Nonlinear Program Construction and Verification Method Based on Partition Recursion and Morgan's Refinement Rules

WANG Changjing\textsuperscript{1,2}, CAO Zhongxiong\textsuperscript{1}, YU Chuling\textsuperscript{1}, WANG Changchang\textsuperscript{1,2}, HUANG Qing\textsuperscript{1}, ZUO Zhengkang\textsuperscript{1,2,*}

1. College of Computer Information Engineering, Jiangxi Normal University, Nanchang 330022, Jiangxi, China;
2. College of Digital Industry Academy, Jiangxi Normal University, Shangrao 334000, Jiangxi, China

Abstract: The traditional program refinement strategy cannot be refined to an executable program, and there are issues such as low verification reliability and automation. To solve the above problems, this paper proposes a nonlinear program construction and verification method based on partition recursion and Morgan's refinement rules. First, we use recursive definition technique to characterize the initial specification. The specification is then transformed into GCL (Guarded Command Language) programs using loop invariant derivation and Morgan's refinement rules. Furthermore, VCG (Verification Condition Generator) is used in the GCL program to generate the verification condition automatically. The Isabelle theorem prover then validates the GCL program's correctness. Finally, the GCL code generates a C++ executable program automatically via the conversion system. The effectiveness of this method is demonstrated using binary tree preorder traversal program construction and verification as an example. This method addresses the problem that the construction process's loop invariant is difficult to obtain and the refinement process is insufficiently detailed. At the same time, the method improves verification process automation and reduces the manual verification workload.

Key words: program construction; partition recursion; Morgan's refinement rules; loop invariant; VCG; Isabelle theorem prover

CLC number: TP311

0 Introduction

With the continuous progress of computer technology and the expansion of application areas, software security has received wide attention from the society. The use of formal methods\textsuperscript{[1-4]} to develop programs can strictly guarantee the correctness and reliability of programs. Formal methods also help to better understand the functions and behaviors of programs and meet user requirements. Therefore, formal methods have a wide range of promising applications in software development. Program derivation and verification techniques are the hot spots in the research of formal methods. The more mature techniques are Dijkstra's weakest predicate

Received date: 2022-05-21
Foundation item: Supported by the National Natural Science Foundation of China (62262031), Science and Technology Key Project of Education Department of Jiangxi Province (GJJ2200302, GJJ210307), and the Graduate Innovative Special Fund Projects of Jiangxi Province (YJS2022064)
Biography: WANG Changjing, male, Ph.D., Professor, research direction: software formal method, trustworthy software. E-mail: wej771006@163.com
* To whom correspondence should be addressed. E-mail: zhengkang2005@iscas.ac.cn

This is an Open Access article distributed under the terms of the Creative Commons Attribution License (https://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.
method\cite{5,6}, Morgan-based program derivation methods\cite{7-9}, and PAR(Partition-and-Recur) methods\cite{4}.

Dijkstra’s weakest predicate approach defines its own language rules by which programs can be developed and verified. The advantage is that the development process is strictly based on mathematical logic and formal verification is performed during the development process. However, the verification process is mostly manual and less automated; and the abstract programs obtained by this method cannot be converted into executable programs.

The PAR method transforms the initial specification by its own defined rules to obtain recursive equations and loop invariant, which eventually lead to an abstract Apla program\cite{4}. The advantage of this method is that the derivation process is guided by a system with its own derivation patterns and rules, which are more widely applicable. However, the procedure of program derivation using this method is not sufficiently refined and the obtained Apla abstract program is not mechanically proven and has a low automation procedure.

This paper proposes a program construction and verification method based on partition recursion and Morgan’s refinement rule. The method first transforms the initial program specification by recursive definition technique, and then obtains recurrence relations and loop invariant by using partitioned recursion rules\cite{10,11}.

Then, based on the loop invariant, the program specification is gradually refined using Morgan refinement rules to finally obtain a reliable abstract program. The verification conditions are obtained by VCG\cite{12}(Verification Condition Generator) and then verified by Isabelle\cite{12-14} theorem provers. After the verification, the obtained abstract program is then generated into a C++ program through the conversion system\cite{15} to realize the whole process from abstract specification to abstract program\cite{15,16}, then mechanical verification\cite{16,17}, and to executable program.

The paper is organized as follows. Section 1 focuses on detailing the program construction and verification methods based on partition recursion and Morgan’s refinement rule; Section 2 takes the binary tree preorder problem\cite{18,19} as an example to develop and verify the binary tree problem using the methods proposed in this paper as a guide; Section 3 concludes the whole paper.

1 Proposed Method of Nonlinear Program Construction and Verification

This section focuses on the method proposed in this paper. The method in this paper contains a complete set of program construction and verification methods, which mainly consists of four steps, as shown in Fig.1.

---

**Fig. 1 Proposed method of nonlinear program construction and verification**
1.1 Initial Specification Generation

A program specification is a detailed description of a program’s functions. It expresses a precise description of the problem in an easy-to-understand manner for the implementer. The program specification consists primarily of the program’s pre-assertion $P$ and post-assertion $Q$. $P$ and $Q$ are conditions that must be met before and after program input, respectively.

Hoare’s formal specification is primarily in the form of triples $(P)S(Q)$\cite{13}, but this paper primarily uses Morgan’s specification representation $w:[P, Q]$. Morgan rewrote the Hoare triples representation to make the specification representation more compact and easier to refine into code.

1.2 Program Refinement and Generation

1.2.1 Initial specification transformation

Following the generation of the initial specification, the attributes of first-order logic are typically used to gradually refine the specification transformation from the post-predicate. After the generation of the binary tree preorder specification in Section 1.1, the initial specification transformation needs to be performed on it. The following is the specific transformation procedure.

Due to the binary tree’s easy decomposition, this paper first introduces the recursive function of the binary tree and strengthens the post-predication of initial specification. Then, Morgan’s framework variables are introduced to convert the initial reduction’s triple form into a Morgan-specific symbolic form. Figure 2 depicts the entire initial specification transformation process. This recursive technique reduces the difficulty of changing the binary tree specification. At the same time, recursion technology provides a foundation for subsequent programs to use partition recursion to find loop invariant.

1.2.2 Loop invariant derivation

Acquiring loop invariant has always been a difficult point in formal derivation, and acquiring tree structure loop invariant is even more difficult. To address this issue, this paper proposes the use of partition recursion in conjunction with recursive definition technology to derive the loop invariant of tree structure.

The basic idea behind partition recursion is to divide a large, difficult-to-solve problem into some smaller problems of the same scale, and then find the recurrence relationship via the relationship between sub-problems.

First, this paper decomposes the original problem using the recursive definition technique. The recurrence relationship between the problems is then determined based on the relationship between the sub-problems in the decomposition process. Finally, the binary tree’s loop invariant is discovered using the recursive relationship. Figure 3 depicts the specific process flow.

![Fig. 2 Initial specification conversion process](image)

![Fig. 3 Loop invariant derivation process](image)

1.2.3 Morgan’s rules refinement

The initial statute and loop invariant have been obtained in Sections 1.2.1 and 1.2.2. Now it is necessary to derive the initial specification to the GCL program based on loop invariant, which are used as Morgan’s refinement rules in this paper. Morgan’s refinement rules are used to gradually refine the abstract specification into GCL\cite{9} code with strong execution, with each refinement step being small and easy to handle. Using the Morgan’s refinement rules, each step of the refinement process must be guaranteed to be justified and proven. As a result, the program obtained is extremely reliable.

The rules used in the Morgan refinement method are specified in Table 1\cite{5}.

1.3 GCL Program Automation Verification

The GCL program obtained in Section 1.2 needs to be mechanically verified by generating verification conditions through VCG and by Isabelle. The specific steps
are shown as follows.

1.3.1 Verification condition generation

The automatic generation of verification conditions is mainly done through VCG, which aims at converting the verification of Hoare triples into the verification of assertions. The advantage of this is that the person or machine verifying the program does not need to know anything about Hoare logic to be able to prove the program.

VCG is a verification condition generator whose working process relies on two conversion functions: pre (this function is different from the following binary tree preorder function Prebt) and vc function. pre function mainly converts the imperative program into specific Hoare logic rules, and vc function mainly converts the result of pre function into specific verification conditions. The specific VCG working schematic is shown in Fig.4.

![Flow step](image)

Fig. 4 Working principle of VCG
1.3.2 Isabelle assisted verification

Manual and mechanical verification are the two types of formal verification technology. Manual verification is more difficult and prone to errors. Mechanical verification is based on mathematical logic, and it is far more efficient and reliable than manual verification. The mechanical theorem prover\(^1\) is further subdivided into automatic and interactive theorem provers.

Isabelle, an automatic theorem prover, is chosen for auxiliary verification in this paper, primarily because Isabelle has the following advantages.

1) The powerful rule base

Isabelle has a powerful rule base, in which every theory contains many related rules and theorems. More importantly, users can add theorems and prove them based on their own proof requirements. If the newly added theorem passes the proof, the Isabelle system will save the newly added lemma and use it in the theorem’s subsequent proof.

2) The Sledgehammer tool

The Isabelle system’s Sledgehammer\(^{12,13}\) tool is extremely powerful, and it can invoke several other automatic theorem provers to find theorems, including E, SPASS, Vampire, and others. In the Isabelle operator interface, the user only needs to click on the apply application in the Sledgehammer button and Sledgehammer can automatically generate the proof process. The user then clicks on the proof process to automatically insert the proof process into the Isabelle script.

1.4 Conversion from GCL Program to Executable Program C++

The GCL programs verified in Section 1.3 are still abstract and cannot be compiled and executed on a computer. As a result, the GCL program will need to be converted into an executable file. First, we can convert the GCL program to an Apla program. GCL and Apla can be converted because they are abstract imperative programs with similar syntax. Then, using the team’s Apla to C++ program automatic converter, convert the above Apla abstract program into an imperative C++ program. The team’s Apla to C++ conversion system is depicted in Fig.5.

With these four steps, this paper realizes the whole process of program development, verification and transformation.

2 Example: Binary Tree Preorder Problem (BTPP)

2.1 Initial Specification Generation of BTPP

Preorder traversal is a way of traversing a binary tree. By preorder traversal, we mean that when traversing a binary tree, we first traverse the root node, then the left subtree, and finally the right subtree. In this paper, we use \( T \) to represent a binary tree, and the Prebt \((T)\) function to represent the preorder traversal of the entire binary tree \( T \). Therefore, the binary tree preorder traversal specification can be expressed as follows:

\[
\text{Prebt}(T) = \begin{cases} 
\text{true, } \text{Prebt}(T) \\
\text{Prebt}(T.r) \end{cases}, \quad T = %
\]

(1)

2.2 Program Refinement of BTPP

2.2.1 Initial specification transformation

There are two types of binary tree traversal algorithms: recursive traversal and nonrecursive traversal. The goal of this paper is to deduce the binary tree’s preorder nonrecursive algorithm program.

This section focuses on the initial specification transformation, which is the transformation of the posterior assertion Prebt. And we know that the process of binary tree preorder traversal is to traverse the root node, then the left subtree, and then the right subtree. So the Prebt function can be written first as follows:

\[
\text{Prebt}(T) = \begin{cases} 
[ ] \\
[T.n] \uparrow \text{Prebt}(T.l) \uparrow \text{Prebt}(T.r) \end{cases}, \quad T \neq %
\]

(2)

We decompose this function according to the content of the Prebt function, and we can obtain the following derivation process:

\[
\text{Prebt}(T) = \begin{cases} 
[T.n] \uparrow [T.l] \uparrow [T.r] \\
[T.n] \uparrow [T.l.n] \uparrow [T.l.l] \uparrow [T.l.r] \uparrow [T.r] \\
[T.n] \uparrow [T.l.n] \uparrow [T.l.l.n] \uparrow [T.l.l.l] \uparrow [T.l.l.r] \uparrow [T.l.r] \uparrow [T.r]
\end{cases}
\]
Figure 6 depicts the binary tree’s change process as a result of the recursive function Prebt. As shown in the derivation and Fig.6, the root node is retained first when traversing a binary tree recursively. The left subtree of the tree will then be traversed, followed by the right subtree of the tree. As a result, three variables, \( X \), \( q \), and \( S \), are introduced. The nodes that have been traversed are stored in the sequence \( X \). The sequence \( S \) is used to store the nodes to be traversed, while \( q \) is used to store the subtree of \( T \) that is about to be traversed. The original specification can then be refined as needed:

\[
X, q, S: \{ \text{true, } X = \text{Prebt}(T) \}
\]

2.2.2 Loop invariant derivation

We have obtained the initial specification after the transformation, next we need to obtain the corresponding loop invariant. In this paper, the loop invariant are obtained according to the division recursion, i.e., we find the law and obtain the loop invariant in the process of binary tree traversal in the preorder.

As illustrated in Fig.7, for the traversed subtree \( q \), the subtree’s head node is always stored in \( X \), and \( q \) continues to traverse the subtree’s left subtree. The right subtree of the subtree is still in \( S \).

After traversing subtree \( q \), the next round will continue to find nodes from sequence \( S \) to traverse. The sequence \( S \)’s length will be reduced accordingly. The definition of the recurrence relation \( F \) function can be derived:

\[
F = \begin{cases} 
F([ ]) = [ ] \\
F([q] \uparrow S) = \text{Prebt}(q) \uparrow F(S)
\end{cases}
\]

We can see from the previous Prebt and \( F \) functions that the binary tree preorder traversal process always places the nodes traversed by \( q \) in the sequence \( X \). Then \( q \) continues to traverse the subtree’s remaining nodes to the left. After traversing the left subtree, \( q \) will look for nodes in the sequence \( S \) to continue traversing, as shown in Fig. 8. So the loop invariant is \( \text{inv = Prebt}(T) = X \uparrow \text{Prebt}(q) \uparrow F(S). \)

---

Fig. 6  The process of binary tree change under the action of recursive function

Fig. 7  Binary tree traversal node change process diagram
2.2.3 Morgan’s rules refinement

1) Loop variable initialization

Based on loop invariant obtained in Section 2.2.2, the specification can be refined using Strengthen post-condition in Table 1, and the specification can then be obtained as:

\[ X, q, S : \text{true; } \text{inv } \land X = \text{Preb}(T) \]  \hspace{1cm} (5)

According to the specification, inv is used as an intermediate assertion connecting the pre-predicate and the post-predicate, then (5) is refined using Sequential composition in Table 1:

\[ X, q, S : \text{true; } \text{inv } \]  \hspace{1cm} (6)

\[ X, q, S : \text{inv; } \text{inv } \land X = \text{Preb}(T) \]  \hspace{1cm} (7)

Assignment in Table 1 is used to refine (6), and the following formula is obtained:

\[ X, q, S : \text{true; } \text{inv } \subseteq X, S, q = [] \]  \hspace{1cm} (8)

Because:

\[ \text{true } \Rightarrow \text{inv}[X, S, q = []], T \]

\[ = \text{Preb}(T) = [X \uparrow \text{Preb}(q) \uparrow F(S)] [X, S, q = []], T \]

\[ = \text{Preb}(T) = [\text{Preb}(T) \uparrow F] \]

\[ = \text{Preb}(T) = [\text{Preb}(T) \uparrow F] \]

\[ \text{true } \]

So the first part of the refinement is established.

2) Loop process refinement

Formula (7) can be refined using Repetition in Table 1, and \( \neg(X = \text{Preb}(T)) \) is a loop condition and can be equivalently converted to \( q \neq \% \lor (q = \% \land S \neq []) \), then the following equation can be obtained:

\[ \text{do } q \neq \% \lor (q = \% \land S \neq []) \]

\[ X, q, S : \left[ \text{inv } \land (q \neq \% \lor (q = \% \land S \neq [])), \text{inv } \land 0 \leq v < v_0 \right] \text{od} \]

\[ \text{true } \]

(9)

The variable \( v \) in the above equation \( 0 \leq v < v_0 \) is the marker for the terminability proof. \( v \) needs to satisfy two conditions: first, it must be guaranteed to be decremented in each loop, and second, it cannot be decremented to a negative number. In this paper, we need to find the specific terminability variable \( v \).

We can see here that the variable \( X \) stores the nodes that are traversed. And the number of nodes in \( X \) keeps increasing. When the loop is completed, the number of nodes stored in \( X \) is equal to the number of nodes in the binary tree \( T \). So, we can select \( (|T| - \#X) (|T|) \) is the node value of the whole binary tree and \( X \) is the stored traversed node value) as the variable \( v \).

We can simplify the equation by substituting \( v = |T| - \#X \) for \( 0 < v < v_0 \):

\[ 0 < v < v_0 \Rightarrow 0 < |T| - \#X < |T| - \#X \Rightarrow \#X < \#X < \#X \]

Therefore (9) is simplified to: \( X, q, S : \text{inv } \land (q \neq \% \lor (q = \% \land S \neq [])), \text{inv } \land \#X < \#X < |T| \).

Under the premise of \( q \neq \% \lor (q = \% \land S \neq []) \), the above formula can be further refined according to Selection in Table 1.

\[ \text{if } q \neq \% \rightarrow \]

\[ X, q, S[q \neq \% \land \text{inv } \land \left( q \neq \% \lor (q = \% \land S \neq []), \text{inv } \land \#X \leq \#X < |T| \right) \]

\[ (q \neq \% \land S \neq []) \rightarrow \]

\[ X, q, S[q \neq \% \land S \neq []] \land \text{inv } \land (q \neq \% \lor (q = \% \land S \neq [])), \text{inv } \land \#X \leq \#X < |T| \]

\[ \text{fi } \]

The above equation (10) can be further refined according to Assignment in Table 1:

\[ X, q, S[q \neq \% \land \text{inv } \land (\#X \leq \#X < |T|)] \subseteq \]

\[ S, X, q = [q \uparrow X \uparrow [q, n], q, l; \]

because
According to the definition of inv, Prebt, and F functions, the above formula can be simplified:

Left side:
\[ q \neq \% \land \text{inv} \Rightarrow \text{inv} \land (\#x_n < \#X \in |T|) [S,X,q]. \]
\[ q.r \uparrow S.X \uparrow [q.n],q,I] \]

Right side:
\[ \text{Prebt}(T) = X \uparrow \text{Prebt}(q) \uparrow F(S) \land (\#x_n < \#X \in |T|) \]
\[ = X \uparrow [q.n] \uparrow \text{Prebt}(q) \uparrow F(S) \land (\#x_n < \#X \in |T|) \]
\[ \Rightarrow \text{Prebt}(T) = X \uparrow [q.n] \uparrow \text{Prebt}(q.l) \uparrow F(S) \land \text{true} \]

So \[ q \neq \% \land \text{inv} \Rightarrow \text{inv} \land (\#x_n < \#X \in |T|) [S,X,q]. \]
\[ q.r \]
\[ S,X \uparrow [q.n],q.l]=\text{True} \]

Similarly, the above equation (11) can be further refined according to Assignment in Table 1.

\[ X.q.S;\{(q \neq \% \land S \neq []) \land \text{inv} \land (\#x_n < \#X \in |T|) \} \]
\[ \subseteq q.S := S[S.h],S[S.h+1...S.r] ; \]

According to the above refinement, the program can be replaced by:

\[ \text{if } q \neq \% \rightarrow \]
\[ S.X.q := [q.r] \uparrow S.X \uparrow [q.n],q.1 ; \quad (12) \]
\[ (q = \% \land S \neq []) \rightarrow \]
\[ q.S := S[S.h],S[S.h+1...S.r] ; \quad (13) \]
\[ \text{fi} \]

Combined with the above formula, the final GCL program is:
\[ X.q.S := [],T[] \]

\[ \text{do } q \neq \% \lor (q = \% \land S \neq []) \rightarrow \]
\[ \text{if } q \neq \% \]
\[ S,X.q := [q.r] \uparrow S.X \uparrow [q.n],q.1 ; \]
\[ (q = \% \land S \neq []) \rightarrow \]
\[ q.S := S[S.h],S[S.h+1...S.r] ; \]

2.3 GCL Program Automation Verification of BTPP

2.3.1 Verification condition generation

In order to verify the GCL code obtained in Section 2.2.3 above, we must first construct the lemma Pre_ bt. Then, using the GCL code from above, we must write the corresponding code in Isabelle. The two have the same code structure, and the Isabelle code is as follows:

\[ \text{lemma Pre_ bt}: \ "\text{VARS } X \ S \ T \ q" \]
\[ \{\text{true} \} \]
\[ X := []; q := T; S := []; \]
\[ \text{WHILE } (q \neq \text{Tnull} \land q = \text{Tnull} \land S \neq []) \]
\[ \text{INV } \{\text{Prebt}(T) = X @ \text{Prebt}(q) @ F(S) \} \]
\[ \text{DO} \]
\[ \text{IF } q \neq \text{Tnull} \text{ THEN} \]
\[ X := X @ \text{data q} ; \]
\[ S := \text{tree q}@S ; \]
\[ q := \text{tree q} \]
\[ \text{ELSE IF } q = \text{Tnull} \land S \neq [] \text{ THEN} \]
\[ q := (\text{hd S}) ; \]
\[ S := (\text{tl S}) \]
\[ \text{ELSE} \]
\[ \text{SKIP} \]
\[ \text{FI} \]
\[ \text{OD} \]

We then use the apply vcg command to get three verification conditions, as shown in Fig.9.

![Fig. 9 Verification condition generation](image-url)
2.3.2 Isabelle assisted verification

Some auxiliary functions and lemmas are required to verify the above three sub-goals. The following are the specific steps:

1) Define two recursive functions `Prebt`, `F`:
   Function 1:
   ```
   primrec Prebt :: "'a BTree ⇒ 'a list"
   where
   "Prebt Tnull = []" | "Prebt (BT t1 x t2) = [x] @ (Prebt t1) @ (Prebt t2)"
   ```
   Function 2:
   ```
   fun F :: "'a BTree list ⇒ 'a list"
   where
   "F [] = []" | "F (x#xs) = (Prebt x) @ (F xs)"
   ```

2) Create the lemmas `prebt_rule` and `F_rule` to prove the final program.

   **Lemma 1:**
   ```
   lemma prebt_rule [simp]: "q ≠ Tnull ⇒ Prebt q = data q @ Prebt (ltree q) @ Prebt (rtree q)"
   apply (induct q)
   apply auto
   done
   ```

   **Lemma 2:**
   ```
   lemma F_rule [simp]: "xs ≠ [] ⇒ F xs = Prebt (hd xs) @ F tl xs"
   apply (induct xs)
   apply auto
   done
   ```

3) Prove the three sub-goals:
   ```
   apply auto
   ```
   With the auto command in 3) above, we can get a successful result of verification. This is shown in Fig.10.

---

2.4 Conversion from GCL Program to Executable Program C++ of BTPP

For the verified GCL program in Section 2.2.3, it is a non-executable abstract program that needs to be converted to executable C++. First, the GCL program needs to be equivalently converted to an Apla program because their syntax is similar. The resulting Apla program is then passed through our team’s conversion system to generate a C++ program, and the conversion result is shown in Fig.11. Finally, the converted C++ program is compiled and run, and the running result is shown in Fig.12.

3 Conclusion

This paper proposes a new method for building and verifying nonlinear programs. Using the preorder traversal of a binary tree as an example, an abstract program GCL is created from the initial specification. Isabelle validates GCL programs before converting them to executables using the C++ conversion platform. The following are the primary benefits of this paper:

1) Partition recursion is used throughout the program construction process to derive loop invariant. The specification is then gradually refined by Morgan’s refinement rule based on loop invariant. This method ensures the acquisition of loop invariant while also refining the program.

2) The method first employs VCG to generate verification conditions automatically. The verification conditions are then mechanistically verified using Isabelle. This greatly improves verification automation.
This method improves the method’s integrity by converting the obtained abstract program GCL into a C++ executable program using the C++ conversion platform.

Our next step is to use this approach and combine it with our previous work[20-25] to derive and synthesize more complex algorithms for nonlinear data structures, such as those related to binary trees or graphs.

References


