    post CONSTRUCTOR_BODY2;

  build{PARAM_LIST7}
    ^= CLASS_NAME1{VALUE_LIST3};
```

```
  // pure function
  function FUNCT_NAME4(PARAM_LIST8):RETURN_TYPE5
     FUNCTION2;     // Some function

  // schema
  schema SCHEMA_NAME3(PARAM_LIST9)
    SCHEMA;        // Some Schema
end;
```

An alternate form of a constructor can be written in the interface section that defines the build in terms of another constructor call. This permits one all purpose constructor to be defined in the confined section and more case-specific constructors to be written in the interface.

4.5 Generics

One may construct template or generic classes in Perfect. Generics allow classes to be developed with different types. When constructing instances of these classes, a type is passed to the constructor.

Perfect allows developers to construct their own Generic classes through the following template:

```
class CLASS_NAME3 of GENERIC_TYPE ^= RESTRICTIONS
```

The class may enforce some set of restrictions on the generic types that can be used. The restrictions may specify: a constrained or absolute type; a class hierarchy to belong in; a set of functions implemented (with possible associated contracts); that equality is defined; that total ordering be defined.

4.6 Axioms

An axiom is a boolean expression that denotes some absolute truth. The theorem prover associated with the Perfect Developer environment is not complete (i.e. there exist truths that it cannot prove). To aid theorem proving, developers may add explicit truth statements to the theorem prover. They will be assumed true and used for verification of the system.

4.7 Properties

Properties are similar to sigma-axioms in algebraic specification languages. 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.

4.8 Recursion and Looping

A loop in Perfect is structured 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;
```

The `var` expression is where all local variables are declared. Variables are not permitted to be declared as part of the loop body. The `change` expression declares those variables outside the loop body that may be modified by the loop. The `keep` expression is the *loop invariant*. The `until` expression is the terminating condition of the loop. The `decrease` expression is the *loop variant*. The loop variant is a integer value that decreases each iteration of the loop and must always be non-negative.

4.9 Conditionals

There are two forms of conditionals in Perfect, one is associated with specifications and the other with implementations. The standard template for a specification is:

```
[BOOLEAN_EXPRESSION_1]:
  STATEMENT_1;
  ...
[BOOLEAN_EXPRESSION_N]:
  STATEMENT_N;
[]:
  DEFAULT_STATEMENT;
```

There can be many disjoint boolean expressions in the conditional. The `[]` denotes all other cases not covered by the union of the boolean expressions that precede. There must be at least two branches of a conditional.

The implementation form of conditionals is similar to above with two additions:

```
if
... // as above
fi;
```

5 Numbers Example

An example of development with Perfect is now illustrated with use of a Number representation problem. It is a toy problem for the purposes of illustrating the

concepts previously discussed in as clear a manner as possible. The example is not be a feasible solution to the problem and would not be used in a software system.

5.1 Problem Description

A software system is required that performs operations on numbers. A Number is any data model that can be returned as a non-negative number through a digit function. The software system should illustrate other common numeric operations (i.e. addition). It is recommended that the data be modelled as strings as data input-output will be performed with strings.

5.2 Requirements

The first requirement that we want to capture is modelling the *Number* class. The only requirement of this class is that it has a *digit* function. We can use inheritance to define our string numbers based on a super class of `Number`.

```
deferred class Number ^=
interface
    build{};
    deferred function digit:nat;
end;
```

The class is `deferred`[1]. It can be used as a template for descendant classes.

Another requirement is to define what it means to be a string modelling a number. For the sake of this example, we consider it to be a non-empty string where all the characters are digits and there are no leading zeros, unless the number is zero. We may define a test for this as:

```
// Determines if an input string is a valid number;
function isNumberString(input:string):bool
  ^= (~input.empty) &
     (forall x::input :- x.isDigit) &
     (#input=1 | input.head~='0')
  assert result ==> (forall x::input :-x.isDigit);
```

This is a global function that takes an input string and returns if it is a valid string, according to our definition. This function can be immediately animated and data tested to ensure that we are capturing the meaning of a string being a number string.

This function allows us to create a *constrained type*[2] representing strings that are numbers.

```
class NumericString ^= those a:string :- isNumberString(a);
```

[1] see Appendix A.3
[2] see Appendix A.6

To guide verification, it must be stated that all numbers converted to strings are number strings. This is accomplished with an axiom:

```
  axiom assert forall n:nat :- isNumberString(n.toString);
```

An abstract model of the actual `StringNumber` class can now be constructed. It will have the general form:

```
class StringNumber ^= inherits Number
abstract
  // The Numeric String data model
internal
  // Our data refinement...
confined
  // Helper functions of the class
interface
  // functions schemas & their refinement
end;
```

The `StringNumber` class is a subclass of `Number`. It is not `deferred` and must provide specifications of all functions.

5.3 Abstract Model

The requirements suggest the data be stored as a string that represents a number. The variable `number` is declared of type `NumericString`.

```
abstract
  var number:NumericString;
```

5.4 Interface Section

Now that the abstract model is decided upon, the methods of the class are constructed. The first step is to decide the constructors of the class. The first constructor accepts a `string` input, the second a `nat` and the third is a default parameterless constructor.

```
interface
  // constructor that accepts a string representing an nat
  build{input:string}
    pre isNumberString(input)
    inherits Number{}
    post number!=input;

  // constructor that accepts a natural number
  build{input:nat}
    inherits Number{}
    post number!=input.toString;
```

```
// default constructor for the zero String Number
build{}
  ^= StringNumber{0};
```

By accepting all string inputs for the the first constructor, incorrect typing may occur in the abstract model. We must encode a pre-condition stating the input is a number string. The pre-condition will be statically verified if this constructor is used. The constructor contains a call to the super-class constructor of Number. Finally the abstract data variable `number` is assigned to with the value of input. Perfect uses *value semantics* so the assignment is equivalent to a clone.

The second constructor accepts a `nat`. By our earlier axiom, it is known that all natural numbers generate NumberStrings through application of the `toString` function. No pre-conditions are included. The final constructor is a default constructor for the class. It is defined in terms of the second constructor of the class.

Now the access functions of the class are declared:

```
function number;

define function digit:nat
  ^= toDigit(number,0);

redefine function toString:string
  ^= number;
```

The first function allows read only access to the data member of the class. No typing information is needed as it is implicitly typed by the abstract model. The second function is the specification of the `deferred` function `digit`. To perform this calculation we rely on some internal function `toDigit` that will be defined later. The third function will `redefine` the `toString` function. This function is defined in the `anything` class that is inherited by all classes in Perfect. This is analogous to `Object` in Java.

Readability of our system may be improved by using the `+` symbol for addition. It is defined as follows:

```
operator +(d:StringNumber):StringNumber
  ^= StringNumber{digit + d.digit};
```

Addition of two `StringNumber` objects is defined as addition on the `digit` representations.

A schema is declared to `update` the data model of an object.

```
schema !update(input:string)
  pre isNumberString(input)
  post number!=input;
```

Like the constructor, the `input` must represent a number string and this is encoded as a pre-condition. The assignment is then permitted.

5.5 Confined Section

Earlier we introduced a `toDigit` function that converts a string to a natural number. This function should not be accessible outside the class and is therefore defined privately in the `confined` section of code. The function will not make use of the abstract data model of the class so it is declared a `nonmember function`. It is defined recursively as:

```
confined
  nonmember function toDigit(input:string,num:nat):nat
    pre forall x::input :- x.isDigit
    decrease #input
    ^= ([input.empty]: num,
        []: (let tail^=input.tail;
             assert forall y::tail :- y.isDigit;
             toDigit(tail,((num*10)+input.head.digit))
            )
       );
```

The function reads each digit of the input string from left to right as a decimal number, and converting it into a natural number. This function only requires all the characters of the `input` represent digits. Each iteration of the recursion the size of the input sequence will `decrease`. When the input has been reduced to an empty string, the value of num is returned. Otherwise, the head is removed from the input. It follows naturally that all the characters of the `tail` are digits, but we `assert` this fact to aid theorem proving. A recursive call to `toDigit` is performed.

5.6 The Refinement

By storing a `Number` as a string value, there will be large costs. Firstly a string representation will consume more memory space internally. There are also computation costs associated with converting the string to a digit. it was required that numbers be represented as strings to the external world, but internally the string could be represented as a `nat`. This would be a valid and advantageous refinement.

5.7 Internal Model

The internal model of the class will consist of a variable `num` that represents the `number` variable. The retrieve function shows how this reconstruction can be performed.

```
internal
 var num:nat;
 // retrieve function
 function number
   ^= num.toString;
```

5.8 Refinement of Interface Model

With the internal model refined, all the class methods must be refined. The `nonmember` functions do not need to be refined. Before we may refine the implementations though, equality must be defined on the class. This is through a simple declaration of:

```
interface
  operator =(arg);      // defined to implement refinement
```

The first two constructors defined previously need to be refined.

```
  build{input:string}
    pre isNumberString(input)
    inherits Number{}
    post number!=input
    via num!=toDigit(input,0)
    end;

  build{input:nat}
    inherits Number{}
    post number!=input.toString
    via num!=input
    end;
```

To construct the new `internal` model from a `string`, it must be converted to a `nat` using the `toDigit` function of our class. Construction of the `internal` model from a `nat` is performed by assignment.

The `digit` and `toString` functions must also be given concrete implementations.

```
  define function digit:nat
    ^= toDigit(number,0)
    via value num
    end;

  redefine function toString:string
    ^= number
    via value num.toString
    end
    assert isNumberString(result);
```

A refinement returns a result through the `value` expression. With the function `digit` the `value` returned is the `num` variable.

The `+` operator is also refined.

```
  operator +(d:StringNumber):StringNumber
    ^= StringNumber{digit + d.digit}
    via value StringNumber{num+d.digit}
    end;
```

The update schema must convert the input to a `nat` before assignment takes place.

```
schema !update(input:string)
  pre isNumberString(input)
  post number!=input
  via num!=toDigit(input,0)
  end;
```

All methods must be refined before the code can be validated.

6 Conclusions

In providing the reader with a high-level overview of the Perfect language, a template was developed for constructing classes. This template documented most of the basic and some of the advanced features in Perfect. Additional language features were also documented. The reader is directed to the appendix for some other interesting language features and to the Perfect Reference Manual[2] for an exhaustive description.

A concrete example of program development with Perfect was also given. This demonstrated many of the language features discussed in this paper. It has illustrated how to construct a real program from a requirements stage through the specification of the program and concluded with a refinement step. The example illustrated concepts at a cost of being practical. It did not aim to be a "*real-world*" solution, but was intended as a first interesting class in the Perfect language.

References

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

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

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

A Appendix

A.1 Ghost

A `ghost` expression in Perfect is an expression for which code will not be generated.

A.2 Opaque

An `opaque` function is one that is non-deterministic.

A.3 Deferred

A `deferred` function, is one that is not yet fully defined. It may be defined (with the `define` keyword) in a descendant classes. If a class has any deferred functions, the class cannot be instantiated.

A.4 Constants

A `const` expression denotes constants in Perfect.

A.5 Enumerations

Enumerations provide a means of listing explicitly a set of values. They are declared as part of an `enum` expression.

A.6 Constrained Types

Constrained Types provides a means of defining a class by restriction of an already defined class. A constrained type cannot add new functionality or take advantage of helper functions in its definition.

A.7 Let Expression

A `let` expression creates a read only variable. A value declared in this fashion may be re-evaluated through each iteration of a loop.

A.8 Rank

Perfect enforces an automatic ranking policy on all objects. This is declared as the objects `rank`. The rank may be `same@rank`, `below@rank` or `above@rank`.