Scientific Research

Implementing the Verified Software Initiative Benchmarks using Perfect Developer

Yan Xu, Rosemary Monahan

Department of Computer Science National University of Ireland, Maynooth Maynooth, Ireland; {YAN.XU.2009, Rosemary.Monahan}@nuim.ie

ABSTRACT

This paper describes research on the Perfect Developer tool and its associated programming language, Perfect. We focus on verification benchmarks that have been presented as part of the Verified Software Initiative (VSI), proposing their specification, implementation and verification in the Perfect language and the Perfect Developer tools. To the best of our knowledge this is the first attempt to meet these benchmarks using the Perfect Developer tools. Our aim is to implement the benchmarks and analyze how well the Perfect language can be used to express these benchmarks. In this paper we present the first benchmark, its specification and its verification in the Perfect Developer tool suite.

Keywords: Verification Benchmarks; Specification; Perfect Developer; Verified Software Initiative

1. Introduction

A suite of verification benchmarks for software verification tools and techniques, presented at VSTTE 2008 [1] provides an initial catalogue of benchmark challenges for the Verified Software Initiative. A partial solution, using the RESOLVE [2] specification language, was also presented for the first benchmark in this benchmark suite. The benchmark suite also aims to provide for the evaluation of the state-of-the-art and to provide a medium that allows researchers to illustrate and explain how proposed tools and techniques deal with known pitfalls and well-understood issues, as well as how they can be used to discover and attack new ones. In this paper, we contribute to this evaluation by determining how to express solutions to these benchmarks in the Perfect language [3] and evaluating how its associated verification tools handle their verification. We present our solutions to one of the seven benchmarks problems that we attempted: integer's addition and multiplication. We also present the solution's advantages and drawbacks in Perfect and illustrate the power of the Perfect Developer verification tools.

2. Background

In this chapter, we introduce the Perfect language and the Perfect developer tool.

2.1. Introduction of Perfect Language

Perfect is a specification language following the Design by Contract [4] methodology and similar to the B specification language [5]. Specification languages are used to describe the system at a much higher level than other programming languages during the systems analysis and systems design. Perfect is also an implementation language although implementations are usually translated to more familiar programming languages such as C++ or Java. In addition, Perfect is an Object-Oriented [4] language which supports the specification of programs following the Verified Design by Contract [6] paradigm, where contracts specify the input-output relationship of features of a class, and are verified by static analysis and automated theorem proving to assure that they will not fail at runtime.

We develop and verify programs in the Perfect language by using Perfect Developer. This tool supports the specification, verification and automatic translation of Perfect code to code that is written in either C++, Ada, Java and C#. A full description of the Perfect language is available in the Perfect Developer Reference Manual [7]. A short summary of some of the features of the language follow below.

Type, Expression: In Perfect, identifiers are case sensitive and made up of letters, digits or underscores. The first character of the identifier cannot be a digit. There is no limit length for an identifier, so users can use any length identifier. There are 208 reserved words in Perfect language, such as *change*, *name*, which cannot be used as an identifier. Perfect Language has many predefined basic data types: *anything*, *bool*, *byte*, *char*, *int*, *real*, *void*, *rank*, *nat*, *string.*

Collection: There are three generic collection classes: *set of X*, *bag of X* and *seq of X*. The parameter X stands for whatever class you wish to have a collection of, like *set of int* or *set of Person* (the user defined class). An object of class *set of X* is an unordered collection of ob-

978-1-935068-30-3 © 2010 SciRes.

jects of *class X* and it does not allow duplicates. Class *bag of X* is similar with class set of X, but it permits duplicates. In addition, an object of class *seq of X* is like an ordered *bag of X*.

Class: The class in Perfect is used to mean a set of allowed values which is neither a proper subset of any other Perfect class nor a union of types [7]. It is the basic element for doing the object-oriented development, and it has certain structure, class constructor, variables, invariants and kinds of method. But there is no destructor in Perfect Language. In Perfect, classes are usually divided into several sections: abstract section, internal section, confined section and interface section [8]. The abstract section and interface section are the common used sections. An example of a Person class is shown below.

```
class Person ^=
abstract
    var personname: string,
         age: int,
         gender: gendertype,
    interface
     build { !personname: string, !age: int, !gender: gendertype};
end;
```

The above code defines a new class *Person*. The symbol ^= is pronounced "is defined as" and means the class Person is defined as the notations which are after it and before the word *end;*. The class Person has three attributes: *personname*, *age* and *gender*. They are declared as three variables after *var*. The colon here is pronounced "of type". It is followed by a type expression and it declares the entity before it to be of that type. For example, *gender* is of type *gendertype* which is a user defined type. A constructor starts with the keyword *build* and the exclamation mark*!* means that a change of value occurs in the entity (variable, parameter or self) that precedes it (or occasionally follows it) [8]. Giving the parameter the same name as the corresponding attributes (preceded by a *!*), the attributes of the class Person are automatically initialized.

Function and Schema: In Perfect, there are five kinds of methods: operator, selector, constructor, function and schema [9]. A constructor defines how to build an object of the class. A function takes one or more parameters and yields a result; it has no side-effect. A schema changes the state of the runtime object.

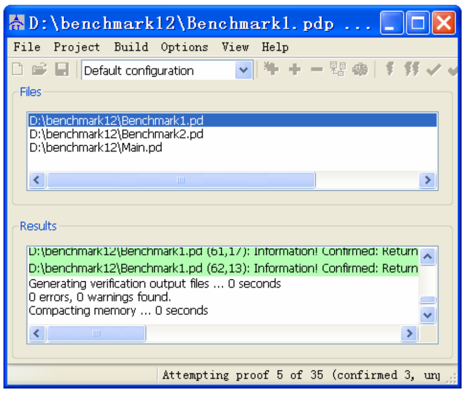
Precondition and Postcondition: We can define the precondition for the constructor, function and schema as restriction, using one or more (comma-separated) Boolean expressions after the key word *pre*. These preconditions must be satisfied whenever the method is called. We also can define the postcondition for the constructor and schemas to express the state of the objects created via the constructors, or to define the changes made by a schema, and (in *after* expressions) to express the value that some expression would have if we made some changes to it [8]. The postcondition defines two things: a frame, which is a set of variables (or parts of variables) that may be changed and a condition that must be satisfied.. The basic form of the postcondition is "*change* frame *satisfy* condition". When we want to change a single variable by making its value equal to some expression, we abbreviate the form *change var satisfy var' = expr* to the form *var! = expr*. In Perfect, the postcondtion specifies precisely which variables have changed and the conditions satisfied by the final values of those variables.

More examples of above features are explained and used in the benchmark implementation in the section 3.

2.2. Introduction of Perfect Developer

Perfect Developer, developed by Escher Technologies Limited, is a software development tool for developing formal specifications and refining them to code [10]. It supports the formal development of object-oriented programs by refinement including formal verification of code. It is a powerful software tool. Not only can it generate ready-to-compile code in C++ or Java from the specifications and refinements expressed in the Perfect Language, but it can also generate the verification conditions needed to verify that the implementation satisfies its specification and prove them.

Perfect Developer is designed for the verification safety-critical applications in industry, and provides a useful tool for teaching formal methods and doing research in universities. Perfect Developer uses a powerful automatic inference engine and theorem prover to reason about the requirements, specifications, and code. So users can import UML models to generate specifications and finally generate the code in Java or C++. Perfect Developer runs standard PCs under both Windows and Linux operating system. A Project Manager interface in Windows is as shown in Fig.1.



```
D:\benchmark12\Benchmark1.pd
D:\benchmark12\Benchmark2.pd
D:\benchmark12\Main.pd
```

```
D:\benchmark12\Benchmark1.pd (61,17): Information! Confirmed: Return
D:\benchmark12\Benchmark1.pd (62,13): Information! Confirmed: Return
Generating verification output files ... 0 seconds
0 errors, 0 warnings found.
Compacting memory ... 0 seconds
```

```
Attempting proof 5 of 35 (confirmed 3, un
```

Figure 1. Project Manager

Through this interface, users can create projects, files, import UML models, build or verify files and users can see the output in the results window.

When using Perfect Developer to do projects, users write code in the Perfect language and then verify this code using the Perfect Developer verifier. This tool includes a compiler for compiling the verified Perfect Language code and automatically translating to code in Java, C++, Ada and C#.

Perfect Developer does not have its own source code editor or C++ compiler or Java compiler, so users need to use the source code editor and the Java JDK or C++ compiler which they installed in their PCs.

3. Solution of Benchmark 1

This section presents the first of the benchmark problems and its solution. This section also gives the result of the verification of this benchmark. In addition, we provide an analysis of the solution for the benchmark.

3.1. Problem Requirements

Using Perfect Language, implement and verify an operation that adds two numbers by repeated incrementing. Then implement and verify an operation that multiplies two numbers by repeated call the adding operation which is implemented before.

3.2. Solution

In this benchmark, there are two parts of the requirement. One is addition specification; the other is the multiplication based on the definition of addition in the first specification.

First we need to implement the addition. The requirement said implementing the adding by repeated incrementing. The basic idea is to pass the two numbers *x*, *y* as input to the add function and loop to get the result. The initial value of the result is the number *x*. In each loop, we add 1 to the previous result. After y times loop, the return result will be the addition of *x* and *y*. Now the problem is how to realize the loop process in Perfect language. According to the introduction reference of Perfect language, there are two ways to do the loop. One is to use the loop statement; another is to call the recursive function. Here we choose using the loop statement to implement our solution. The *add* function code is as follows:

```
function add(x,y:int):int
  ^=x+y
  via
      var addresult:int!=x,
  addy:int!=([y>=0]: y,[]: -y);
      loop
           var j: nat!=0;
           change addresult
      keep addresult'= ([y>=0]: x+j',[]: x-j')
           until j'=addy
           decrease addy-j';
  addresult!= ([y>=0]:addresult+1, []: addresult-1),
           j!=j+1;
       end;
       value addresult
  end;
```

In the code above, we define a function named *add*, which has two input parameters *x* and *y*. The type of *x* and *y* both are integer. The *add* function was defined to return an integer value which equals to the result of *x*+*y* by using the token ^=.

To implement the addition by repeated incrementing, we must refine the add function. The refinement codes are between the word *via* and the last *end*.

First, we define two integer type valuables *addresult* and *addy*, which are initialized to the value of *x* and the absolute value of *y*.

Then we define the loop process, which is between the word *loop* and the first *end*. In the Perfect language, there is a certain structure for the loop statement. At the beginning, we use *var* to define the temp valuable in the loop. Here we define a valuable named *j* and initialize it to 0. Next, we use the word *change* to show which valuable could be changed during the loop, and we let the *addresult* be changeable. The temp valuable *j* doesn't need to declare to be changeable, because it is an inner valuable of the loop. Then we declare the invariant statement which should be maintained during the whole loop. The invariant statement is following the word *keep*. Here the invariant code means that new value of *addresult* should equal to the result of *x*+*j* (if *y*>=0) or the result of *x*-*j* (if *y*<0). Next we set the condition for when the loop should be stopped, by using the word *until*. When the value of *j* equals to the value of *addy*, the program will run out of the loop. The expression *decrease addy-j'* is a loop variant and the PD will attempt to prove that it is never negative unless the loop is about to terminate.

After that, we give the loop body which has two statements here. One is "*addresult != ([y>=0]: addresult+1, []: addresult-1)*", which means the new value of *addresult* will equal to the result of its old value plus one when the *y*>=0 or equal to the result of its old value minus one; the other is set the value of *j* increase one. At the end of the refinement, we set the return value to be *addresult* by following the word *value*. Now the addition function is completed.

In the code above, there are many condition statements for checking whether the value of *y* is less than 0 or not. We use these statements to solve the problem that the input number *y* may be less than 0. In our algorithm for repeated incrementing, we want the loop to run *y* times. But if the value of y is less than 0, there will be a problem. So we improve the algorithm to let the loop run |*y*| times. |*y*| means the absolute value of integer *y*. The return value of this function will be the addition of the two input numbers *x* and *y*, no matter *y* is less than 0 or not.

Now we have the adding function and we need to design solutions to implement the multiplication by using the adding function built by us. The main idea is using the loop to call the add function repeatedly. We define a multiply function which accepts two input numbers x and y, and refine this function to calculate the product of these two input numbers by repeatedly calling the add function. The product of x and y equals the value of repeatedly adding x for |y| times (if y>=0) or its inverse number (if y<0). In the end, this function will return the value of the product. The function code is as follows:

```
  function multi(x,y:int):int
     ^=x*y
     via
                 var  produ:int!=0,  muly:int!=  ([y>=0]:
y,[]: -y);
           loop
                 var j:nat!=0;
                 change produ
                 keep produ'=x*j'
                 until j'=muly
                 decrease muly-j';
                 produ!=add(produ,x),
                 j!=j+1
           end;
           if
           [y>=0]: value produ;
           []: value -produ
           fi
     end;
```

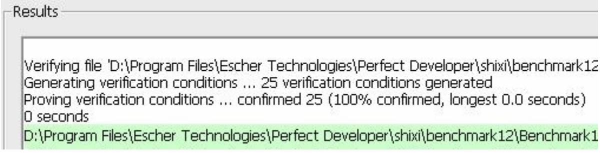
In the *multi* function above, we input two integers *x* and *y* into the function as the parameters and set the type of return value to be integer. The return value of the function is defined to be the product of *x* and *y*. Then we do the refinement of the multiplication by repeatedly calling the adding function between the word *via* and the last *end*.

Firstly, we define two valuables *produ* and *muly*, and initial their value to be 0 and |*y*|. The loop structure is similar to the one in the add function. We define a temp value *j*, initializee it to be 0, and only let the variable *produ* be changeable during the loop. The invariant, which we should maintain in each step of the loop, states
\
that the new *produ* must equal to the product of *x* and the new value of *j*. The loop exits when the value of *j* equals to the *muly*. We use the sentence "*decrease muly-j'*" to make sure the loop will not be infinite. Here the loop body also has two sentences. One is "*produ!=add(produ,x)*", which sets the new value of *produ* by calling *add* function to add the old value of *produ* and *x*; another is same to the one in the add function which is just increasing the *j* by one. Therefore, the program sets the valuable *produ* to be the result of the old value of *produ* plus *x* in each loop iteration and exits the loop after running |*y*| times. In the end, we set the return value depending on whether the *y* is less than 0 or not. So the return value of the multi function will be the product of the input number *x* and *y* and the refinement is completed.

Now we provide the main parts of the solution for the benchmark 1. We build the code in the Perfect Developer and generate the Java code to test running result.

3.3. Verification

So far we've focused on how to structure a class and how to write method specifications. In the Perfect language, we also need to focus on how to use preconditions to specify what needs to hold when a method is called and how to use the verification facility of Perfect Developer to make sure that the specification doesn't involve something untoward such as dividing by zero or indexing off the end of a sequence [8]. We use Perfect Developer 3.12 to verify our Perfect language program and give the result of it here. The screenshot of the verifying the "add" function and "multi" function is as follows:



```
Verifying file 'D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12
Generating verification conditions ... 25 verification conditions generated
Proving verification conditions ... confirmed 25 (100% confirmed, longest 0.0 seconds)
0 seconds
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1
```

Fig.2 Verification Result of Benchmark 1

From the tool feedback, we can see that all the verification conditions are confirmed which are showed by green color. This means that all the conditions of the "add" function and "multi" function can be proved. Feedback is provided from the tool on all verification condition proofs. A sample is: "…\Benchmark1.pd (18,14): Information! Confirmed: Variant non-negative".

3.4. Summary

In conclusion, the implementation of the solution satisfies the requirements of the Benchmark 1. In the implementation, the add function solves the problem that one number adds a negative number, and the multi function also solves the problem of multiplying a negative number. In addition, the add function and multi function are both proved successfully in the verification. The Java code generated by the Perfect Developer from the Perfect code has been tested and is without error. We also find that the implementation of the solution has some drawbacks. The add function runs for a long time due to the loop process for adding, however, the benchmark requires that the implementation of adding must be the repeated increment. The multi function also has the same problem due to the requirement that the implementation of multiplying must be the repeated calling the add function. Further analysis follows in section 4.

4. Analysis

In this section, we analyze the result of the implementation and give some suggestions to improve it.

Generic: The implementation of this benchmark is not generic. The *add* function and the *multi* function of the implementation can only deal with integer values. It will be more useful if the function is generic to more types. The Perfect Language supports the generic class. For example, we can define a generic Queue class as follows:

```
class Queue of X ^=
abstract
        var queue: seq of X, size:int;
  interface
        function head:X
        …
        schema !Enqueue(y:X)
        …
  end;
```

This Queue class accepts type parameterization. We can use it to build the Queue of the built-in type or self defined type. In further research, we will continue to make the implementations of other benchmarks generic.

Reusable: The solution of benchmark 1 implemented here can be reused to generate a new benchmark solution. Building the solution to a new benchmark by using the earlier benchmarks' solution can show the capabilities of the earlier benchmarks. The reusability is one of the most significant features of these benchmarks. In the further research, we will focus on how to generate the benchmark from the earlier benchmarks solutions by reusing earlier benchmark solutions..

Schema: A deliberate design decision in Perfect is that expressions are side-effect free. This allows all expressions to be used in specification statements. The effect of a function with side-effects can be achieved by defining a schema with an out-parameter. The out-parameter is used to pass back what would be the "return value" if it were a function.

5. Conclusion

This paper proposes the solution of implementing and verifying one VSI benchmark using the Perfect language. After implement the benchmark, we also verify the solution and analyze how well the Perfect language can be done in this benchmark. Our solutions to the other benchmarks are available in the M.Sc Dissertation "Implementing the Verified Software Initiative Benchmarks using Perfect Developer" [11]. Our work may assist the Verified Software Initiative by providing solutions to the proposed verification benchmarks. We will continue to research on these benchmarks, optimize the implementation of them and use them to build a Perfect application program in the future.

6. Acknowledgements

The authors appreciate the ideas from Bruce W. Wide, Murali Sitaraman, Heaher K. Harton, Bruce Adcock, Paolo Bucci, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum and David Frazier, the authors of the paper "Incremental Benchmarks for Software Verification Tools and Techniques" as well as the work of those at Escher Technologies Ltd. who are responsible for the production of Perfect Developer. Many thanks to David Crocker, Escher Technologies, for his ongoing support and helpful feedback. Thanks also to the reviewers for their helpful comments.

References

[1] Bruce W. Weide, Murali Sitaraman, Heather K. Harton, Bruce Adcock, Paolo Bucci, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum and David Frazier (2008).: Incremental Benchmarks for Software Verification Tools and Techniques. Technical Report RSRG-08-02

[2] Edwards, S.H., Heym, W.D., Long, T.J., Sitaraman, M., Weide, B.W.: Specifying components in RESOLVE. Software Engineering Notes **19**(4) (1994) 29-39.

[3] Gareth Carter.: Introducing the Perfect Language. 2005.

[4] Bertrand Meyer, Object Oriented Software Construction , 2nd Edition.

[5] J. R. Abrial, The B-book: assigning programs to meanings 1996 Cambridge University Press.

[6] Escher Technologies. What is Verified Design-by-Contract? http://www.eschertech.com/products/verified_dbc.php (Aug 2010)

[7] Escher Technologies. The Perfect Developer Language ReferenceManual Version 3.12.

[8] Escher Technologies. Perfect Developer: the Basic Tutorials. http://www.eschertech.com/tutorial

978-1-935068-30-3 © 2010 SciRes.

/tutorials.htm (Aug 2010)

[9] Gareth Carter, Rosemary Monahan, Joseph M. Morris (2005).: Software Refinement with Perfect Developer. ISBN:0-7695-2435-4

[10] David Crocker. Perfect Developer: A tool for Object-Oriented Formal Specification and Refinement. In FME 2003, Tools Exhibition Notes.

[11] Yan Xu, Implementing the Verified Software Initiative Benchmarks using Perfect Developer, M.Sc Dissertation, Dept. of Computer Science, NUIM (January 2010).