Implementing the Verified Software Initiative Benchmarks using Perfect Developer

Yan Xu

Final Year Project – 2010

M.Sc. in Computer Science and Software Engineering

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Department of Computer Science,

National University of Ireland, Maynooth (Ireland)

Supervisor: Rosemary Monahan

A thesis submitted in partial fulfillment of the requirements for the M.Sc. Computer Science and Software Engineering

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master of Science in Software Engineering, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: _______________ Date: _________________

Acknowledgments

I would like to take this opportunity to thank my supervisor, Rosemary Monahan, for giving me a lot of help during the entire project. I also would like to thank Dr. Diarmuid O' Donoghue, who helps me on work placement and dissertation.

Implementing the Verified Software Initiative Benchmarks using Perfect Developer

Yan Xu

Department of Computer Science,
National University of Ireland, Maynooth (Ireland)
YAN.XU.2009@nuim.ie

Supervisor: Rosemary Monahan

Abstract

This paper describes research on the Perfect Developer tool and its associated programming language, Perfect. We focus on seven 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 and the first full presentation of solutions to these benchmarks in any verification support tool. Our aim is to implement the benchmarks and analyse how well the Perfect language can be used to express these benchmarks. Furthermore, we provide suggestions on how to make the Perfect language and Perfect Developer better.

Table of Contents

1. Introduction

This paper describes research on the Perfect Developer tool for programming verification language. A number of verification-benchmark programs have been proposed in the Clemson University Researchers' paper [1] with only one solution suggested. This single solution is written in the RESOLVE [2] specification language. In this paper we target seven of these benchmarks, to determine how they can be expressed in the Perfect language [3]. The benchmark problems cover several classic algorithms such as binary search, queue sorting and represent some complicated data structures like Queue, Map and Linked-List. These algorithms and data structures are all very useful in assessing the expressiveness of the Perfect language and the power of the Perfect Developer verification tools.

Perfect is a specification language and also an implementation language. In this paper, we present our solutions to seven of these benchmark problems. They show that we could implement and verify the benchmarks in Perfect. And we also discuss the solutions' advantages and drawbacks in Perfect.

The structure of this paper:

- Section 1 provides an overview of this paper.
- Section 2 introduces the Perfect Language and the tool Perfect Developer. It also compares the Perfect language with the Spec# programming language.
- Section 3 presents the work we have done on those benchmarks. In each benchmark subsection, we will give the requirement and the design idea, process and solutions. It also includes the problems we met during the work and how we dealt with these problems.
- Section 4 summaries assessment of the expressiveness of the Perfect language and the power of the Perfect Developer verification tools.
- Section 5 summarizes our work and presents our conclusions and suggests some future work for improving verification tools.

2. Background

In this chapter, we introduce the Perfect language and the Perfect developer. Furthermore, we discuss the difference between the Perfect language and Spec# language [4].

2.1 Introduction of Perfect Language

Perfect language is a specification language based on Object-Oriented Paradigm [5]. It acts both as a specification language and an implementation language. It defines some basic types which can be used to specify abstract data types. It also supports the object oriented features like classes, objects and inheritance. The primary strategy of Perfect language is Design by Contract which uses contracts to specify the input-output relationship of features of a class.

We develop and verify Perfect language by using Perfect Developer. The code can be generated automatically from Perfect language to C++, Ada, Java and C# by Perfect Developer. Now we introduce the Perfect language as follows:

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 [6] in Perfect language, such as “change”, “name”. They cannot be used as an identifier. Perfect Language has many basic data types: **anything**, **bool**, **byte**, **char**, **int**, **real**, **void**, **rank**, **nat**, **string**. The first 8 types are predefined classes; the last 2 types are predefined types.

Collection: There are another three types which called 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 in or set of Person (the user defined class). An object of class set of X is an unordered collection of objects of class X and this collection does not allow duplicates. Class bag of X is similar with class set of X, but it permits duplicates. An object of class seq of X is like the class bag of X, but it is ordered.

Class: The class 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]. The class is the basic element which is used to do the object-oriented development. The class in Perfect 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 sample 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 identifier after “class” is the class name:”Person”. The symbol “^=” is pronounced “is defined as” and means something is defined as something. Here it means the class Person is defined as the notations which are after it and before the “end;” The class Person has three attributes: personname, age and gender. They are declared as three variables after **var**. The colon between personname and string is pronounced “of type”. It is followed by a type expression and it declares the entity before it to be of that type. Here “personname’ is of type **string**, “age” is of type **int** and “gender” is of type gendertype which is a user defined type. The class Person has a constructor, which is the declaration that starts with the keyword **build**. In Perfect, 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]. Because of giving each parameter the same name as each attributes and before parameter with an exclamation mark, the attributes of the class Person will be initialized directly from the corresponding parameters. More examples of class are shown in the section 3.

Function and Schema: In Perfect, there are three kinds of methods: 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. The examples of function and schema in Perfect Language are shown in the section 3.

Precondition and Postcondition: We can define the precondition for the constructor, function and schema. The preconditions act as restriction, which are the 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 schema. We use the postcondition to define the objects created by constructors, 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]. In Perfect, the postcondition defines two things: a frame, which is a set of variables (or parts of variables) that may be changed; a condition to be satisfied. The basic form of 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 be "`var! = expr`". In Perfect, the postcondtion specifies precisely which variables have changed and the conditions satisfied by the final values of those variables. The examples of precondition and postcondition are shown in the section 3.

There are more features of the Perfect Language. We explain them in the implementations of benchmarks in section 3.

2.2 Introduction of Perfect Developer

Perfect Developer 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 and includes formal verification of code. Perfect Developer from Escher Technologies Limited is a powerful software tool. It can generate ready-to-compile code in c++ or Java from the specifications and refinements in Perfect Language [7]; it can also generate the verification conditions for the software system and automatically prove these verification conditions. Perfect Developer is based around object-oriented design because it is the popular paradigm which wildly used in the industry today.

Perfect Developer can be used for the safety-critical applications or other applications, and also be used for teaching formal methods or doing research in universities. Because Perfect Developer is a high productivity software tool and uses advanced automated reasoning in the prover [10]. A lot of specifications can be automatically refined to code, which reduce the mount of code has to be written. In addition, Perfect Developer uses a powerful automatic inference engine and theorem prover to reason about the requirements, specifications, and code [11]. So users don't need to have advanced mathematic knowledge. The Perfect Developer tool can import UML models to generate specifications and finally generate the code in Java or C++. These are of great benefit to teaching software development in Java and modeling by using UML in universities. Perfect Developer runs standard PCs under both Windows and Linux. The interface to use Perfect Developer is via the Project Manager. (When the Project Manager runs under windows, it will appear as shown below (Fig.1))

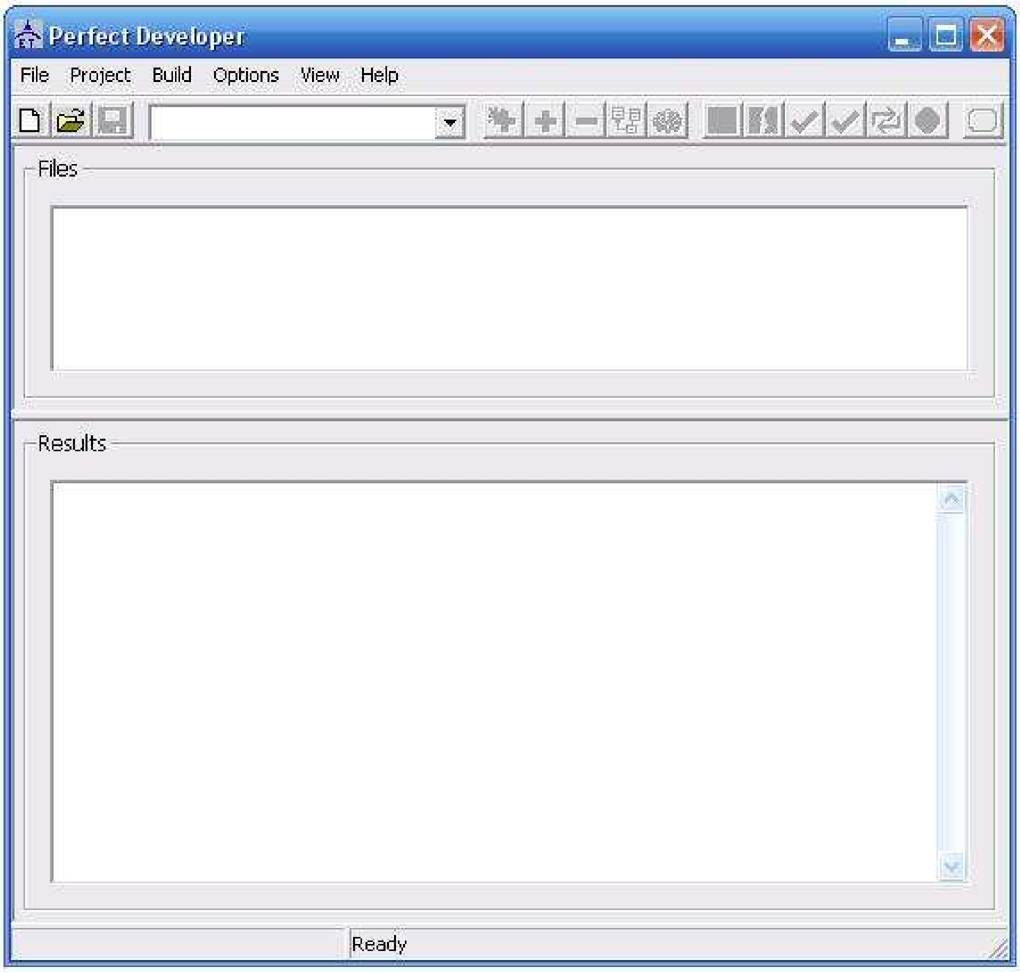


Fig.1 Project Manager

Through this interface, users can create projects, files, import UML models, build or verify files and users can see the output in the results area. The building result example is shown in the Appendix B.

The powerful tool Perfect Developer supports writing specifications, verifying these specifications, refining them to code within the same notation, verifying these refinements, and translating them automatically to code in C++ or Java language [12]. When using Perfect Developer to do projects, users write code in the Perfect Language and then verify this code using the verifier which is included in the Perfect Developer. Perfect Developer also includes a compiler for compiling the Perfect

Language code and generating the Java, C++ or Ada95 code.

Perfect Developer does not have its own source code editor, so users need to make configuration in Perfect Developer’s project manager to choose the source code editor which you installed in your PC. There are a lot of editors suitable for the Perfect Developer. So you can choose one which you like. Perfect Developer neither includes C++ compiler nor Java compiler. So users need to configure it in Project Manager and use the Java JDK or C++ compiler installed in their PCs.

2.3 Comparison between Perfect Language and Spec# Language

In this subsection, we compare the Perfect language and Spec# language and discuss the good points and drawbacks of them.

The Perfect Language provides the Verified Design By Contract [13] while the spec# language supports the Design By Contract [14]. They are both specification language [15]. They both specify the Object-Oriented Program and they have verifiers to check the programs whether satisfy their specifications.

But there are many differences between the Perfect Language and Spec# Language.

First, the Perfect Language is not only a specification language but also an implementation language. This makes the Perfect Language more complete, which means that the Perfect Language can specifie precisely what variables are changed and how they are changed. In contrast, the spec# language only specifies something that must be true when the method returns. For example, the postcondition in Perfect language defines what variables are changed and also defines the way of changing them.

Second, the Perfect Language can implement the whole program. In contrast, the Spec# language is used to add assertions to the C# programming language [16]. The Perfect language has its own class, methods, library and the rules of programming. The Spec# language is based on the C# language. But the programming of the Perfect language is not very flexible. In some conditions, the Perfect Language can not implement the requirements. For example, the Perfect Language doesn’t support the pointer or reference which are commonly supported by other language.

Third, the Perfect Language could be automatically translated to code in Java, C++, Ada and C# (which is a new features in the Perfect Developer version 4.0) by using Perfect Developer. When the program of Perfect Language is verified successfully, it can be changed to the programs in Java, C++ and Ada. The generated programs are also verified by the specification. So we can generate the bug-free program by using the Perfect Language.

3. Benchmarks and Solutions

This section presents all the benchmark problems from Benchmark 1 to Benchmark 7 and the solutions of them. This section also gives the results of the verifications of each benchmark. In addition, we give the analysis of the solution for the benchmark at the end of each subsection.

3.1 Benchmark 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.

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. So the basic idea is to input the two numbers x, y as parameters into the add function and do the loop to get the result. In each loop, we increment 1 to the old result. The first result is the number x. After y times loop, the return result will be the addition of x and y. Now the problem we need to solve is how to realize the loop process in Perfect language. According to the material 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 in 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;
        assert addresult=x+y;
        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 will return an integer value, and we define the return value equals to the result of "x+y" by using the token "`^=`". To implement the addition by repeated incrementing, we need 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". The "addresult" is initialized to the value of x, and the "addy" is initialized to the absolute value of y.

Then we define the loop process, which is between the word **loop** and the first **end**. In perfect language, there is a certain structure for the loop statement. First we use the **var** to define the temp valuable in the loop. Here we define a valuable named "j" and initial it to 0. Second we use the word **change** to show which valuable could be changed during the loop. Here 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. And we also use "**`decrease`** `addy-j'`" to make sure that the loop is not an infinite loop. The loop will be stopped after "`addy-j'`" times looping. After decrease statement, 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.

After the loop process, we use the word assert to declare the invariant statement "`addresult=x+y`" which should be maintained during the refinement. At the end of the refinement, we set the return value to be "addyresult" by following the word **value**. Now the add function for addition is completed.

In the code above, there are many condition statements for checking the value of y is less than 0 or not. We use these statements to solve the problem which is: the input number y may be less than 0. In our algorithm for repeated incrementing, we want the loop will 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 result 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 which we just built. The main idea is using the loop to perform repeated calling the add function. We define a multiply function which accept two input numbers x and y, refine this function to calculate the product of these two input numbers by repeated calling the add function. The product of x and y equals to the value of repeated adding x for |y| times (if y>=0) or the inverse number of it (if y<0). In the end, this function will return the value of the product. The multi 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;
        assert ([y>=0]: produ=x*y,[]: produ= -(x*y));
        if
        [y>=0]: value produ;
        []: value -produ
        fi
    end;
```

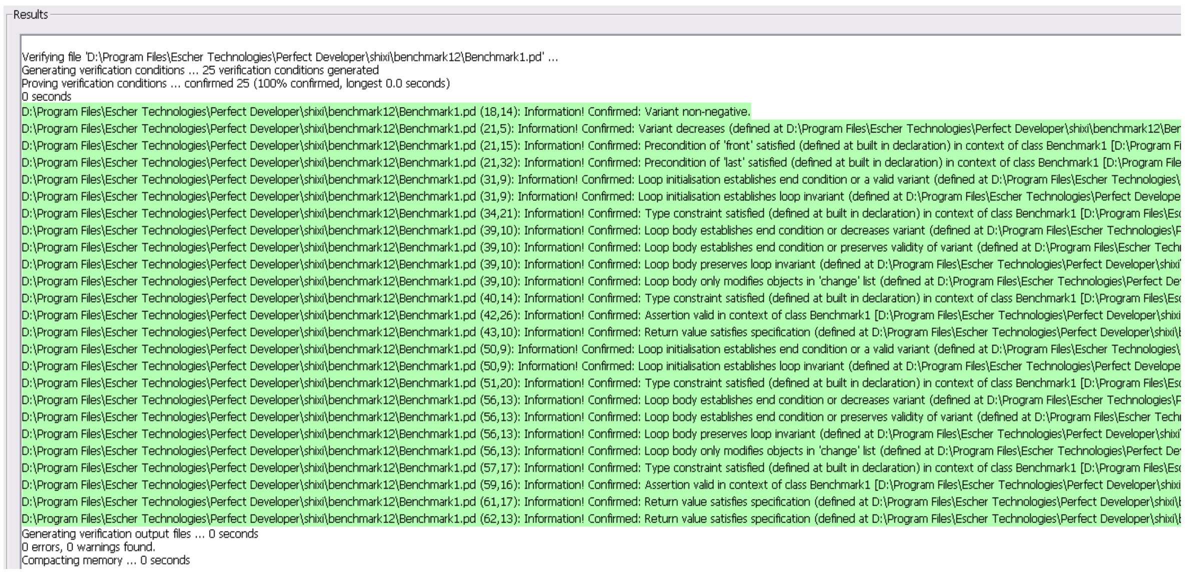
In the multi function above, we input two integers x and y into the function as the parameters and set the type of return value to be integer. The return value of the function is defined to be the product of x and y. Then we do the refinement of the multiplication by repeated calling the adding function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

First 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” and initial it to be 0. And we only let the “produ” to be changeable during the loop. The invariant which we should maintain in each step of the loop is the new “produ” must equal to the product of the 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)`”. It 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. So in the loop process, the program sets the valuable “produ” to be the result of the old value of “produ” plus “x” in each time and exit the loop after running |y| times. In the end, we use assert sentence to make sure the invariant is always maintained after the loop and 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.

So now we give the main parts of the solution for the benchmark 1. We build the code in the Perfect Developer and generate the Java code to run.

Verification: So far we've focused on how to structure a class and how to write method specifications. In 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 for something untoward such as dividing by zero or indexing off the end of a sequence [8]. In this subsection we verify our Perfect language program and give the result of it here. The Perfect Developer which we used to verify is of the version 3.12.

The screenshot of the verifying the “add” function and “multi” function is as follows:



```
Results

Verifying file 'D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd' ...
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.pd (18,14): Information! Confirmed: Variant non-negative.
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (21,5): Information! Confirmed: Variant decreases (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Ber
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (21,15): Information! Confirmed: Precondition of 'front' satisfied (defined at built in declaration) in context of class Benchmark1 [D:\Program Fi
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (21,32): Information! Confirmed: Precondition of 'last' satisfied (defined at built in declaration) in context of class Benchmark1 [D:\Program File
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (31,9): Information! Confirmed: Loop initialisation establishes end condition or a valid variant (defined at D:\Program Files\Escher Technologies\
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (31,9): Information! Confirmed: Loop initialisation establishes loop invariant (defined at D:\Program Files\Escher Technologies\Perfect Develope
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (34,21): Information! Confirmed: Type constraint satisfied (defined at built in declaration) in context of class Benchmark1 [D:\Program Files\Esc
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (39,10): Information! Confirmed: Loop body establishes end condition or decreases variant (defined at D:\Program Files\Escher Technologies\P
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (39,10): Information! Confirmed: Loop body establishes end condition or preserves validity of variant (defined at D:\Program Files\Escher Techn
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (39,10): Information! Confirmed: Loop body preserves loop invariant (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (39,10): Information! Confirmed: Loop body only modifies objects in 'change' list (defined at D:\Program Files\Escher Technologies\Perfect De
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (40,14): Information! Confirmed: Type constraint satisfied (defined at built in declaration) in context of class Benchmark1 [D:\Program Files\Esc
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (42,26): Information! Confirmed: Assertion valid in context of class Benchmark1 [D:\Program Files\Escher Technologies\Perfect Developer\shixi
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (43,10): Information! Confirmed: Return value satisfies specification (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\b
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (50,9): Information! Confirmed: Loop initialisation establishes end condition or a valid variant (defined at D:\Program Files\Escher Technologies\
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (50,9): Information! Confirmed: Loop initialisation establishes loop invariant (defined at D:\Program Files\Escher Technologies\Perfect Develope
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (51,20): Information! Confirmed: Type constraint satisfied (defined at built in declaration) in context of class Benchmark1 [D:\Program Files\Esc
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (56,13): Information! Confirmed: Loop body establishes end condition or decreases variant (defined at D:\Program Files\Escher Technologies\P
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (56,13): Information! Confirmed: Loop body establishes end condition or preserves validity of variant (defined at D:\Program Files\Escher Techn
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (56,13): Information! Confirmed: Loop body preserves loop invariant (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (56,13): Information! Confirmed: Loop body only modifies objects in 'change' list (defined at D:\Program Files\Escher Technologies\Perfect De
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (57,17): Information! Confirmed: Type constraint satisfied (defined at built in declaration) in context of class Benchmark1 [D:\Program Files\Esc
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (59,16): Information! Confirmed: Assertion valid in context of class Benchmark1 [D:\Program Files\Escher Technologies\Perfect Developer\shixi
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (61,17): Information! Confirmed: Return value satisfies specification (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\b
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark1.pd (62,13): Information! Confirmed: Return value satisfies specification (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\b
Generating verification output files ... 0 seconds
0 errors, 0 warnings found.
Compacting memory ... 0 seconds
```

Fig.2 Verification Result of Benchmark 1

From the screenshot, 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.

Summary: In this sub section, we use table to show the good points and drawbacks of this benchmark’s solution.

Summary of Benchmark 1’s Solution	
Good points	Drawbacks
1. The add function solves the problem that one number adds a negative number.	1. The running time of add function is too long. This is due to using the loop process for adding. This problem can not be solved, because the benchmark requires the implementation of adding should be the repeated increment.
2. The multi function also solves the problem that one number multiplies a negative number.	2. The running time of multi function is too long. This is due to using the loop process for multiplying. This problem can not be solved, because the benchmark requires the implementation of

	multiplying should be the repeated calling the add function.
3. The add function and multi function are both proved ok in the verification.	

Table 1: Summary of Benchmark 1's Solution

3.2 Benchmark 2

Problem Requirements: Implement and verify an operation by using binary search algorithm [17] to find the entry of an input data in a sorted sequence in Perfect language.

Solution: According to the requirements of this benchmark, we need to perform the binary searching in Perfect language. The binary search algorithm is explained very clearly in the reference article. So in this subsection we just focus on the implementation of binary search in Perfect. First we give an array which contains N elements, and the array is increasingly ordered on the values of its elements. Then we use the binary search algorithm to find the index of the element whose value equals to the input parameter "b". In the end we get the index if the input parameter "b" is in the array or return -1 if "b" is not in there. There are several types of implementing the binary search, like iterative, recursive etc. Here we use the recursive to implement the binary search which is the most straightforward implementation. A recursive function named "bisearch" is built to perform the binary search algorithm. The "bisearch" function code is as follows:

```
function bisearch(a: seq of int, b:int, low:int, high:int):int
    pre  #a~=0, forall i::(1..((#a)-1)) :- a[i-1]<=a[i],
0<=low<=high<=(#a-1), forall e::a :- e#a=1
    decrease (high-low)
    ^=([b in a]:a.findFirst(b),[]: -1)
    via
        if
        [(high~=low)&a[(low+high)/2]>b]:  value bisearch(a, b,
        low, (low+high)/2);
        [(high~=low)&a[(low+high)/2]<b]:  value bisearch(a,b,
        ((low+high)/2)+1,high);
        [a[(low+high)/2]=b]: value (low+high)/2;
        [(high=low)&a[(low+high)/2]~=b]: value -1
        fi
    end;
```

In this function, it takes four input parameters: "a" is a sequence of integer which is used to perform the array in Perfect; "b" is an integer valuable for searching; "low" is an integer valuable which presents the most left index of the ordered sequence "a"; "high" is an integer valuable which presents the most right index of the ordered sequence “a”.

Before running this function, these parameters need to satisfy some preconditions which are following the word **pre**. First the input sequence “a” must not be an empty sequence, which also means the length of the sequence “a” should not be 0. In Perfect, it can be expressed as “`#a~=0`”. Second the elements of the sequence are sorted in the increasing order on their value, which can be expressed as each element in the sequence is larger than the previous one. To achieve this precondition, we use the Perfect’s built-in expression for the operation on collection which is in the form like:

```
        forall identifier::collection :- condition
```

This expression yields true is all the elements of collection satisfy the condition, or the collection is empty. To ensure each element is larger than the previous element in the sequence, we use the expression “`forall i::(1..((#a)-1)):- a[i-1]<=a[i]`”. Here the collection is the integers from 1 to (#a)-1, the #a means the length of the sequence “a”. This expression yields **true** if each integer “i” from 1 to #a-1 satisfies the condition “`a[i-1]<=a[i]`”. So if this expression yields **true**, it ensures that each element is larger than the previous element in the sequence “a”. Next the value of input “low” should be less than the value of input “high”, and they are both in the domain of sequence “a”. At the end of the precondition, we use the expression “`forall e::a :- e#a=1`” to ensure that each element in the sequence is unique. The condition “e#a=1” means the the frequency of each element “e” in the collection “a” equals to one. In another words, each element “e” is unique in the collection “a”.

After precondition, keyword **decrease** and the expression “`(high-low)`” followed compose the variant part which is used to guarantee termination in this recursive function. The expression “`(high-low)`” is guaranteed to decrease on every recursion but never to become negative. Then we set the return value of the function to be the index of the element which has the same value as “b” in the sequence “a” if “b” is an element of “a”, or to be -1 if “b” is not an element of “a”.

In Perfect language, the recursive function is also implemented by the refinement. The refinement part is from the word **via** to the last **end**. In the recursive version of binary search, it recursively searches the subrange between “low” and “high”. If the middle element of the subrange equals to the input “b”, function will return the index of this element. Otherwise, the function will call itself recursively and set the new parameters. If the middle element is larger than “b”, we set the new “high” to be the index of the middle element which is “`(low+high)/2`”; if the middle element is less than “b”, we set the new “low” to be the index of the middle element. And if the value of “low” equals to the value of “high”, we will return -1 to indicate that the key word “b” cannot be found in the collection “a”. To perform this in Perfect, we use the conditional expressions in refinement. There are four conditions of the refinement of this post-condition.

a) If “low” isn’t equal to “high” and the middle element is larger than the input “b”, then call the bisearch function and pass then new parameters which replaces the “high” with the “`(low+high)/2`”. The code in Perfect is as follows:

```
“[(high~=low)&a[(low+high)/2]>b]:  value bisearch(a, b, low,
```

`(low+high)/2);`”

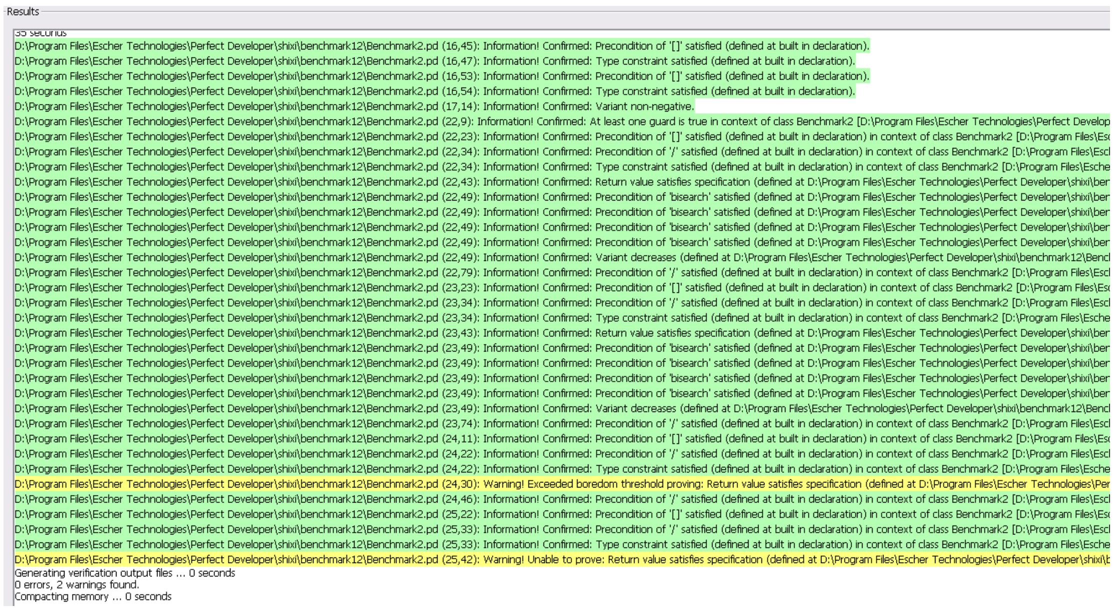
b) If “low” isn’t equal to “high” and the middle element is less than the input “b”, then call the bisearch function and pass then new parameters which replaces the “low” with the “`(low+high)/2`”. The code in Perfect is as follows:
“`[(high~=low)&a[(low+high)/2]<b]:  value bisearch(a,b, ((low+high)/2)+1, high);`”

c) If the middle element equals to the input “b”, then the function return the value of “`(low+high)/2`” which is the index we are searching for. And the function will stop recursively calling. The code in Perfect is as follows:
“`[a[(low+high)/2]=b]: value (low+high)/2;`”

d) If “low” equals to “high” and the middle element is not equal to “b”, then return value -1 which means the input “b” is not in the sequence. And the function stop recursively calling. The code in Perfect is as follows:
“`[(high=low)&a[(low+high)/2]~=b]: value -1`”.

The problem we met during the implement is how to define the return value of this function before its refinement. We want this function return the index of the element which equals to the input “b” or -1. How to use the input parameters to define this return value is a problem. The built-in method of Perfect collection solves this problem. The input parameter “a” is type of sequence which has a method named “findFirst(x)”. This method returns the index of the leftmost occurrence of x in sequence. So the return value of bisearch function can be defined by using this method. And now we have given the solution of benchmark for binary search in Perfect.

Verification: Here we verify the solution program in the Perfect Develop to prove all the conditions in it. The result screenshot is as follows:



```
Results
35 seconds
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (16,45): Information! Confirmed: Precondition of '[]' satisfied (defined at built in declaration).
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (16,47): Information! Confirmed: Type constraint satisfied (defined at built in declaration).
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (16,53): Information! Confirmed: Precondition of '[]' satisfied (defined at built in declaration).
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (16,54): Information! Confirmed: Type constraint satisfied (defined at built in declaration).
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (17,14): Information! Confirmed: Variant non-negative.
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (22,9): Information! Confirmed: At least one guard is true in context of class Benchmark2 [D:\Program Files\Escher Technologies\Perfect Develop
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (22,23): Information! Confirmed: Precondition of '[]' satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Es
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (22,34): Information! Confirmed: Precondition of '/' satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Escl
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (22,34): Information! Confirmed: Type constraint satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Esche
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (22,43): Information! Confirmed: Return value satisfies specification (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\ber
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (22,49): Information! Confirmed: Precondition of 'bisearch' satisfied (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\ber
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (22,49): Information! Confirmed: Precondition of 'bisearch' satisfied (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\ber
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (22,49): Information! Confirmed: Precondition of 'bisearch' satisfied (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\ber
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (22,49): Information! Confirmed: Precondition of 'bisearch' satisfied (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\ber
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (22,49): Information! Confirmed: Variant decreases (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Bencl
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (22,79): Information! Confirmed: Precondition of '/' satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Escl
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (23,23): Information! Confirmed: Precondition of '[]' satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Es
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (23,34): Information! Confirmed: Precondition of '/' satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Escl
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (23,34): Information! Confirmed: Type constraint satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Esche
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (23,43): Information! Confirmed: Return value satisfies specification (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\ber
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (23,49): Information! Confirmed: Precondition of 'bisearch' satisfied (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\ber
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (23,49): Information! Confirmed: Precondition of 'bisearch' satisfied (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\ber
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (23,49): Information! Confirmed: Precondition of 'bisearch' satisfied (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\ber
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (23,49): Information! Confirmed: Precondition of 'bisearch' satisfied (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\ber
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (23,49): Information! Confirmed: Variant decreases (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Bencl
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (23,74): Information! Confirmed: Precondition of '/' satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Escl
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (24,11): Information! Confirmed: Precondition of '[]' satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Es
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (24,22): Information! Confirmed: Precondition of '/' satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Escl
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (24,22): Information! Confirmed: Type constraint satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Esche
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (24,30): Warning! Exceeded boredom threshold proving: Return value satisfies specification (defined at D:\Program Files\Escher Technologies\Per
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (24,46): Information! Confirmed: Precondition of '/' satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Escl
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (25,22): Information! Confirmed: Precondition of '[]' satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Es
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (25,33): Information! Confirmed: Precondition of '/' satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Escl
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (25,33): Information! Confirmed: Type constraint satisfied (defined at built in declaration) in context of class Benchmark2 [D:\Program Files\Esche
D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (25,42): Warning! Unable to prove: Return value satisfies specification (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\b
Generating verification output files ... 0 seconds
0 errors, 2 warnings found.
Compacting memory ... 0 seconds
```

Fig.3 Verification Result of Benchmark 2

The verification result has two warnings:

The first warning is "`D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (24,30): Warning! Exceeded boredom threshold proving: Return value satisfies specification (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (18,7)) in context of class Benchmark2 [D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (7,1)] (see D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2_unproven.htm#5), cannot prove: ((low + high) / 2) = ([b in a]: a.findFirst(b), []: -1).`"

This warning means the code in the line 24 position 30 of the Benchmark2.pd can not be proved. The code in that position is: "`[a[(low+high)/2]=b]: value (low+high)/2`". The prover said it cannot prove "`((low + high) / 2) = ([b in a]: a.findFirst(b), []: -1)`". But we can see the function returns the "`(low+high)/2`" only when "`a[(low+high)/2]=b`" is satisfied. So the problem here is the prover thinks the condition "`a[(low+high)/2]=b`" doesn't equal to the condition "`((low + high) / 2) = [b in a]: a.findFirst(b)`". This warning still cannot be solved.

The second warning is "`D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (25,42): Warning! Unable to prove: Return value satisfies specification (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (18,7)) in context of class Benchmark2 [D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2.pd (7,1)] (see D:\Program Files\Escher Technologies\Perfect Developer\shixi\benchmark12\Benchmark2_unproven.htm#1), cannot prove: -1 = ([b in a]: a.findFirst(b), []: -1).`"

This warning means the code in the line 25 position 42 of the Benchmar2.pd can not be proved. The code in this position is: "`[(high=low)&a[(low+high)/2]~=b]: value -1`". The problem is like the first one. The prover think the condtion "`[b ~in a]: -1`" not equal to the condtion "`[(high=low)&a[(low+high)/2]~=b]: value -1`". This warning still cannot be solved.

Summary: We list the good points and drawbacks of the solution for the benchmark 2 in the following table.

Summary of Benchmark 2's Solution	
Good points	Drawbacks
1. Directly implement the binary search in the Perfect language and it works fine.	1. Not all the conditions can be proved in the program.
	2. The binary search is only for the integer type sequence.

Table 2: Summary of Benchmark 2's Solution

3.3 Benchmark 3

Problem Requirements: Sort an ordered queue in Perfect language. Specify a queue ADT which is generated by the built-in types in Perfect language. Verify and implement the sort method for this queue.

Solution: In this benchmark, the main job is to realize the sort algorithm for a queue in Perfect language. There are two parts of this benchmark. One is building a queue class by the other built-in types in Perfect; another is implementing the sort algorithm on the queue ADT in Perfect.

First we solve the problem of building the queue class. A queue ADT [18] is a collection of elements and has some operations such as head (return head element), enqueue(input a new element into the queue), dequeue(remove the head of the queue) and isempty(check whether the queue is empty) etc. To perform this in Perfect, we need to use the built-in types to generate a new class named Queue and implement the methods for it. The "Queue" class code is as follows:

```
class Queue^=
abstract
    var myqueue: seq of int;
interface
    function myqueue;
    function head:int
        pre #myqueue>0
        ^=myqueue.head;
    function getLength:int
        ^=#myqueue;
    function findmax:int
        pre #myqueue>0
        ^=myqueue.max;
    schema !Enqueue(input:int)
       post ([input ~in myqueue]: myqueue!=myqueue.append(input),
       []: pass);
    schema !Dequeue
        pre #myqueue>0
        post myqueue!=myqueue.tail;
    build{}
        post myqueue!=seq of int{};
    build{inputseq: seq of int}
        post myqueue!=inputseq;
end;
```

Class Queue has a variable named "`myqueue`" which is used to store the elements of Queue. It is a sequence of integer elements. Class Queue also has four functions, two schemas and two constructors. The function "`myqueue`" makes the variable "myqueue" accessible. The function "head" returns the value of the first element of the Queue by using the built-in method head. The function "getLength" gets the value of the length of the Queue. The function "findmax" yields the maximum element of the Queue by calling the built-in method "myqueue.max". The schema "Enqueue" appends the input element to the sequence if the element is not in it. This schema calls the method append of the collection sequence to add the input element. The schema "Dequeue" removes the head element from the queue. The method "myqueue.tail" returns the sequence "myqueue" with the "myqueue.head" removed. There are two constructors for the Queue class. One has no input parameters; the other has one parameter for assigning the value of myqueue.

Next we need to build a function to sort the Queue which we just defined. This function should sort the input Queue and return the sorted Queue. There are several sort algorithms [19]. The main idea of the sorting algorithm we used here is: First find the max element in the input Queue1 and compare the head element of Queue1 with the max element. If they are not equal, remove the head element of Queue1 and then add it to the end of Queue1; if they are equal, remove the head of Queue1 and add it to the end of Queue2. Repeat this process until the Queue1 only left one element and then return the Queue2. Now we build a recursive function named "SortQueue" to implement the above sort algorithm. The code of function "SortQueue" is as follows:

```
function SortQueue(Q1: Queue, Q2:Queue):Queue
    pre Q1.getLength>0
    decrease Q1.getLength
    satisfy
    forall i::(0..((Q1.getLength)-2)):-result.myqueue[i] <=
result.myqueue[i+1]
    via
    let max^=Q1.findmax;
    let temp^=Q1.head;
    let Q2T ^=Q2;
    let QT^=Q2 after it!Enqueue(Q1.head);
    let Q1T ^=Q1 after it!Dequeue;
    let Q1TT ^=Q1 after it!Dequeue then it!Enqueue(temp);
        if
        [Q1.getLength >1 & temp=max]:  value SortQueue(Q1T,QT);
        [Q1.getLength >1 & temp~=max]: value SortQueue(Q1TT,Q2T);
        []: value QT
        fi
    end;
```

The function "SortQueue" has two parameters Q1 and Q2 which are both in the type of the class Queue defined above. The return value of function "SortQueue" is also in the type of class Queue. The precondition of the function is the length of the Q1 is above 0. And we also use the statement "decrease Q1.getLength" to guarantee the function's termination. One problem here is how to define the return value of the function. The return value should be a sorted queue, but express it by the input parameters is a little difficult. In Perfect, there are two options for defining a function: one is using “`^=`” statement; another is using “`satisfy`” statement. Here we use the “`satisfy`” statement to define the return value of this function should satisfy what conditions. A sorted queue in increasing order means each element in the queue is less than the next element. In Perfect, it can be expressed as: “`forall i::(0..((Q1.getLength)-2)):-result.myqueue[i] <= result.myqueue[i+1]`”. The “result” in the statement is instead of the return value of the function which should be a sorted queue.

To perform the sort algorithm, we also do the refinement to this function. First we define six variables: “max” is the max element of the Q1; “temp” is the head element of the Q1; “QT” is equal to the Q2 adding the head of Q1; “Q1T” is equal to the Q1 removing the head; “Q1TT” is equal to the Q1 removing the head and then adding the head element again. Next we use conditional expressions which are between the “if” and “fi”.

a) If the length of Q1 more than 1 and “temp” equals to “max”, call the function again and pass the parameters with “Q1T” and “QT”.

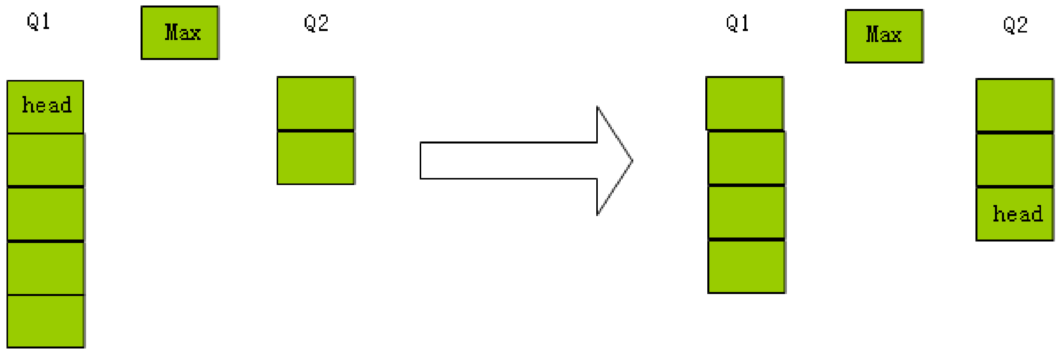


Fig.4 The First Condition of Processing the Sorting Queue.

b) If the length of Q1 more than 1 and “temp” is not equal to “max”, call the function again and pass the parameters with “Q1TT” and “Q2T”.

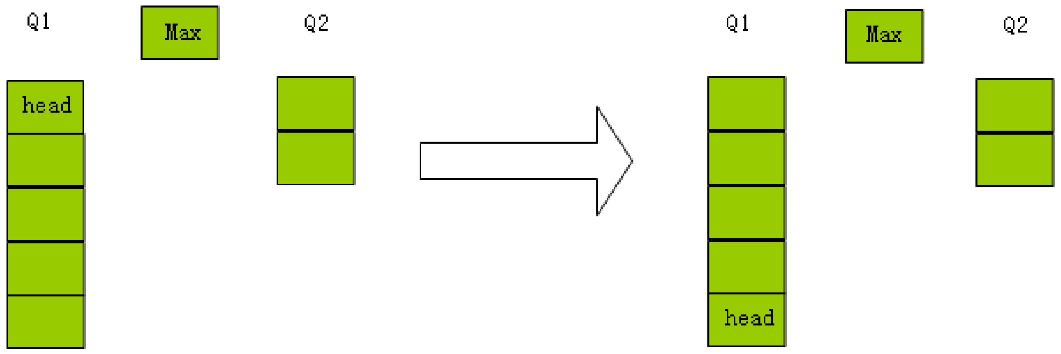


Fig.5 The Second Condition of Processing the Sorting Queue.

c) Otherwise the function returns the QT.

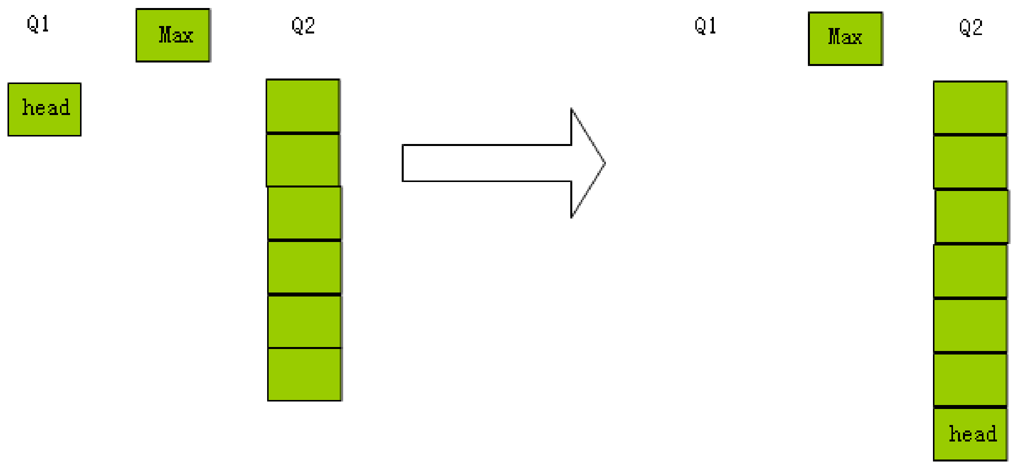
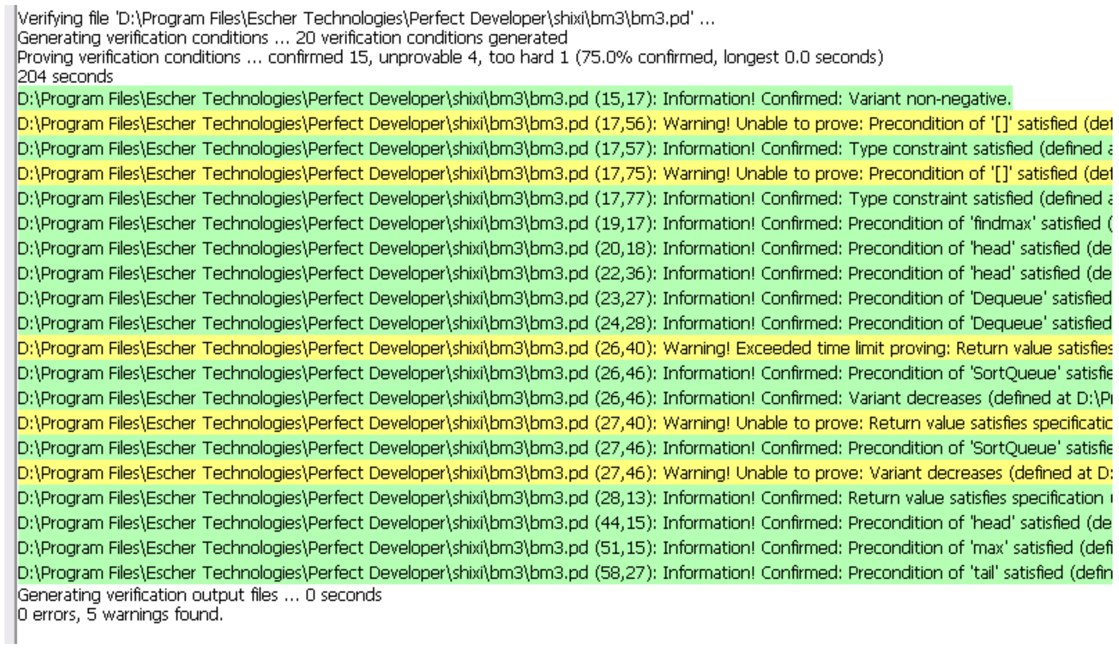


Fig.6 The Third Condition of Processing the Sorting Queue.

So the benchmark 3 has been solved by this solution. We can generate a queue ADT by the class Queue and sort this queue object by pass it into the function "SortQueue" which will sorting the queue and return the sorted one.

Verification: Here we give the result screenshot of verifying the program in Perfect Developer as follows.



```
Verifying file 'D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd' ...
Generating verification conditions ... 20 verification conditions generated
Proving verification conditions ... confirmed 15, unprovable 4, too hard 1 (75.0% confirmed, longest 0.0 seconds)
204 seconds
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (15,17): Information! Confirmed: Variant non-negative.
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (17,56): Warning! Unable to prove: Precondition of '[]' satisfied (def
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (17,57): Information! Confirmed: Type constraint satisfied (defined a
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (17,75): Warning! Unable to prove: Precondition of '[]' satisfied (def
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (17,77): Information! Confirmed: Type constraint satisfied (defined a
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (19,17): Information! Confirmed: Precondition of 'findmax' satisfied (
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (20,18): Information! Confirmed: Precondition of 'head' satisfied (de
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (22,36): Information! Confirmed: Precondition of 'head' satisfied (de
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (23,27): Information! Confirmed: Precondition of 'Dequeue' satisfied
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (24,28): Information! Confirmed: Precondition of 'Dequeue' satisfied
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (26,40): Warning! Exceeded time limit proving: Return value satisfies
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (26,46): Information! Confirmed: Precondition of 'SortQueue' satisfie
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (26,46): Information! Confirmed: Variant decreases (defined at D:\Pr
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (27,40): Warning! Unable to prove: Return value satisfies specificatio
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (27,46): Information! Confirmed: Precondition of 'SortQueue' satisfie
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (27,46): Warning! Unable to prove: Variant decreases (defined at D:
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (28,13): Information! Confirmed: Return value satisfies specification
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (44,15): Information! Confirmed: Precondition of 'head' satisfied (de
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (51,15): Information! Confirmed: Precondition of 'max' satisfied (defi
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd (58,27): Information! Confirmed: Precondition of 'tail' satisfied (defin
Generating verification output files ... 0 seconds
0 errors, 5 warnings found.
```

Fig.7 Verification Result of Benchmark 3

The result shows that there are 0 errors and 5 warnings being found. We are trying to fix all the warnings of this benchmark and later ones now. So each warning will not be described here. We just give the current verification's result.

Summary: We list the good points and drawbacks of the solution for the benchmark 3 in the following table.

Summary of Benchmark 3’s Solution	
Good points	Drawbacks
1. Directly implement the queue sorting in Perfect language and it works fine.	1. Not all the conditions can be proved in the program.
2. Generate the class Queue in Perfect, and use it for sorting.	2. The running time is too long for sorting the queue.
	3. The queue is not generic for collection of other types
	4. Don’t give the find max function while just using the built-in method to find it.

Table 3: Summary of Benchmark 3’s Solution

3.4 Benchmark 4

Problem Requirements: Implement and verify a map [20] data type which is made by the other built-in types in Perfect language. This new type should have the common map's features and functions.

Solution: In the map ADT, a value is mapped to a unique key. A map object has two collections to store key and value pairs. And the map object also has methods to add key and value, find value by key or remove key and value. In this benchmark, we need to build a map class by the other built-in types in Perfect. To perform this in Perfect language, the map’s two collections of key and value are represented by two sequences and the map’s methods are implemented by the functions and schemas. The class Map’s code is as follows:

```
class Map^=
abstract
    var Keys: seq of int,
    Values: seq of string;
interface
    schema !add(key:int, inputvalue: string)
        post ([key in Keys]: (let index^=FindIndex(key);
        Values[index]!=inputvalue), []: Keys!=Keys.append(key) &
        Values!=Values.append(inputvalue));
    schema !remove(key:int)
        post ([key in Keys]: (let index^=FindIndex(key);
        Keys!=Keys.take(index)++Keys.drop(index+1), Values!=
        Values.take(index) ++Values.drop(index+1) ), []:pass);
    function IsEmpty: bool
        ^=([#Keys>0]:false,[]:true);
    function FindIndex(key:int):nat //linear? or Binary?
        pre #Keys >0
        ^=([key in Keys]: Keys.findFirst(key), []: -1);
    function Findkey(key:int): string
```

```
            pre #Keys>0
            ^=([Keys.findFirst(key)~=-1]:Values[Keys.findFirst(key)],
            []:"-1");
        build{}
            post Keys!= seq of int{}, Values!=seq of string{};
        build{inputkeys : seq of int,inputvalues: seq of string}
            pre #inputkeys=#inputvalues
            post Keys!=inputkeys,Values!=inputvalues;
    end;
```

In the abstract section of class Map, we declare two valuables Keys and Values which are both in the type of sequence. One sequence contains the key data of the map; another sequence contains the value data of the map. Each pair of key and value has a mapping relationship. Here this mapping relationship is represented as the same index of them in the two sequences.

In the interface section of class Map, we define two schemas: one is named “add”, the other is named “remove”.

The “add” schema adds the input parameters “key” and “inputvalue” into the map. In Perfect we use a conditional postcondition to check whether the key is in the map before adding. If the key is in the map, then find the index of Key in the sequence “Keys” and change the value to the input value in the sequence “Values” which has the same index. If the key is not in the map, we just add the key and value at the end of each sequence. They have the same index in each sequence.

The “remove” schema removes the key and value in the map if the input “key” is in the map. We also use the conditional postcondition to perform it.

a) If the input “key” is in the map, we do the following steps to remove the “key” and the value which the “key” is mapping to: First, find the index of “key” and store it to value “index”. Next, merge the elements before the element in “index” and the elements after the element in “index”. Then change the “Keys” to be the new sequence. The expression “`Keys.take(index)`” returns a sequence comprising the first index elements of Keys and the expression “`Keys.drop(index+1)`” returns a sequence with the first index elements of seq1 removed. So combine these two returns, we can get the sequence which removes the element key. And also we do it again to remove the mapping value in the “Values”.
b) If the input “key” is not in the map, the function doesn’t do anything.

We also have defined three functions in the class Map: IsEmpty, FindIndex and Findkey.

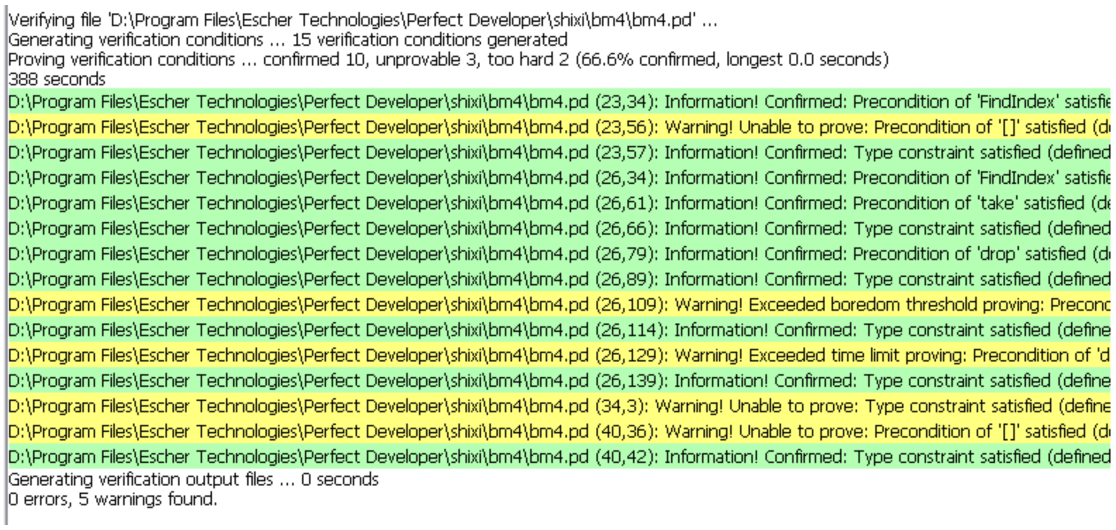
a) The function “IsEmpty” returns a bool value to show whether the map is empty. The function “FindIndex” has one input value “key”. It finds the index of the “key” in the map. If the “index” is in the map, the function “FindIndex” returns the index; otherwise it returns -1 which means the “key” is not in the map.
b) The function “Findkey” find the value which is mapped by the “key”. If the

"key" is in the map, the function "Findkey" returns the value; if not, the function returns -1 which means the key is not in the map and nothing is found.

The class Map has two constructors. One has no input parameter; anther has two parameters which are the sequence of integer and the sequence of string. The previous constructor just initials the values "Keys" to be empty sequence of integer and "Values" to be empty sequence of string.

So now the solution implements the benchmark4 and it has been tested ok in the main program.

Verification: Here we give the result screenshot of verifying the program in Perfect Developer as follows.



```
Verifying file 'D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd' ...
Generating verification conditions ... 15 verification conditions generated
Proving verification conditions ... confirmed 10, unprovable 3, too hard 2 (66.6% confirmed, longest 0.0 seconds)
388 seconds
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (23,34): Information! Confirmed: Precondition of 'FindIndex' satisfie
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (23,56): Warning! Unable to prove: Precondition of '[]' satisfied (d
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (23,57): Information! Confirmed: Type constraint satisfied (defined
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (26,34): Information! Confirmed: Precondition of 'FindIndex' satisfie
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (26,61): Information! Confirmed: Precondition of 'take' satisfied (de
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (26,66): Information! Confirmed: Type constraint satisfied (defined
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (26,79): Information! Confirmed: Precondition of 'drop' satisfied (d
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (26,89): Information! Confirmed: Type constraint satisfied (defined
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (26,109): Warning! Exceeded boredom threshold proving: Preconc
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (26,114): Information! Confirmed: Type constraint satisfied (define
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (26,129): Warning! Exceeded time limit proving: Precondition of 'd
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (26,139): Information! Confirmed: Type constraint satisfied (define
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (34,3): Warning! Unable to prove: Type constraint satisfied (define
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (40,36): Warning! Unable to prove: Precondition of '[]' satisfied (d
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm4\bm4.pd (40,42): Information! Confirmed: Type constraint satisfied (defined
Generating verification output files ... 0 seconds
0 errors, 5 warnings found.
```

Fig.8 Verification Result of Benchmark 4

The result shows that there are 0 errors and 5 warnings being found. We are trying to fix all the warnings of this benchmark and later ones now. So each warning will not be described here.

Summary: We list the good points and drawbacks of the solution for the benchmark 4 in the following table.

Summary of Benchmark4's Solution	
Good points	Drawbacks
1. Successfully implement the map ADT by using the other built-in types in Perfect..	1. Not all the conditions can be proved in the program.
	2. The map is not generic for other types besides integer mapping to string.

Table 4: Summary of Benchmark 4's Solution

3.5 Benchmark 5

Problem Requirements: Implement a linked-list Queue [21] ADT in Perfect language. This Queue ADT is represented by using a linked data structure.

Solution: In other languages, the linked-list queue can be performed in several ways. In C language, the linked-list queue is built by using pointers to get the valuable's memory location. In Java language, the linked-list queue can be performed by using the reference valuables to link each other. But in Perfect, there is not pointer and only has reference which is used for heap type. So the difficulty of this benchmark is how to perform the link relationship between nodes.

First we build a class named "Node" which is used to be the element in the linked-list queue. The "Node" object contains two valuables. One is the value of this node; the other one is used to link the next node. The code of the class "Node" is as follows:

```
class Node ^=
abstract
    var data: int, next: seq of Node;
interface
    selector data;
    selector next;
    schema !setnext(nextnode: Node)
        post ([#next=0]: next!=next.append(nextnode),[]: next[0]!=nextnode);
    build{}
        post data!= 0, next!= seq of Node{};
    build{dt: int}
    post data!=dt,next!= seq of Node{};
end;
```

The class "Node" has two valuables in the abstract section. The "data" stores the value of current node. The "next" is of the type "seq of Node", which means it is a sequence of Nodes. The "next" is used to link the next node. In Perfect language if a class is referred to within a type expression in one of its own member declarations, , the reference must be conditional either by the use of a when-clause or a union of types or a suitable template [7]. This is used to avoid the infinite recursion. So here we use the "seq of Node" instead of the "Node", and just use the "next[0]" to link to the next node.

In the interface section, we declare the valuable "data" and "next" to be writable by using the key word **selector**. The class Node also has a schema named "setnext" which is used to set the next node to the current object. The input next node object is stored in the first element of sequence "next". There are two constructors to initial the Node object with no parameter or one parameter.

Now we come to build the class ListQueue which contains the Node objects as elements. The linked data structure should be performed for the class ListQueue. The class also should have the basic features of Queue like add Node, remove Node and get the first Node. The difficulty here is how to design the data structure. First we give the basic structure of the class ListQueue. The code is as follows:

```
class ListQueue ^=
abstract
  var head: Node, spine: seq of Node;
interface
  function Isempty: int ^=([#spine>0]:1,[]:0);
  function findtail(headnode : Node):Node
      …
  function getnewhead(headnode : Node, newnode: Node):Node
      …
  schema !Enqueue(data:int)
      …
  schema !Dequeue
      …
  function Front: int ^= ([#spine>0]: head.data,[]:0);
  function End: int ^= ([#spine>0]: findtail(head).data,[]:0);
  build{}
      post (head!=Node{}, spine!=seq of Node{});
end;
```

The definitions of the function and schema are not given here. In the structure of the class ListQueue, there are two member valuables: “head” and “spine”. The “head” contains the first Node in the List-Queue. The “spine” contains all of the elements in the List-Queue. And in the interface section, there are five functions to perform basic capabilities of Queue. The function “Isempty” checks whether the ListQueue is empty. The function “Front” returns the first element of the List Queue which is the valuable “head”. The function “End” returns the last element of the List by calling the function “findtail”. The function “findtail” returns the last element of the list. It travels through the List Queue from the head element. The main idea of this function is using loop or recursive to find the current Node object’s next Node. If the next Node exists, the function continues the process. If not, it means the current Node object is the last element in the List Queue. The code of the function “findtail” is as follows:

```
function findtail(headnode : Node):Node
  pre #spine ~=0
  decrease #spine
  satisfy #result.next = 0
  via
      if
      [#headnode.next >0]: value findtail(headnode.next[0]);
      []: value headnode
      fi
end;
```

The function “findtail” has one parameter which is the head node of the list. The refinement performs the recursion of find the tail element. If the current node’s next is not empty, the function will check the next node. If the current node’s next node doesn’t exist, the function returns the current node which is the tail element of the list.

Now we come to solve the most difficult problem: how to add a new node to the ListQueue. There are some limitations in the Perfect language. First we can not use the pointer or reference. To solve the first problem, we use the valuable refer to the Node object directly. Second, the link relationship will be changed if we change the next object. When a new node is added into the List, the last node’s “next” will be set to refer to the new node. The last node is changed which breaks the old relationship. To solve the second problem, we have five ideas to be decided at the beginning. The five ideas are as follows:

1) We use the index of spine as the pointer. Each node has the next node’s index in the sequence spine. But this is not a really List Queue.
2) We use a function to find the tail of the list and then add the new node to it. This will have the problem which breaks the relationship between the last node and the second last node.
3) We use the map in the benchmark 4 to perform the pointer and value. But this is not really List Queue and very complicated.
4) We use a function to find the tail’s index of the spine and add the new node to the last element in the spine. This is also breaking the relationship between the last two nodes.
5) In the above four ideas, the major problem is the change of the object will break the referring relationship of last two nodes. The change cannot be avoided. So we generate a solution which uses the change. The fifth idea is that we use a function to find the tail of the list, add the new node to it and then add each node to the previous node again when the function recursive returns back.

The fifth idea is the solution we choose in the end to solve the adding new node problem. We build a function named “getnewhead” which returns the new head of the new list. The code is as follows:

```
function getnewhead(headnode : Node, newnode: Node):Node
  pre #spine~=0
  decrease #spine
  satisfy #result.next ~= 0
  via
      if
      [#headnode.next >0]: let newhead^= headnode after
it!setnext(getnewhead(headnode.next[0], newnode)); value newhead;
      []: let newhead^= headnode after it!setnext(newnode); value newhead
      fi
  end;
```

The function “getnewhead” has two parameters which are both in the type of Node. “headnode” is the current node to check and “newnode” is the node which need to be added. We use the refinement to perform the recursion for adding the add node to the list. In each time of recursion, we check that if the current node has next node then set the next node of current node be to the return value of passing next node to the function and return the changed current node which named “newhead”; if the current node doesn’t have the next node then set the new node to be the next node of current node “headnode” and return the changed current node which named “newhead”. The function will call itself recursively until find the tail node of the list then it adds the new node to the tail node. When the function returns back recursively, the return value “newhead” would be added to the previous node in each time of recursion.

So the schema “Enqueue” just calls the function “getnewhead” and changes the current head to the new head. The code of the schema is simple as follows:

```
schema !Enqueue(data:int)
    post ([#spine=0]: ( let newnode^= Node{data};  head!=newnode then
spine!=spine.append(newnode)),
    []:( let newnode^= Node{data}; head!=getnewhead(head, newnode) then
spine!=spine.append(newnode)) );
```

The schema “Enqueue” has one parameter “data” which is the data for building the new node. If the list queue is empty, we create a new node and assign it to be the head node. If the list queue is not empty, we add the new node to the last node by calling the “getnewhead” function and assign the return value of the function to be the head. The whole process of adding a new node to the list is showed in the following graphs:

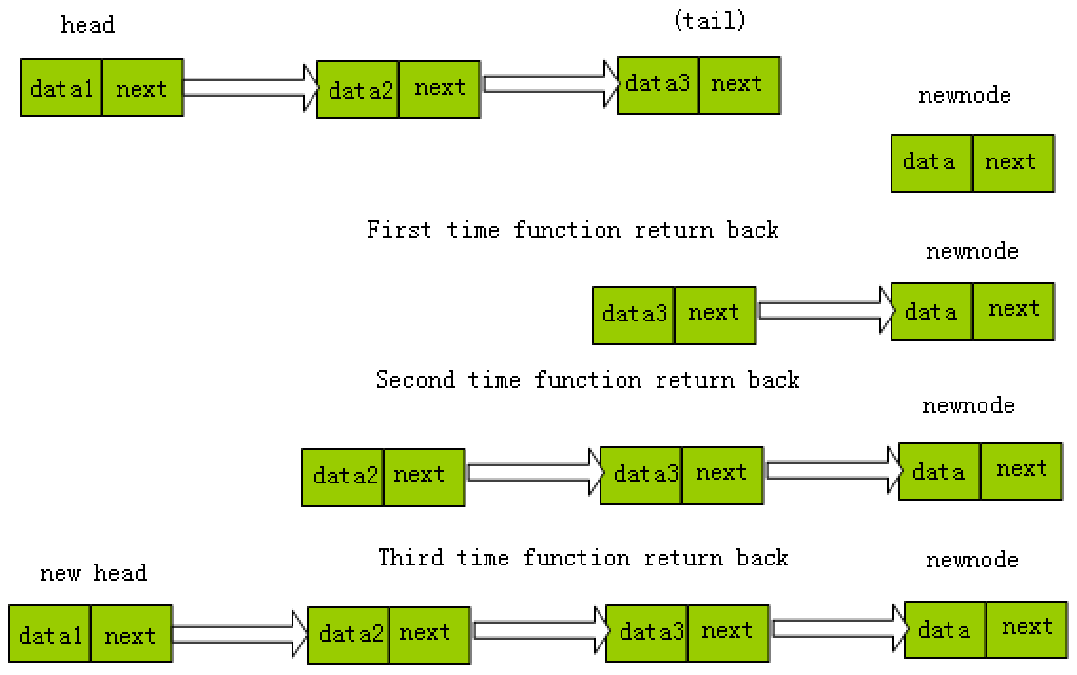


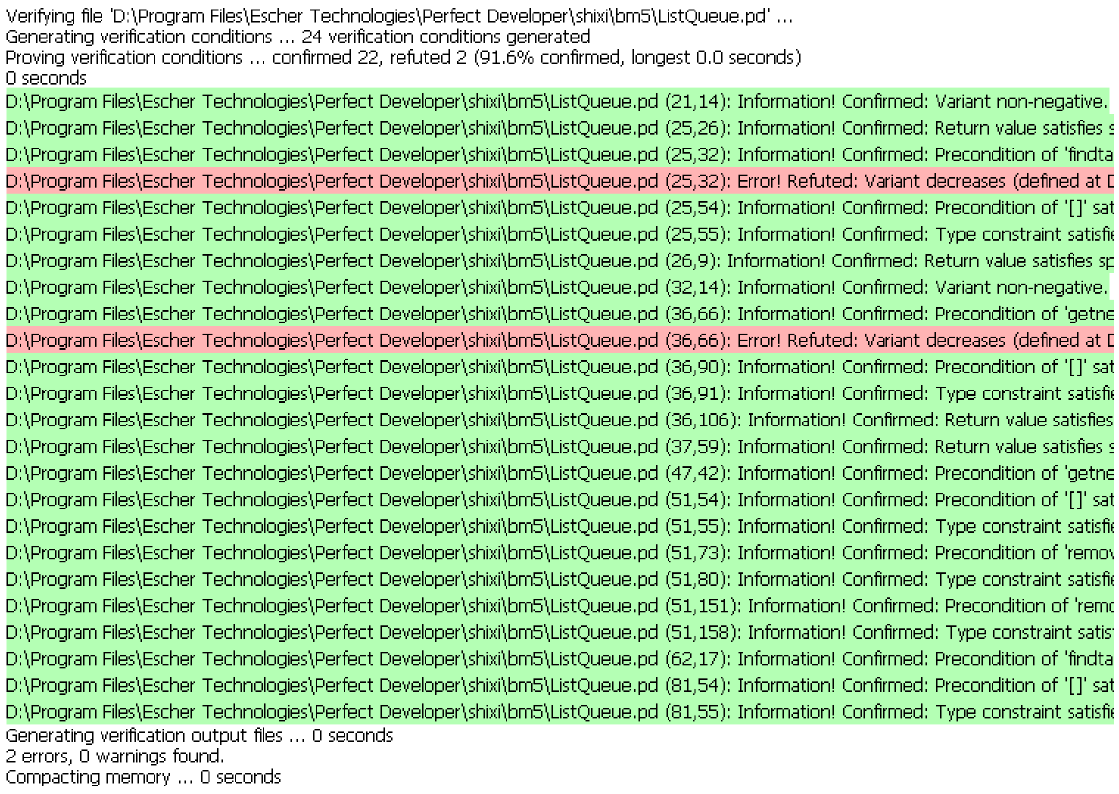
Fig.9 The Process of Adding a new Node to the List

The schema “Dequeue” removes the head node of the list queue. The main idea is just let the head to be the next node of current head node. The code of it is as follows:

```
schema !Dequeue
  pre #spine>0
 post ([#spine>1 & #head.next>0]:(head!=head.next[0] , spine!=spine.remove(0)),
  []:(head.data!=0 then head.next!=seq of Node{} then spine!=spine.remove(0)));
```

So now we have the solution which can perform the implementation of Linked-List queue in Perfect language.

Verification: Here we give the result screenshot of verifying the solution of benchmark 5 in Perfect Developer as follows.



```
Verifying file 'D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd' ...
Generating verification conditions ... 24 verification conditions generated
Proving verification conditions ... confirmed 22, refuted 2 (91.6% confirmed, longest 0.0 seconds)
0 seconds
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (21,14): Information! Confirmed: Variant non-negative.
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (25,26): Information! Confirmed: Return value satisfies s
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (25,32): Information! Confirmed: Precondition of 'findta
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (25,32): Error! Refuted: Variant decreases (defined at D
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (25,54): Information! Confirmed: Precondition of '[]' sat
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (25,55): Information! Confirmed: Type constraint satisfie
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (26,9): Information! Confirmed: Return value satisfies sp
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (32,14): Information! Confirmed: Variant non-negative.
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (36,66): Information! Confirmed: Precondition of 'getne
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (36,66): Error! Refuted: Variant decreases (defined at D
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (36,90): Information! Confirmed: Precondition of '[]' sat
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (36,91): Information! Confirmed: Type constraint satisfie
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (36,106): Information! Confirmed: Return value satisfies
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (37,59): Information! Confirmed: Return value satisfies s
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (47,42): Information! Confirmed: Precondition of 'getne
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (51,54): Information! Confirmed: Precondition of '[]' sat
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (51,55): Information! Confirmed: Type constraint satisfie
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (51,73): Information! Confirmed: Precondition of 'remov
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (51,80): Information! Confirmed: Type constraint satisfie
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (51,151): Information! Confirmed: Precondition of 'remo
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (51,158): Information! Confirmed: Type constraint satis
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (62,17): Information! Confirmed: Precondition of 'findta
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (81,54): Information! Confirmed: Precondition of '[]' sat
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (81,55): Information! Confirmed: Type constraint satisfie
Generating verification output files ... 0 seconds
2 errors, 0 warnings found.
Compacting memory ... 0 seconds
```

Fig.10 Verification Result of Benchmark 5

There are o warning and 2 errors in the verification, while the program of solution for the benchmark 5 is working fine. The first error is “`D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (25,32): Error! Refuted: Variant decreases (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (21,5)), cannot prove: #self.spine < #self.spine.`” It shows the error occurs at the line 25 and position 32 which is refuted by the code at the line 21and position 5 in the ListQueue.pd file. The code at that position is “`decrease #spine`” in the function “findtail”. And the second error is “`D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (36,66): Error!`

`Refuted: Variant decreases (defined at D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm5\ListQueue.pd (32,5)), cannot prove: #self.spine < #self.spine.`" It show the error occurs at the line 36 and position 66 which is refuted by the code at the line 32and position 5 in the ListQueue.pd file. The code at that position is "`decrease #spine`" in the function "getnewhead". So we can see that these two errors occur due to the same problem. The function "findtail" and "getnewhead" are both recursive functions. They go through the list from the head. In the recursive function, the variant decrease expression "`decrease #spine`" doesn't change during the recursion. So the prover think the each time the "#spine" is not less than the previous one, which causes the error. We know the recursive function "findtail" and function "getnewhead" wouldn't cause the infinite recursion. But the prover doesn't know it. So we need to find some variant which will decrease itself at each time of calling the recursive function. The solution of removing the errors is add a new input parameter "count" to the function "findtail" and function "getnewhead". Each time of the recursion, the "count" decreases itself. Then we change the variant decrease expression to be "decrease count". But this solution doesn't have an effect on the program's running and makes the function look complicated. So we don't change the functions and let the error remain in the verification.

Summary: We list the good points and drawbacks of the solution for the benchmark 5 in the following table.

Summary of Benchmark 5's Solution	
Good points	Drawbacks
1. Successfully performs the linked-list in Perfect language without using pointer and reference.	1. Two errors in the verification of the solution in Perfect Developer.
2. Develops a new algorithm to add node to the list in Perfect.	2. The List Queue ADT is not generic for other types besides integer type.
3. A good example of class own member referring to the class self.	

Table 5: Summary of Benchmark 5's Solution

3.6 Benchmark 6

Problem Requirements: Implement the iterator and verify a program which uses the iterator for some collection type.

Solution: Iterator is an object that allows a programmer to traverse through all the elements of a collection in unordered fashion, regardless of its specific implementation [22]. In Perfect, there is not no single iterator type or class, besides the forall() expression. In this benchmark, we need to realize the implementation of iterator and test it in a program. We want the iterator for the certain collection type can go through the collection by calling the next method of the iterator class. The class also should be able to show the current element of the collection and could check whether the collection has the next element or not. After we have the iterator class, we need to use it for some collection type in a program. So this benchmark has two parts: one is implementing the iterator class in Perfect; another is using the iterator object in a program. Here we first give the solution for the class Iterator. The code of class Iterator is as follows:

```
class Iterator ^=
abstract
    var collection: seq of int, count:int;
interface
    function currentvalue: int
        pre 0<=count< #collection
        ^= collection[count];
    function hasnext: int
        ^= ([count<#collection]:1,[]:0);
    schema !next
        pre 0<=count<#collection & #collection >0
        post count!= count+1;
    build{coll:seq of int}
        post collection !=coll, count!=0;
    build{coll:seq of int, cot: int}
        post collection !=coll, count!=cot;
end;
```

In the abstract section, we declare two valuables: “collection” and “count”. The “collection” is in the type of sequence of integer, which stores the collection to travel through. The value “count” is used to count the current index in the collection.

In the interface section, there are defining two functions. The function “currentvalue” returns the value of current element in the collection when the iterator travels through the collection. The function “hasnext” checks whether the collection has the next element. One schema also is defined in this section. The schema “next” change the value of count to perform the travelling through the collection. There are two constructors. One constructor only passes the input parameter “coll” which is the collection for traveling through. The other constructor passes two parameters “coll” and “cot”. The input “cot” is used to initial the valuable “count”.

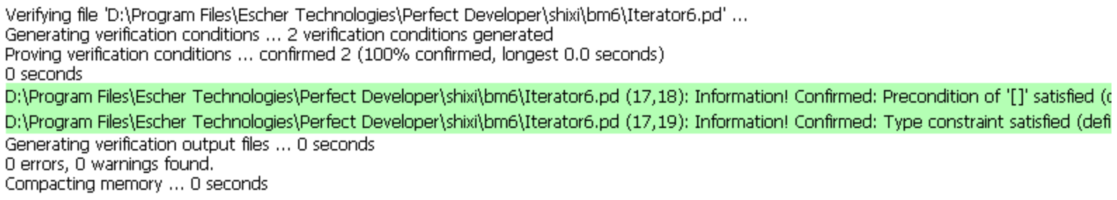
One problem here is that the schema in Perfect cannot return value. We want to use the next method of Iterator to go through the collection and return each element. But the next method will change the Iterator object and in Perfect only schema can change the object’s valuables. Because the next method can only change the valuable, we need another function “currentvalue” to return the value of current element. So when iterator travels through the collection, it calls the function “currentvalue” and schema “next” together.

Now we have the Iterator class, and the next thing to do is building a program to use the Iterator for collection of integer. The program of Perfect language is based on the class. So we need to build a class and use the Iterator in the methods of the class. Here we have a class named "testIterator" and it has a schema named "next5time". This schema will use the Iterator to travel through a collection 5 times. The code of class "testIterator" is as follows:

```
class testIterator ^=
abstract
    var stringseq: seq of string, It: Iterator;
interface
     schema !next5time
         pre #It.collection>5 & It.count=0
         post stringseq!=stringseq.append(It.currentvalue.toString)then It!next
         then stringseq!=stringseq.append(It.currentvalue.toString)then It!next
         then stringseq!=stringseq.append(It.currentvalue.toString)then It!next
         then stringseq!=stringseq.append(It.currentvalue.toString)then It!next
         then stringseq!=stringseq.append(It.currentvalue.toString)then It!next;
    build{inputit: Iterator}
         post stringseq != seq of string{}, It!=inputit;
end;
```

The class "testIterator" has two valuables: "stringseq" is a string type valuable; "It" is in the type of Iterator. The class also has a schema named "next5time". This schema lets the Iterator "It" go through the collection five times and appends the value of element to the value "stringseq" each time. We intend to perform a loop to let Iterator "It" go through the collection. But the problem is that the schema cannot perform a loop in Perfect. So we have to write the whole process of traveling through the collection five times into the code. The constructor of this class has one parameter "inputit" which is an object of class Iterator. It initials the value "It" to be the Iterator object "inputit". To run this schema, we also need a main program to create a collection, an Iterator and generate a "testIterator" object by the Iterator. Then we run schema of the "testIterator" object to show the using of Iterator in Perfect.

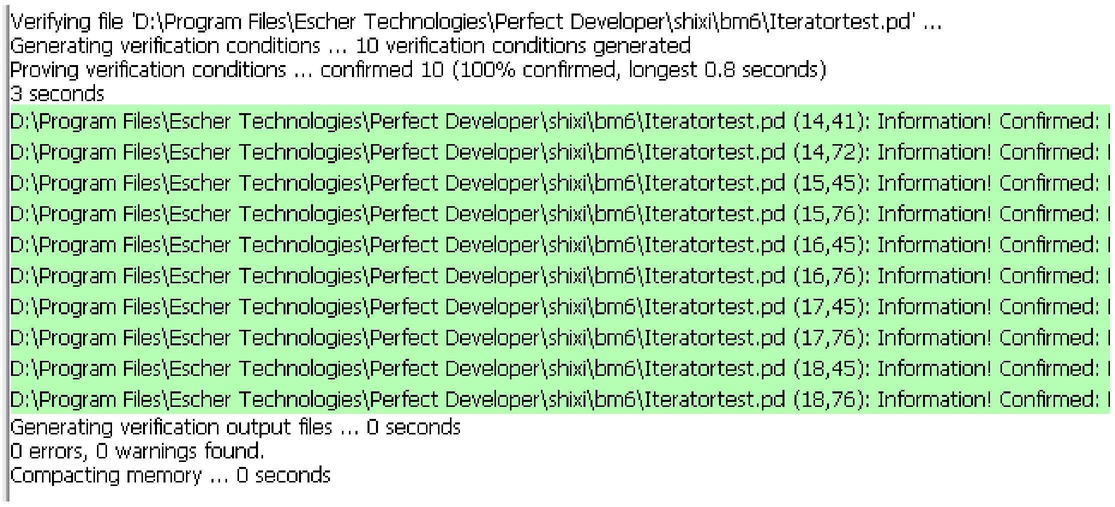
Verification: Here we give the result screenshot of verifying the class Iterator of benchmark 6 in Perfect Developer as follows.



```
Verifying file 'D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iterator6.pd' ...
Generating verification conditions ... 2 verification conditions generated
Proving verification conditions ... confirmed 2 (100% confirmed, longest 0.0 seconds)
0 seconds
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iterator6.pd (17,18): Information! Confirmed: Precondition of '[]' satisfied (c
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iterator6.pd (17,19): Information! Confirmed: Type constraint satisfied (defi
Generating verification output files ... 0 seconds
0 errors, 0 warnings found.
Compacting memory ... 0 seconds
```

Fig.11 Verification Result of the Class Iterator in Benchmark 6

There is no warning or error found. Next we give the result screenshot of verifying the class testIterator as follows:



```
Verifying file 'D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iteratortest.pd' ...
Generating verification conditions ... 10 verification conditions generated
Proving verification conditions ... confirmed 10 (100% confirmed, longest 0.8 seconds)
3 seconds
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iteratortest.pd (14,41): Information! Confirmed: I
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iteratortest.pd (14,72): Information! Confirmed: I
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iteratortest.pd (15,45): Information! Confirmed: I
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iteratortest.pd (15,76): Information! Confirmed: I
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iteratortest.pd (16,45): Information! Confirmed: I
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iteratortest.pd (16,76): Information! Confirmed: I
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iteratortest.pd (17,45): Information! Confirmed: I
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iteratortest.pd (17,76): Information! Confirmed: I
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iteratortest.pd (18,45): Information! Confirmed: I
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm6\Iteratortest.pd (18,76): Information! Confirmed: I
Generating verification output files ... 0 seconds
0 errors, 0 warnings found.
Compacting memory ... 0 seconds
```

Fig.12 Verification Result of the Class testIterator in Benchmark 6

There is no warning or error found in the conditions of class testIterator.

Summary: We list the good points and drawbacks of the solution for the benchmark 6 in the following table.

Summary of Benchmark 6's Solution	
Good points	Drawbacks
1. Successfully performs the Iterator in Perfect language by using the other built-in type.	1. Cannot perform traveling through the Iterator by loop or recursion, which is due to the schema's features in Perfect.
2. Shows the result of the using Iterator.	
3. All the conditions of class Iterator and class testIterator are proved ok in the verification.	

Table 6: Summary of Benchmark 6's Solution

3.7 Benchmark 7

Problem Requirements: Specify simply input and output stream [23] such as character input streams and output streams. Use these streams in an application program to verify them.

Solution: In this benchmark, we don't need to implement the whole I/O system for Perfect language. We just perform the stream class which can be used as input and output stream. The main idea is using the sequence in Perfect to store the data of the stream object. A class Stream is built which can be used in application programs to perform as an input stream or output stream. This stream class also has basic stream capabilities which are implemented as member functions or schemas. The code of the class Stream is as follows:

```
class Stream ^=
abstract
    var InputStream: seq of int, Count: int, IsOpen: int;
    invariant Count=#InputStream;
interface
    function getLength:int ^= Count;
    function IsOpen;
    function getchar:int
        ^= ([#InputStream ~=0]:InputStream.head,[]: 0);
    schema !open
        post ([IsOpen =0]: IsOpen !=1, []: pass);
    schema !close
        post ([IsOpen =0]: pass, []: IsOpen !=0);
    schema !putchar(pc:int)
        post ([IsOpen =1]: (InputStream!=InputStream.append(pc),
        Count!=Count+1), []: pass);
    schema !remove
        pre #InputStream ~=0
       post ([IsOpen =1]: (InputStream != InputStream.tail, Count!= Count-1),
       []: pass);
    build{}
        post InputStream!= seq of int{}, Count!=0, IsOpen!=1;
    build{instr: seq of int}
        post InputStream!= instr, Count!=#instr, IsOpen!=1;
end;
```

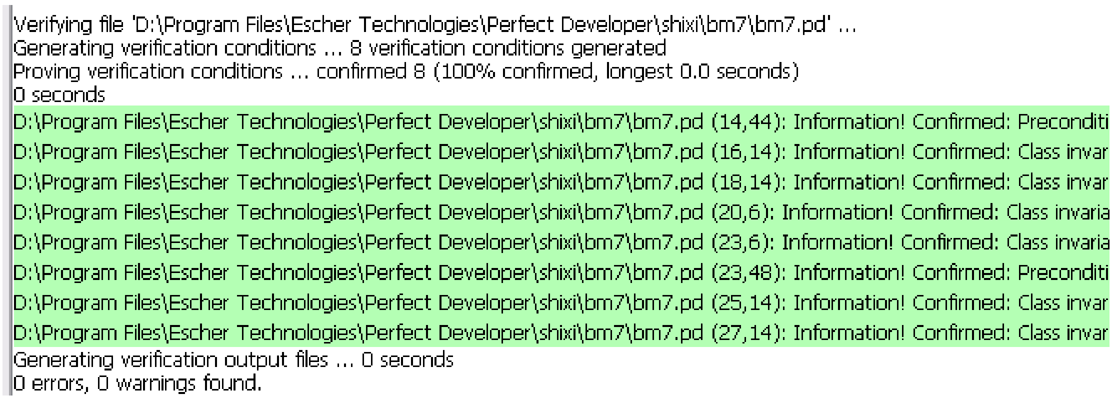
In the abstract section of this class, we first declare three valuables: “InputStream”, count and “IsOpen”. The valuable “InputStream” stores the Stream elements. The valuable “IsOpen” stores the status of the Stream object. If “IsOpen” equals to 0, it means the Stream is closed for the input or output process. If “IsOpen” equals to 1, it means the Stream object is open for doing the input or output operation. Here we also declare an invariant expression “invariant Count=#InputStream;”. It declares the invariant expression “Count=#InputStream” should always be satisfied in the class. When we do any change to the object of class Stream, the valuable “Count” should equal to the length of “InputStream” which is represented by “#InputStream”.

In the interface section, we declare three functions. The first function “getLength” returns the value of the valuable “Count”. The second function “IsOpen” makes the valuable “IsOpen” readable to the users. The third function “getchar” returns the head element of the stream or 0 if the stream has no element.

We also declare four schemas for the class. The first schema “open” changes the valuable “IsOpen” to be 1, which means letting the object be able to do the input or output operation. The second schema “close” changes the valuable “IsOpen” to be 0, which means closing the object’s capability of processing input or output operations. The third schema “putchar” is used to put the new value into the Stream object and also increase the value of “Count”.

After the class has the “putchar” schema, we can do the input processing to the Stream object. But we also need the Stream be able to do the output operation which means doing output of each element in the Stream. But the same problem here as in the benchmark 6 is that: The Schema cannot return the value in Perfect. When we want do some change and store the change in the object, we need use the schema. But the schema doesn't permit to have a return value. In another words, the schema can only do the operation which change the object's status. It cannot return a value as the function. However the function also cannot change the status of the object. To solve this problem, we need to use the function “getchar” and schema “remove” together.

Verification: Here we give the result screenshot of verifying the class Stream of benchmark 7 in Perfect Developer as follows.



```
Verifying file 'D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm7\bm7.pd' ...
Generating verification conditions ... 8 verification conditions generated
Proving verification conditions ... confirmed 8 (100% confirmed, longest 0.0 seconds)
0 seconds
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm7\bm7.pd (14,44): Information! Confirmed: Preconditi
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm7\bm7.pd (16,14): Information! Confirmed: Class invar
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm7\bm7.pd (18,14): Information! Confirmed: Class invar
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm7\bm7.pd (20,6): Information! Confirmed: Class invaria
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm7\bm7.pd (23,6): Information! Confirmed: Class invaria
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm7\bm7.pd (23,48): Information! Confirmed: Preconditi
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm7\bm7.pd (25,14): Information! Confirmed: Class invar
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm7\bm7.pd (27,14): Information! Confirmed: Class invar
Generating verification output files ... 0 seconds
0 errors, 0 warnings found.
```

Fig.13 Verification Result of the Benchmark 7

There is no warning or error found in the conditions of class Stream. All the conditions are confirmed in the verification of benchmark 7.

Summary: We list the good points and drawbacks of the solution for the benchmark 7 in the following table.

Summary of Benchmark 7’s Solution	
Good points	Drawbacks
1. Successfully performs the Input/Output Stream in Perfect language by using the other built-in type.	1. Lack of generic. Can’t let the Stream do the input or output processing for the elements in the other types besides integer.
2. All the conditions of class Iterator and class testIterator are proved ok in the verification.	2. The solution could use the Queue type which is built in the solution of benchmark 3.
	3. Limited capabilities of Stream.

Table 7: Summary of Benchmark 7’s Solution

4. Analysis

This section presents the analysis of Perfect Language based on how it performed on the benchmarks. We summarize the good points and drawbacks about the implementation of these Benchmarks. And we also give the suggestion of how to improve the benchmarks and Perfect Language.

4.1 Analysis of the implementations of benchmarks

In this subsection, we discuss the question: “Can we make the implementations of benchmarks better?” We give some suggestions of improving the current implementations of benchmarks.

Generic: According to the summary of each benchmark in section 3, we can find most implementations of benchmarks are not generic. These implementations only deal with one type of input valuable, like add function, queue type, map type, iterator and stream. These benchmarks would be more useful if they are generic to more than one type. So now the problem is: Can we use the Perfect Language to generate a generic class or function? The answer is “yes”. The Perfect Language supports the generic class. If we define a queue which is generic class, the code is as follows:

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

The above class Queue accepts type parameterization. We can use it to build the Queue of the built-in type or self defined type. So in the further research, we continue to change the implementations of these benchmarks to be generic.

Reusable: The benchmark implemented here can be reused to generate new benchmark. We can build the benchmark 6 by using the linked-list queue in benchmark5. And we also can build the benchmark7 by using the queue in the benchmark3. Build the new benchmark from the earlier benchmarks can show the capabilities of the earlier benchmarks. So in the further research, we will focus on how to generate the benchmark from the earlier benchmarks.

Functional: Some implementations of benchmarks have little functionality. The benchmark 3 sorts the queue in the decreasing order. We can also add the option to sort the queue in the user defined order. The benchmark 7 has few methods of class stream. We can add more methods for the class stream to make it have more capabilities. So in the further research, we will add more features to these benchmarks to improve their functionality.

4.2 Analysis of the Perfect Language

In this subsection, we discuss the question: "Can we make the Perfect Language better for expressing benchmarks?" We give some suggestions of improving the Perfect Language.

Schema: There are two things can be improved in the schema of Perfect language. One is the schema doesn't have return value; the other is the schema doesn't support the loop or recursion.

In the benchmark 6, we want to show we can travel through the collection by iterator. We try to display each element of the collection by calling the iterator's "next" method. But the schema "next" can not return a value. So we have to add a new function "currentvalue" to return the current element value. This makes the program more complicated. During the traveling through the collection by the iterator, we have to call the schema "next" and the function "currentvalue" both in each time. In other language, we just call the "next" method which can return the current value and travel to the next element. In the benchmark 7, we also have this problem. The Steam cannot remove the current element and return the current element's value in the same schema. We suggest the Perfect language implement a new method which can return the value and also change the status of the object.

In the schema of Perfect, we cannot use the refinement to do the loop or recursion operation. We often need to do the loop operation to the class object which may change the status of the object. But we have to use the schema if we want to change some status of the object in Perfect. However, the schema isn't permitted to do the loop operation by refinement. Currently the solution to solve this problem is using a function. The function returns a new object which is the result of changing the current object. This method makes the function complicated, like the binary search function in the benchmark 2. This method cannot solve the problem in the benchmark 6 which is using the iterator. In benchmark 6, we want to go through the collection by using the iterator's schema "next" which would change the status of the member valuable "It" of the object. We cannot perform the loop or recursion in the schema "test5time". So we have to writes the entire process of calling "next" schema five times. This schema "test5time" can only be used to show the traveling 5 times by the iterator. We suggest the Perfect language implement a new method which can change the status of the object and also can do the loop operation or recursion.

Main program: The biggest trouble in main program is that we cannot call the schema of an object separately. In the main program, we must call the schema of the object behind its initialization as follows: "`let testbm4^= Map{} after it!add(2,string1) then it!add(45,string2) then it!remove(2) then it!add(78,string3);`" The map object must call its schema only in the sentence which create the map. It cannot call its schema in another sentence. This makes the program not flexible. If we want to use the schema in another sentence, the only way is copying the object to a new object and then calling the schema. The code of the main program for benchmark 5 is as follows:

```
    let testbm5^= ListQueue{} after it!Enqueue(4) then it!Enqueue(9) then
it!Dequeue then it!Enqueue(13);
    …
     let testbm51^=testbm5 after it!Enqueue(79) then it!Enqueue(92) then
it!Enqueue(192) then it!Enqueue(4992) then it!Dequeue then it!Dequeue
then it!Dequeue;
```

The object testbm5 is of class ListQueue. We copy the testbm5 to testbm51 to use the schema in another sentence. But this method is not good. The code here is more complicated and confusion. So we suggest the Perfect language should permit calling the schema separately in main program.

Pointer and Reference: The most difficult problem in this paper is to perform the lined-list queue in benchmark5. Usually we use pointer or reference to implement it. But in Perfect, there is no pointer and the reference is only used for heap. To solve this problem in benchmark 5, we have to recreate a new list and return the new head at each time of adding node to the list queue. This solution has a big problem is that it costs a longer time to run and uses more space to store than using the pointer or reference. It also has the problem which we need to ensure that the new list contains all the nodes in the old list and the new node. So we suggest the pointer or the reference should be supported in the Perfect Language.

4.3 Analysis of the verifier of Perfect Developer

In this subsection, we discuss the question: "Can we improve the verifier of Perfect Developer?" We analyze the problem in the verification of these benchmarks' solution and give the suggestion of improving the verifier of Perfect Developer.

The one thing we suggest the verifier of Perfect Developer should be improved is comparing the condition "a[x]=b" and the condition "x=a.findFirst(b)". If the elements in the collection "a" are unique, then these two conditions should be proved by each other. But in the benchmark 2 for implementing the binary search, the verifier of the Perfect Developer gives two warnings. The verifier thinks the condition "`((low + high) / 2) = ([b in a]: a.findFirst(b), []: -1)`" cannot be proved by the condtion "`[a[(low+high)/2]=b]: value (low+high)/2;`". It's hard to cover these two warnings. We think the problem is the verifier can not compare the conditions with considering the certain context. In the benchmark 2, the context is the elements are unique and sorted in the queue. In this context, these two conditions should be proved by each other. So we suggest the Perfect Developer improving the verifier to let it be able to prove the conditions under the certain context.

5. Conclusion

This paper proposes the solutions of implementing and verifying seven VSI benchmarks in Perfect language, which are not performed by others before. This paper shows these seven benchmarks can be performed in Perfect language. And these proposed solutions describe how to implement these benchmarks. After implement each benchmark, we also verify the solution and analyze how well the Perfect language can be done in these benchmarks.

All the programs of these benchmarks are built successfully in the Perfect Developer and the java .jar files which are generated for testing. The test results satisfy the expected results. In the implementation part, all the goals are achieved.

In the verification part, there are three benchmarks' solutions can be proved successfully without warnings or errors. Another three benchmark's solutions have several warnings which are reported in the result of verification. Some of them are due to the verifier of Perfect Developer; others are due to the program's algorithm. The verification result of Benchmark 5 has two errors. These two errors are also due to the algorithm chosen which is used to solve the problem of lacking for pointer and reference.

This paper also gives the suggestion for how to make the Perfect language and Perfect Developer better. We suggest these benchmark solutions could be more generic and plan to add more features on benchmark 6 and benchmark 7. And the earlier benchmarks can be used to perform the new benchmark's solution.

This paper also proposes some suggestions to the Perfect language. We suggest the Perfect language should support pointer or reference, and also want it to provide some kind of method which can change the class object and have return value. The loop operation or recursion can only be performed in function now. We suggest the Perfect language provide a new way of implement loop operation or recursion which could change the class object's members.

The specification, implementation and verification of programs in our work will assist the Verified Software Initiative by providing solutions to the proposed verification benchmarks. It is hoped that others will provide their solutions in their "favourite" verification languages and tools so that a comparison on tools and languages for verification can be made. The overall goal is the improvement of verification tools. We will continue to do more research on these benchmarks, optimize the implementation of them and use them to build a Perfect application program in the future.

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. May 4, 2005

[4] K. Rustan M. Leino, Rosemary Monahan.: Program Verification Using the Spec# Programming System (ETAPS Tutorial). March 29, 2008

[5] Sinan Si Alhir.: The Object−Oriented Paradigm. October 23, 1998

[6] Tony Mullins. Lectures on Formal Specification and Design, University of Shanghai for Science and Technology (ppt), June, 2004.

[7] Escher Technologies. The Perfect Developer Language Reference Manual. http://www.eschertech.com/product_documentation/Language%20Reference/LanguageReferenceManual.htm

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

[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] Escher Technologies. Perfect Developer User Guide. http://www.eschertech.com/product_documentation/User%20Guide/UserGuide.htm

[12] David Crocker. Teaching Formal Methods with Perfect Developer. Escher Technologies Ltd.

[13] Escher Technologies. What is Verified Design-by-Contract? http://www.eschertech.com/products/verified_dbc.php

[14] Wikipedia. http://en.wikipedia.org/wiki/Design_by_contract

[15] Wikipedia. http://en.wikipedia.org/wiki/Specification_language

[16] The C# Language. http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx

[17] Wikipedia. http://en.wikipedia.org/wiki/Binary_search_algorithm

[18] The ADT Queue. http://www.maths.abdn.ac.uk/~igc/tch/mx4002/notes/node51.html.

[19] Queue Sort (Selection sort). http://www.csd.uwo.ca/~morey/cs27a02/queueSort.html.

[20] The Map Data Type. http://www.panix.com/~elflord/cpp/stdlib/map.html.

[21] Queue - Linked-List Implementation. http://www.cs.bu.edu/teaching/c/queue/linked-list/types.html

[22] Wikipedia. http://en.wikipedia.org/wiki/Iterator.

[23] Wikipedia. http://en.wikipedia.org/wiki/Standard_streams.

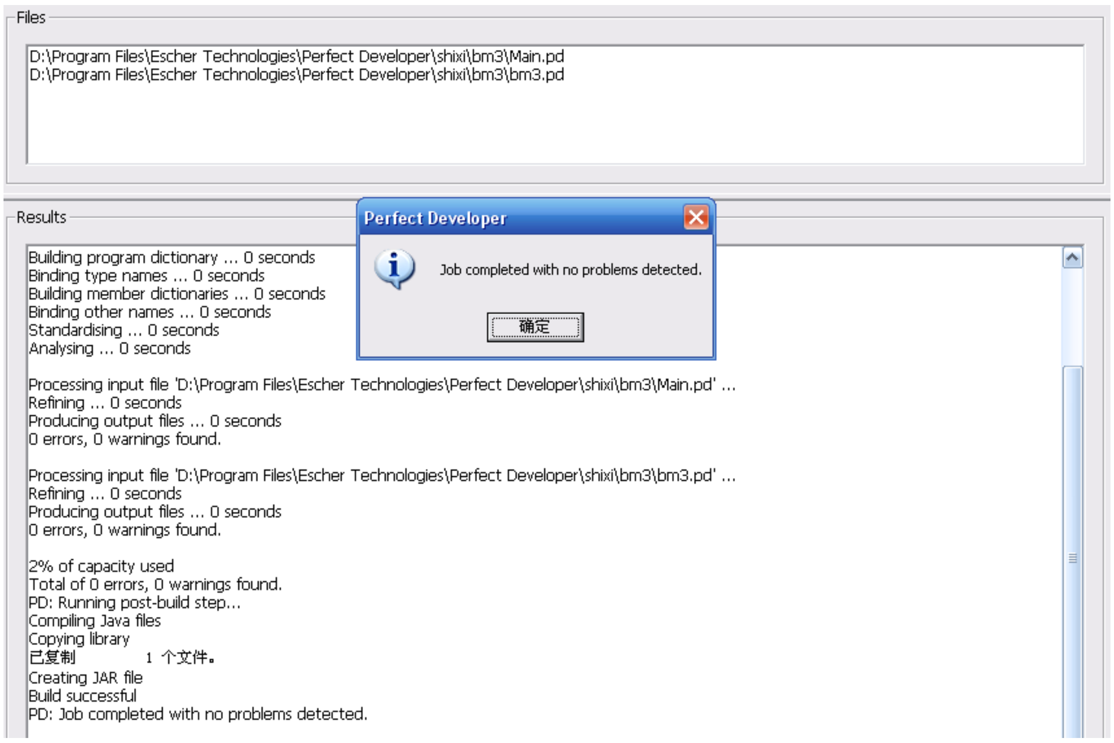
Appendix A: Main Program of Benchmark3

Here we show the main program of the Perfect solutions as an example, which is for testing benchmark3. The code of “Main.pd” is shown as follows:

```
import "bm3.pd";
schema main(context!: limited Environment, args: seq of string,
ret!: out int)
  pre #args > 0
  post
      (
          let sequence1^= seq of int{422,6,34,56,12,87,3};
          let queue1^=Queue{sequence1};
          let emptyqueue^=Queue{};
          let sorttest^=bm3{};
          let
sortedqueue^=sorttest.SortQueue(queue1,emptyqueue);
          let sequencesorted^=sortedqueue.myqueue;
          context!print("sequence sorted is:
"++sequencesorted[0].toString++" "++
sequencesorted[1].toString++"
"++sequencesorted[2].toString++"
"++sequencesorted[3].toString++" ")
      ),
      ret! = 0;
```

Appendix B: Result Screenshot of Benchmark 3.

Here we give the screenshot of the building result of Benchmark 3 as example (Figure.14) and also show the screenshot of the generated java jar file's running result of Benchmark 3 as Example (Figure.15):



```
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\Main.pd
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd
```

Job completed with no problems detected.

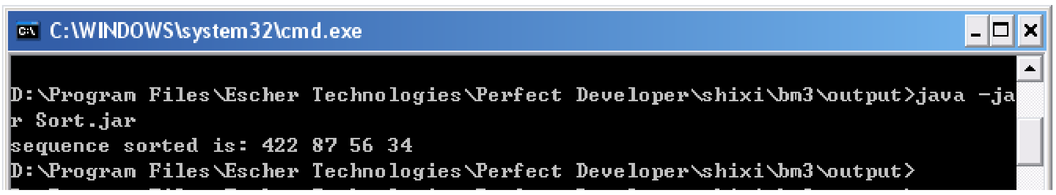
```
Building program dictionary ... 0 seconds
Binding type names ... 0 seconds
Building member dictionaries ... 0 seconds
Binding other names ... 0 seconds
Standardising ... 0 seconds
Analysing ... 0 seconds

Processing input file 'D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\Main.pd' ...
Refining ... 0 seconds
Producing output files ... 0 seconds
0 errors, 0 warnings found.

Processing input file 'D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\bm3.pd' ...
Refining ... 0 seconds
Producing output files ... 0 seconds
0 errors, 0 warnings found.

2% of capacity used
Total of 0 errors, 0 warnings found.
PD: Running post-build step...
Compiling Java files
Copying library
已复制          1 个文件。
Creating JAR file
Build successful
PD: Job completed with no problems detected.
```

Figure.14 The program building result of benchmark 3.



```
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\output>java -ja
r Sort.jar
sequence sorted is: 422 87 56 34
D:\Program Files\Escher Technologies\Perfect Developer\shixi\bm3\output>
```

Figure.15 The Sort.jar file running result of benchmark 3.

Appendix C: The Generated Java Codes of Benchmark 3

Here we present the generated Java codes of benchmark 3 as Example. These following codes contain the class bm3 which has the SortQueue function and the class Queue.

Class bm3's Java codes:

```
class _n_bm3 extends _eAny
{
   public Queue _n_SortQueue (Queue _n_Q1, Queue _n_Q2)
   {
      if  (_eSystem.enablePre  &&  _eSystem.currentCheckNesting  <=
_eSystem.maxCheckNesting)
      {
         _eSystem.currentCheckNesting ++;
         try
         {
            if  (!((0  <  _n_Q1.getLength  ())))  throw  new  _xPre
("bm3.pd:14,21");
         }
         catch (_xCannotEvaluate _lException)
         {
         }
         _eSystem.currentCheckNesting --;
      }
      int _vLet_max_19_9 = _n_Q1.findmax ();
      int _vLet_temp_20_9 = _n_Q1.head ();
      Queue _vLet_Q2T_21_9 = _n_Q2;
      Queue _vit_22_13 = ((Queue) _n_Q2._lClone ());
      _vit_22_13._n_Enqueue (_n_Q1.head ());
      Queue _vLet_QT_22_9 = _vit_22_13;
      Queue _vit_23_15 = ((Queue) _n_Q1._lClone ());
      _vit_23_15._n_Dequeue ();
      Queue _vLet_Q1T_23_9 = _vit_23_15;
      Queue _vit_24_16 = ((Queue) _n_Q1._lClone ());
      _vit_24_16._n_Dequeue ();
      _vit_24_16._n_Enqueue (_vLet_temp_20_9);
      Queue _vLet_Q1TT_24_9 = _vit_24_16;
      if  (((1  <  _n_Q1.getLength  ())  &&  (_vLet_max_19_9  ==
_vLet_temp_20_9)))
      {
         return _n_SortQueue (_vLet_Q1T_23_9, _vLet_QT_22_9);
      }
      else if (((1 < _n_Q1.getLength ()) && (!(_vLet_max_19_9 ==
_vLet_temp_20_9))))
      {
         return _n_SortQueue (_vLet_Q1TT_24_9, _vLet_Q2T_21_9);
      }
      else
      {
         return _vLet_QT_22_9;
      }
   }
   public _n_bm3 ()
   {
      super ();
   }

   public boolean _lEqual (_n_bm3 _vArg_8_1)
   {
      if (this == _vArg_8_1) return true;
      return true;
   }

   public boolean equals (_eAny _lArg)
   {
      return _lArg == this || (_lArg != null && _lArg.getClass () ==
_n_bm3.class && _lEqual ((
         _n_bm3) _lArg));
   }
}
```

Class Queue’s generated Java code

```
class Queue extends _eAny
{
   public _eSeq myqueue;
   public int head ()
   {
      if (_eSystem.enablePre && _eSystem.currentCheckNesting <=
_eSystem.maxCheckNesting)
      {
         _eSystem.currentCheckNesting ++;
         try
         {
            if (!((0 < myqueue._oHash ()))) throw new _xPre
("bm3.pd:43,17");
```

```
            }
            catch (_xCannotEvaluate _lException)
            {
            }
            _eSystem.currentCheckNesting --;
        }
        return ((_eWrapper_int) myqueue.head ()).value;
    }
    public int getLength ()
    {
        return myqueue._oHash ();
    }
    public int findmax ()
    {
        if  (_eSystem.enablePre  &&  _eSystem.currentCheckNesting  <=
_eSystem.maxCheckNesting)
        {
            _eSystem.currentCheckNesting ++;
            try
            {
                if  (!((0  <  myqueue._oHash  ())))  throw  new  _xPre
("bm3.pd:50,17");
            }
            catch (_xCannotEvaluate _lException)
            {
            }
            _eSystem.currentCheckNesting --;
        }
        return ((_eWrapper_int) myqueue.max ()).value;
    }

    public void _n_Enqueue (int input)
    {
        if ((!myqueue._ovIn (((_eAny) new _eWrapper_int (input)))))
        {
            myqueue  =  myqueue.append  (((_eAny)  new  _eWrapper_int
(input)));
        }
        else
        {
        }
    }

    public void _n_Dequeue ()
    {
       if   (_eSystem.enablePre   &&   _eSystem.currentCheckNesting   <=
_eSystem.maxCheckNesting)
       {
          _eSystem.currentCheckNesting ++;
          try
          {
             if  (!((0  <  myqueue._oHash  ())))  throw  new  _xPre
("bm3.pd:57,17");
          }
          catch (_xCannotEvaluate _lException)
          {
          }
          _eSystem.currentCheckNesting --;
       }
       myqueue = myqueue.tail ();
    }

    public Queue ()
    {
       super ();
       myqueue = new _eSeq ();
    }

    public Queue (_eSeq inputseq, int _t0inputseq)
    {
       super ();
       myqueue = inputseq;
    }
    public boolean _lEqual (Queue _vArg_37_5)
    {
       if (this == _vArg_37_5) return true;
       return _vArg_37_5.myqueue._lEqual (myqueue);
    }

    public boolean equals (_eAny _lArg)
    {
       return _lArg == this || (_lArg != null && _lArg.getClass () ==
Queue.class && _lEqual ((
          Queue) _lArg));
    }
}
```