Wuhan University Journal of Natural Sciences

2021, Vol.26 No.6, 481-488

Article ID 1007-1202(2021)06-0481-08

DOI https://doi.org/10.1051/wujns/2021266481

A Method to Deduce and Synthesize the Dafny Programs

□ WANG Changjing[1,2], DING Xilong[1], HE Jiangfei[1], CHEN Xi[1], HUANG Qing[1], LUO Haimei[3], ZUO Zhengkang[1†]

1. College of Computer Information Engineering, Jiangxi Normal University, Nanchang 330022, Jiangxi, China;

2. Management Science and Engineering Research Center, Jiangxi Normal University, Nanchang 330022, Jiangxi, China;

3. College of Physics and Communication Electronics, Jiangxi Normal University, Nanchang 330022, Jiangxi, China

Abstract: We propose a systematic method to deduce and synthesize the Dafny programs. First, the specification of problem is described in strict mathematical language. Then, the derivation process uses program specification transformation technology to perform equivalent transformation. Furthermore, Dafny program is synthesized through the obtained recursive relationship and loop invariants. Finally, the functional correctness of Dafny program is automatically verified by Dafny verifier or online tool. Through this method, we deduce and synthesize Dafny programs for many typical problems such as the cube sum problem, the minimum (or maximum) contiguous subarray problems, several searching problems, several sorting problems, and so on. Due to space limitation, we only illustrate the development process of Dafny programs for two typical problems: the minimum contiguous subarray problem and the new local bubble sorting problem. It proves that our method can effectively improve the correctness and reliability of Dafny program developed. What's more, we demonstrate the potential of the deductive synthesis method by developing a new local bubble Sorting program.

Key words: Dafny; deductive synthesis; specification transformation technology; recursive relationships; loop invariants

CLC number: TP 305

Received date: 2021-10-16

Foundation item: Supported by the National Natural Science Foundation of China (61762049, 61862033, 61902162 ,11804133) and Natural Science Foundation of Jiangxi Province (20202BABL202025, 20202BABL202026, 20202BAB202015)

Biography: WANG Changjing, male, Ph. D., Professor, research direction: software formal method, trustworthy software. E-mail: wcj771006@ 163.com

† To whom correspondence should be addressed. E-mail: zhengkang2005@iscas.ac.cn

0 Introduction

From the perspective of computer science, it is almost impossible to create completely reliable software except for very small programs[1]. We have known for decades about the principles and techniques for ensuring that programs are correct, but it is not easy to verify the program correctly. As verification tools become powerful and practical in the past decade, it makes formal software verification feasible. For example, in order to improve the efficiency and reliability of software development, Microsoft recruits experts and outstanding graduates to specialize in the research of formal methods and verification tools. At the same time, with the rapid development of automatic proof theory, the improvement of CPU computing power, and the enhancement of theorem prover, the program proof tasks emerge in large numbers, and many well-designed program provers appear[2].

With the development of formal verification technology, the successful application of formal verification tools in program verification continues to increase. Due to the advantages of formal methods[3] in the development of complex and reliable software, formal verification tools will significantly improve our efficiency in verifying reliable algorithm software in the future. In addition, formal verification tools have the following advantages: higher verification efficiency and no need to write certificates manually. Dafny, one of the most representative formal verification tools, can verify the correctness of the program and provide complete automation, and also can reduce the workload of our comprehensive verification of the program[4].

This paper uses formal method to develop Dafny programs for two typical problems: the minimum contiguous subarray problem and the new local bubble sorting problem. The final developed programs are automatically verified by Dafny verifier. The main contributions are as follows:

1) We propose a system method to deduce and synthesize the Dafny programs. Initially, the specification of problem is described in strict mathematical language. Then, the derivation process uses program specification transformation technology to perform equivalent transformation. Furthermore, Dafny program is synthesized through the obtained recursive relationship and loop invariants. Finally, the functional correctness of Dafny program is automatically verified by Dafny verifier or online tool.

2) We deduce and synthesize Dafny programs for many typical problems such as the cube sum problem, the minimum (or maximum) contiguous subarray problems, several searching problems, several sorting problems. Due to space limitation, we only illustrate the development process of Dafny by focusing on two typical problems: the minimum contiguous subarray problem and the new local bubble sorting problem. It proves that our method can effectively improve the correctness and reliability of the developed Dafny program.

3) We demonstrate the potential of the deductive synthesis method by developing a new Local Bubble Sorting program.

The organizational structure of this article is as follows: Section 1 is the introduction of related work; Section 2 is about related technologies and tool, briefly introducing the specification transformation technology and Dafny; Section 3 is the deductive synthesis Dafny programs for minimum contiguous subarray problem and bubble sorting problem; Section 4 is conclusion and future work.

1 Related Work

In this section, we survey prior work that are closely related to the method proposed in this paper.

Program synthesis. There has been a great deal of interest in automated synthesizing programs from high-level expressions of user intent[5-7] in the past decade. Some of these techniques are geared towards computer end-users and they utilize informal specifications such as input-output examples, natural language, or a combination of both. On the other hand, program synthesis techniques are geared towards programmers who often utilize additional information, such as a program sketch or types in addition to test cases or logical specifications. Most of the program synthesis methods are obtained from semi-formal description (such as UML) or unformal description (such as natural language). The main difference between those methods and ours is that we deduce the Dafny program from formal specification, which is helpful to subsequent formal reasoning. In addition, our method synthesizes recursive relationships and loop invariants to develop the Dafny programs.

Program transformation. Program transformation was introduced by researchers 50 years ago, and then it was formalized[8-10]. From then on, program transformation technology has gradually developed. Program transformation converts one program into another[10]. Learnig the idea of program transformation, we design the specification transformation technology. Program transformation converts an initially inefficient program into an equivalent program. Our method is different from program transformation by generating an effective algorithm from the problem specification. We also learn from the idea of transformation strategies[11-13] to get better transform rules.

Program refinement[14]. At present, some good program refinement work has been applied to practice. The authors of Refs.[15-17] proposed a mechanism to gradually refine the specification into executable code by introducing implementation details. In our method, we use strict formal derivation technology and specification transformation technology to make the final derivation procedure more rigorous and correct.

2 Related Technology and Tool

In this section, we mainly introduce program specification transformation technology and Dafny.

2.1 Program Specification Transformation Technology

The proof of correctness of program, formal derivation and functions depends on whether the specification is correct and whether the construction is proper either to a great extent. Some quantifiers should be used when describing the program specification of the problem[18]. The general form is: $(Qi{:}r(i){:}f(i))$, $r(i)$ is the range of the constraint variable i, and $f(i)$ is the function of the constraint variable. The meaning of this form is "the quantity obtained by performing q operation on the function $f(i)$ in the range of $r(i)$". The quantifier Q mainly includes $\forall$(universal quantifier), $\exists$(existential quantifier), Σ (summation quantifier), Π (quadrature quantifier), MIN(minimum quantifier), MAX(maximum quantifier), N(count quantifier), etc[19]. What we mentioned below will be used in the transformation of program specification.

1) Range division

$$(Qi :r(i): f(i))=(Qi : r(i)\wedge b(i) :f(i))q(Qi :r(i)\wedge\neg b(i): f(i))$$

2) Cross-product nature

$$(Qi, j :r(i)\wedge s(i, j): f(i, j))=(Qi :r(i):(Qj :s(i, j) :f(i, j)))$$

3) Single range

$$(Qi : i = k:f(i)) = f(k)$$

4) General distribution law

$$(Qi: r(i):g(i)\ q\ f(i))= (Qi :r(i): g(i))q(Qi :r(i): f(i))$$

where i is not a free variable in g.

2.2 Dafny Overview

Dafny is a programming language and static program prover. The Dafny language combines functional programming and object-based programming paradigms. It supports generic classes and dynamic allocation, with built-in specification structures (like Eiffel, JML, and Spec#)[20]. Dafny programming language supports static verification of the program, and it contains annotations for specifying programs. Annotations include pre-conditions and post-conditions, framing specifications (reads and modifies sets), loop invariants, and termination metrics. To further support the specification, the language also provides updatable ghost variables, user-defined functions, algebraic data types, sets, and sequences. These features allow modular verification[21] of the specified program, so the individual verification of each part of the program means the correctness of the entire program. Dafny requires some unique tools to support formal verification. The terms of related keywords and other keywords are shown in Table 1.

Dafny uses a syntax similar to other programming languages, but it requires some special tools to help with formal verification. Annotation is a tool provided by Dafny to help users verify the program. Annotation is not part of the actual program, but it provides specifications and information about methods in the program. By considering the annotations in the function, Dafny can more directly verify the correctness of the program. Dafny transforms the burden of writing bug-free code into the burden of writing bug-free annotations. It is usually easier than writing code, because annotations are shorter and more direct. In addition, the behavior of writing annotations can help people understand what the code does at a deeper level. Dafny can also prove that there are no runtime errors, such as index out of bounds, null cancellation, division by zero, etc.

Table 1 Keywords table

ensures	boolean expression (postconditions)
assert	boolean expression
invariant	boolean expression (that holds before during, and at the conclusion of a while loop)
decreases	ranking function
forall	counter :: range ⇒ boolean expression
reads	immutable object
modifies	mutable object

An overview of the entire Dafny system is given in Fig. 1. The way programmers interact with it is the same as static type checker, when the tool reports an error, the programmer will respond by changing the program's type declaration, specifications, and statements. After the Dafny code passes the Dafny compiler, there are two paths: one is that the Dafny compiler directly generates C # code, and then compiles it to the MSIL byte code of the .NET platform; the other is that Dafny's program verifier converts the given Dafny program into the intermediate verification language Boogie[22]. Then, the Boogie tool generates a first-order verification condition, and finally passes it to the Z3[23] SMT solver[24] for verification. Any content that violates these conditions will be returned as a verification error.

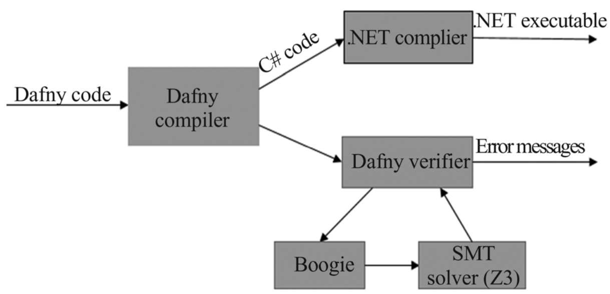

Fig.1 Verification mechanism diagram

3 Deductive Synthesis of Dafny Programs for Two Typical Problems

In this section, we will use the deductive and synthesis method to develop Dafny programs[25] for two typical problems. One is the minimum contiguous subarray problem; another is the new local bubble sorting problem.

3.1 Minimum Contiguous Subarray

Problem description: Calculate the minimum sum of adjacent elements in a given integer array $a[0: n-1]$.

● Provide problem specification

Using minsum($n-1$) to represent the smallest sum of contiguous subarray of its input array $a[0: n-1]$.

Q: $n>0$

R: minsum($n-1$) = (MIN i, j:$0\leqslant i\leqslant j<n$:sum($i, j$))

The auxiliary function sum (i, j) is defined as follows:

sum(i,j)= ($\sum k$:$i\leqslant k\leqslant j$: $a[k]$)=sum(i, $j-1$)+$a[j-1]$, $i\leqslant j$

sum(i,j)=0, if $i>j$

The auxiliary function min is defined as follows:

min(a,b)= if $a>b$ then b else a

● Partition the original problem

Divide the original problem into sub-problems with the same structure, then construct a recursive relationship for solving the problem and a preliminary algorithm. Obviously, the most commonly used method for this problem is the exhaustive method. That is to list the sum of all adjacent elements and compare them to get the solution of the minimum contiguous subarray program, and the time complexity of the algorithm is $O(n^3)$. But, we derive another algorithm to reduce the time complexity and optimize the algorithm. We divide the original problem as follows:

minsum($a[0:n-1]$)=F(minsum($a[0:m-2]$,$a[m-1]$)), $0\leqslant m\leqslant n$

We start with the post-assertion to find the recursive relationship F.

● Find the recurrence relations

minsum($a[0:m]$)

=(MIN i, j : $0\leqslant i\leqslant j\leqslant m$: sum($i$,$j$))

=[cross-product nature]

(MIN j:$0\leqslant j\leqslant m$: (MIN i: $0\leqslant i\leqslant j$: sum(i, j)))

=(MIN j:$0\leqslant j\leqslant m$: ms(j)))

Define ms (j) = (MIN i: $0\leqslant i\leqslant j$: sum (i, j))

=min((MIN j: $0\leqslant j\leqslant m$: ms(j))), ms(m))

=[range division and single range]

=min(minsum($a[0: m-1]$), ms(m))

The first recursive relationship is:

Recurrence1:

minsum($a[0:m]$)=min(minsum($a[0:m-1]$),ms(m)), $0\leqslant m\leqslant n$

Because Recurrence 1 contains the function of ms (m), we need to find the relationship between ms (m) and ms ($m-1$):

ms(m)

=(MIN i:$0\leqslant i\leqslant m$:sum(i,m))

=(MIN i:$0\leqslant i\leqslant m$: ($\sum k$:$i\leqslant k\leqslant m$:$a[k]$))

=[range division and single range]

=(MIN i:$0\leqslant i\leqslant m$: ($\sum k$:$i\leqslant k\leqslant m-1$:$a[k]$)+ $a[m]$)

=[General distribution law and Definition of sum]

=(MIN i: $0\leqslant i\leqslant m$:sum(i,$m-1$))+$a[m]$

=[range division and single range]

=min((MIN i:$0\leqslant i<m$:sum(i,$m-1$)), sum(m, $m-1$)) $a[m]$

=[Definition of ms(m)]

=min(ms($m-1$),0)+$a[m]$

=min(ms($m-1$)+$a[m]$,$a[m]$)

The second recursive relationship is:

Recurrence 2:

ms(m)=min(ms($m-1$)+$a[m]$, $a[m]$)

When m = 0, ms($m-1$) = 0, minsum(a [0: $m-1$]) = 0, we can get Initialization 1.

Initialization 1: $m = 0 \wedge$ ms($m-1$)=0 $\wedge$ minsum ($a[0: n-1]$) = 0.

● Write the loop invariant

We can get the loop invariant by storing minsum ($a[0:m-1]$) and ms(m) into the variable s and c.

$\boldsymbol{\rho}$: $0\leqslant m\leqslant n\wedge s=$ minsum($a[0:m-1]$)$\wedge c=$ms(m)

● Generate the Dafny algorithm program

Then, the algorithm with time complexity of $O(n)$ is obtained by combining the Initiation 1, Recurrence 1 and Recurrence 2. Compared with time complexity $O(n^3)$ of the exhaustive method, the algorithm has better optimization efficiency. Algorithm program is as follows:

```
method minsum(a: array〈int〉 ) returns(s: int)
{
 var m , c := 0 , 0;   s := 0;
 while(m<a.Length)
  decreases a.Length−m
  {
   c := min(c + a[m],a[m]);
   s := min(s, c);
   m := m + 1;
 }
}
```

The Q, R, auxiliary functions in the first step, and the loop invariant in the fourth step are respectively expressed by Dafny and added to the Dafny program for verification. The Dafny program for the smallest sum of contiguous subarrays is as follows:

```
function method sum(a: array〈int〉, i: int, j: int): int
  reads a
  requires a!= null
  requires 0≤i≤j≤a.Length
  decreases j−i
  {
```

```
      if j == i then 0 else sum(a, i, j −1) + a[j−1]
   }
   function method min(a:int, b:int):int
      {if a>b then b else a}
   method minsum (a: array 〈int〉 ) returns(s: int)
      requires a != null
      ensures for all i, j :: 0≤i<j≤a.Length−1 ==>
      sum(a, i, j) >= s
      {
         var m , c := 0 , 0;    s := 0;
         while(m<a.Length)
            invariant 0≤m≤a.Length
            invariant for all i, j :: 0≤i<j≤m
            ==>sum(a, i, j) >= s
            invariant for all i :: 0≤i<m ==> sum(a, i,
            m) >= c
   decreases a.Length−m
      {
         c := min(c + a[m],a[m]);
         s := min(s, c);
         m := m + 1;
      }
   }
```

● Verification of the final Dafny algorithm program

The following are the main methods and verification results of minsum (Fig. 2).

```
   method Main()
   {
      var a:array<int>:= new int[7][4,−7,6,−3,−9,1,2];
      var m := minsum(a);
      print "minsum:", m;
}
```

```
Dafny 2.3.0.10506

Dafny program verifier finished with 4 verified, 0 errors
Running···

minsum: −13
```

Fig.2 Minsum verification result

3.2 New Local Bubble Sorting

Problem description: Arrange the given integer array a [0: $n-1$] in no descending order.

● Provide problem specification

Using BubbleSort(a,0,n) to represent the post-conditions of Bubble Sort.

Q: a != null & $n>0$

R: BubbleSort(a,0,n) = $(\forall j : 1\leqslant j<n : a[j-1]\leqslant a[j])$ $\wedge$perm(a,b,0,$n-1$)

The auxiliary function perm(a,b,0,$n-1$) is defined as follows:

perm(a,b,0,$n-1$)$\equiv(\forall i{:}0\leqslant i<n{:}\ (Nj{:}\ 0\leqslant j<n{:}a[j]=a[i])=(Nk{:}\ 0\leqslant k<n{:}b[k]=a[i]))$

● Partition the original problem

Dividing the original problem into sub-problems with the same structure, constructing a recursive relationship for solving the problem and a preliminary algorithm.

BubbleSort(a, m, n)=F(BubbleSort(a, $m+1$, n), BubbleStep(a, m, n)), $0\leqslant m\leqslant n$

We start with the post-assertion to find the recursive relationship F.

● Find the recurrence relations

Dividing the original problem and finding the recursive relationship. Defining partition function through constructing sub-problems.

BubbleSort(a , m , n)

$=(\forall j{:}m\leqslant j<n{:}\ a[j-1]\leqslant a[j])$

$=(\forall i , j{:}m\leqslant j\leqslant i<n{:}a[j-1]\leqslant a[j])$

$=(\forall i , j{:}m\leqslant j<n\wedge m\leqslant i<n{:}a[j-1]\leqslant a[j])$

=[cross-product nature]

$=(\forall i{:}m\leqslant i<n{:}(\forall j{:}m\leqslant j<n{:}a[j-1]\leqslant a[j]))$

$=(\forall i{:}m+1\leqslant i<n{:}(\forall j{:}m\leqslant j<n{:}a[j-1]\leqslant a[j]))$
$\wedge(\forall j{:}m\leqslant j<n{:}a[j-1]\leqslant a[j]))$

=[range division and single range]

=BubbleSort(a, $m+1$, n)$\wedge(\forall j{:}\ m\leqslant j<n{:}\ a[j-1]\leqslant a[j]))$

=[Definition of BubbleSort]

Letting BubbleStep (a, m, n) = $(\forall j{:}\ m\leqslant j<n{:}\ a[j-1]\leqslant a[j]))$, we get the first recursive relation of bubble sorting algorithm.

1-BubblesortUp:BubbleSort(a,m,n)=BubbleSort(a, $m+1$,n)$\wedge$BubbleStep(a,m,n), m:1..n

The following is the recursive relationship of the BubbleStep function.

BubbleStep(a, m, j)

$=(\forall k{:}m\leqslant k\leqslant j-1{:}a[k-1]\leqslant a[k])$

=[restricted variable name change, definition]

$=(\forall k{:}m\leqslant k\leqslant j-2{:}a[k-1]\leqslant a[k])\wedge a[j-1]\leqslant a[j]$

=[range division and single range]

=bubbleStep(a ,m , $j-1$)$\wedge a[j-1]\leqslant a[j]$

We can get the second recursion relation from right to left.

2-BubbleStepRL:

BubbleStep(a, m, j)=BubbleStep(a,m,$j-1$)$\wedge a[j-1]\leqslant a[j]$, j:m..1

Combining the recursive relations 1, 2 and changing the name of the variable m to i, we obtain the following two recursive relations.

BubbleSort(a,i,n)
=BubbleSort(a,i+1,n)∧BubbleStep(a,i,n), i:1..n

BubbleStep(a,i,j)=BubbleStep(a,i,j−1)∧$a[j-1] \leqslant a[j]$, j:i..1

● Write the loop invariant

Combining the definitions of the recursive relation *BubbleSort* and *BubbleStep* and the variable scope of i , j. we can obtain the loop invariant:

$\boldsymbol{\rho}$: BubbleSort(a, i, n)=($\forall i$:m+1$\leqslant i < n$:($\forall j$:$m \leqslant j < n$:$a[j-1] \leqslant a[j]$))∧BubbleStep(a,i,j−1)=∧$a[j-1] \leqslant a[j]$; i: 1..n, j:i..1

● Generate the Dafny algorithm program

Here are two Dafny methods to achieve the above recursive relationship:

```
method BubbleSort(a:array〈int〉)
{
  if a.Length>1
  {
    var i := 1;
    while i<a.Length
    decreases a.Length−i;
    {
      BubbleStep (a , i);
      i := i + 1;
    }
  }
}
method BubbleStep (a:array〈int〉, i:int )
{
  var j := i;
    while (j>0 && a[j−1]>a[j])
    decreases j
    {
      a[j−1],a[j] := a[j], a[j−1];
      j := j−1;
    }
}
```

Add Q, R, auxiliary functions, and loop invariants to Dafny methods for verify:

```
predicate permutation (a: seq〈int〉,b: seq〈int〉)
{
  multiset (a) == multiset (b)
}
predicate ord(a: array〈int〉, lo : int , hi : int )
requires a != null && 0 <= lo <= hi <= a . Length
reads a
{
  For all i, j ::lo <= i < j < hi ==> a[i] <= a[j]
}
predicate sorted (a: array〈int〉)
requires a != null
reads a
{
  ord(a ,0, a.Length)
}
method bubbleSort(a:array〈int〉)
requires a != null
modifies a
ensures sorted(a)
ensures permutation (a[..] , ord (a[..]))
{
  if a.Length>1
  {
   var i := 1;
   while i<a.Length
   invariant 1<= i <= a.Length;
   invariant ord(a,0,i);
   invariant permutation (a[..], ord(a[..]));
   decreases a.Length−i;
    {
      bubbleStep (a , i);
      i := i + 1;
    }
  }
}
method BubbleStep (a:array〈int〉, i:int )
requires a != null && 0 <= i<a.Length &&
ord(a ,0 , i )
modifies a
ensures ord(a ,0 , i+1)
ensures permutation (a[..] , ord (a[..]))
{
  var j := i;
  while (j>0 && a[j−1]>a[j])
  invariant 0 <= j <= i && ord (a ,0 , j ) && ord(a,
j , i+1)
  invariant 1<j+1 <= i ==> a[j−1]<= a[j+1]
  invariant permutation (a[..], ord(a[..] ))
  decreases j
  {
    a[j−1],a[j]:= a[j],a[j−1];
    j:= j−1;
  }
}
```

● Verification of the final Dafny algorithm program

The following are the main method and verification results of the BubbleSort (Fig.3).

```
method Main()
{
var a:array〈int〉 := new int[7] [4, 0, 1, 9, 7, 1, 2];
    print "Before: ", a[0], a[1], a[2], a[3], a[4], a[5], a[6], "\n";
    BubbleSort(a);
    print "After: ", a[0], a[1], a[2], a[3], a[4], a[5], a[6], "\n";
}
```

```
Dafny 2.3.0.10506

Dafny program verifier finished with 7 verified, 0 errors
Running···

Before: −13
After: 0112479
```

Fig.3 BubbleSort verification result

4 Conclusion

In this paper, we propose a system method to deduce and synthesize the Dafny programs. First, the specification of problem is described in strict mathematical language. Then, the derivation process uses program specification transformation technology to perform equivalent transformation. Furthermore, Dafny program is synthesized through the obtained recursive relationship and loop invariants. Finally, the functional correctness of Dafny program is automatically verified by Dafny verifier or online tool. Through this method, we deduce and synthesize Dafny programs for many typical problems such as the cube sum problem, the minimum (or maximum) contiguous subarray problems, several searching problems, several sorting problems. Due to space limitation, we only illustrate the development process of Dafny programs for two typical problems: the minimum contiguous subarray problem and the new local bubble sorting problem. It proves that our method can effectively improve the correctness and reliability of Dafny program developed. What's more, we demonstrate the potential of the deductive synthesis method by developing a new local bubble sorting program.

Our next plan is to use this method and combine with our previous work[26-31] to deduce and synthesize more complex nonlinear data structure algorithms, such as the binary tree or graph related algorithms.

References

[1] Leino K R. Accessible software verification with Dafny [J]. *IEEE Software*, 2017, **34**(6): 94-97.

[2] Leino K R. Dafny: An automatic program verifier for functional correctness [C]// *International Conference on Logic Programming*.Berlin: Springer-Verlag, 2010: 348-370.

[3] Wang J, Zhan N J, Feng X Y, *et al*. Overview of formal methods [J]. *Journal of Software*, 2019, **30**(1): 33-61(Ch).

[4] Leino K R, Monahan R. Dafny meets the verification benchmarks challenge [C]// *Verified Software Theories Tools Experiments*. Berlin: Springer-Verlag, 2010: 112-126.

[5] Burstall R M, Darlington J. A transformation system for developing recursive programs [J]. *Journal of the ACM*, 1977, **24**(1): 44-67.

[6] Visser E. A survey of strategies in rule-based program transformation systems [J]. *Journal of Symbolic Computation*, 2005, **40**(1): 831-873.

[7] Balog M, Gaunt A L, Brockschmidt M, *et al*. DeepCoder: Learning to write programs [C]// *Proceedings of International Conference on Learning Representations.* Washington D C : IEEE, 2017: 1-21.

[8] Bornholt J, Torlak E. Synthesizing memory models from framework sketches and litmus tests [J]. *ACM SIGPLAN Notices*, 2017, **52**(6): 467-481.

[9] Brockschmidt M, Allamanis M, Gaunt A. Generative code modeling with graphs [C]// *Proceedings of International Conference on Learning Representations.* Louisiana: ICLR, 2019: 1-24.

[10] Exlcovisser. Program-transformation.org [EB/OL]. [2007-02-14]. *http:// www. programtransformation.org.*

[11] Pettorossi A, Proietti M. Automatic derivation of logic programs by transformation [J]. *Course Notes for European Summer School on Logic*, *Language*, *and Information*, 2000, **45**(1): 1-87.

[12] Secher J P. Unfold/Fold transformation, graduate course of university of copenhagen [EB/OL]. [2001-02-12]. *http:// www.diku.dk/topps/activities/pgmtrans/unfold-fold.ps*.

[13] Pettorossi A, Proietti M. Program derivation = rules + strategies [C]// *Computational Logic*: *Logic Programming and Beyond* (*Essays in Honour of Robert A. Kowalski-Part I*). Berlin: Springer-Verlag, 2002: 273-309.

[14] Morgan C. *Programming from Specifications* [M]. Oxford: Oxford University Press, 1991.

[15] Pavlovic D, Smith D R. Software development by refinement [C]// *Formal Methods at the Crossroads*: *From Panaea to*

Foundational Support, *LNCS*. Berlin: Springer-Verlag, 2003: 267-286.

[16] Wang X, Dillig I, Singh R. Program synthesis using abstraction refinement [J]. *Proceedings of the ACM on Programming Languages*, 2017, **45**(2): 1-31.

[17] Leavens G T, Abrial J R, Batory D. Roadmap for enhanced languages and methods to aid verification [C]// *5th Int'l Conf on Generative Programming and Component Engineering*. Philadelphia: ACM Press, 2006: 221-236.

[18] Xue J Y, You Z, Hu Q. PAR: A practicable formal method and its supporting platform [C]// *International Conference on Formal Engineering Methods*. Berlin: Springer-Verlag. 2018: 70-86.

[19] You Z, Xue J Y. Formal verification of algorithmic programs based on the Isabelle theorem prover [J]. *Computer Engineering and Science*, 2009, **31**(10): 85-89(Ch).

[20] Barnett M, Leino K R M, Schulte W. The spec# programming system: An overview [C]// *International Workshop on Construction and Analysis of Safe*, *Secure*, *and Interoperable Smart Devices*. Berlin: Springer-Verlag, 2004: 1-20.

[21] Fisler K, Krishnamurthi S. Modular verification of collaboration-based software designs [C]// *European Software Engineering Conference*; *ESEC*; *ACM SIGSOFT Symposium on the Foundations of Software Engineering*; *FSE*-9. Washington D C: IEEE, 2001: 10-14.

[22] Barnett M, Chang B E, Deline R, *et al*. Boogie: A modular reusable verifier for object-oriented programs [C]// *4th International Symposium Lecture Notes in Computer Science*, Berlin: Springer-Verlag, 2006: 364-387.

[23] Moura L D, Bjorner N. Z3: An efficient SMT solver [C]// *Tools and Algorithms for Construction and Analysis of Systems*. Berlin: Springer-Verlag, 2008: 337-340.

[24] Leino K R. Automating induction with an SMT solver [C]// *Proc 13th Int'l Conf Verification*, *Model Checking*, *and Abstract Interpretation.* Berlin: Springer-Verlag, 2012: 315-331.

[25] Hu Q M, Xue J Y, You Z, *et al*. Research on the formal verification technology of several software components in the PAR platform [J]. *Computer Engineering and Science*, 2018, **40**(2): 268-274(Ch).

[26] Wang C J, He J F, Luo H M, *et al*. Model-driven formal generation and automatic validation of Dafny programs [J]. *Journal of Jiangxi Normal University*, 2020, **44**(4): 378-384(Ch).

[27] Wang C J, Yu X J, Shen D M, *et al.* A two-layer verification method for concurrent distributed shared memory algorithm based on concurrent Apla [J]. *Journal of Jiangxi Normal University*, 2020, **44**(3): 301-306(Ch).

[28] Zuo Z K, Fang Y, Huang Q, *et al.* Nonrecursive binary tree sorting algorithm and its formal proof [J]. *Journal of Jiangxi Normal University*, 2020, **44**(6): 625-6329(Ch).

[29] Zuo Z K, Lu Z H, Huang Q, *et al*. A comparative study of generic features between Apla and programming languages [J]. *Journal of Jiangxi Normal University*, 2019, **43**(5): 454-461(Ch).

[30] Zhou W X, Zuo Z K, WANG C J, *et al*. The contrastive study of generic programming in object-oriented languages [J]. *Journal of Jiangxi Normal University*, 2018, **42**(3): 304-310(Ch).

[31] Zhang Q, Wang C J, Luo H M, *et al*. The generation method and automatical transformation system of WSDL→Radl-WS [J]. *Journal of Jiangxi Normal University*, 2018, **42**(3): 298-30(Ch).

□