Automatic Algorithm Programming Model Based on the Improved Morgan’s Refinement Calculus

ZHOU Zhengkang1, HU Ying1, HUANG Qing1, WANG Yuan2, WANG Changjing†

1. School of Computer Information Engineering, Jiangxi Normal University, Nanchang 330022, Jiangxi, China;
2. School of Software, Jiangxi Normal University, Nanchang 330022, Jiangxi, China

© Wuhan University 2022

Abstract: The automatic algorithm programming model can increase the dependability and efficiency of algorithm program development, including specification generation, program refinement, and formal verification. However, the existing model has two flaws: incompleteness of program refinement and inadequate automation of formal verification. This paper proposes an automatic algorithm programming model based on the improved Morgan’s refinement calculus. It extends the Morgan’s refinement calculus rules and designs the C++ generation system for realizing the complete process of refinement. Meanwhile, the automation tools VCG (Verification Condition Generator) and Isabelle are used to improve the automation of formal verification. An example of a stock’s maximum income demonstrates the effectiveness of the proposed model. Furthermore, the proposed model has some relevance for automatic software generation.

Key words: automatic algorithm programming model; program refinement; VCG; Isabelle; Morgan’s refinement calculus

CLC number: TP 311

0 Introduction

The development of reliable software has become a critical point on which we should concentrate, particularly for systems with high safety requirements, such as aerospace control systems and medical equipment systems. Even minor errors in these systems can result in massive losses.

Formal methods are mathematically based techniques for specifying, developing, and testing computer software and hardware systems. Their mathematical basis is the formal logic systems consisting of formal languages, semantics, and proof systems. Formal methods are increasingly being used with appropriate levels of rigor at various stages of a computing system’s lifecycle. It contributes significantly to reliability and security of software.

Algorithm program correctness is the key to software security. However, programmers often write code intuitively. They hope that if there are small misjudgements, those errors will be exposed in the test. The algorithm obtained in this way is very error-prone. It may cause some hidden issues with software security. Automatic algorithm programming model has the potential to improve the reliability and efficiency of algorithm program development.

Automatic algorithm programming model consists of three core processes: specification generation, program refinement, and formal verification. Among the three processes, specification generation uses formal languages and proof systems. Formal methods to describe user requirements; program refine-
ment turns a specification into an executable program by gradually lowering the level of abstraction; and formal verification verifies the correctness of abstract programs generated during program refinement.

The process of specification generation is relatively simple. As a result, in the following, we would introduce a variety of automatic algorithm programming models based on Dijkstra’s weakest precondition method\(^\text{(8-9)}\), Morgan’s refinement calculus\(^\text{(10-12)}\) and PAR (Partition and Recur) method\(^\text{(13)}\) from the two core processes of program refinement and formal verification. In program refinement, the three methods all provide formal refinement rules, with Morgan’s refinement calculus rules being the most numerous. However, only PAR method implements the complete program refinement process, from program specification to executable program (C++ program). In formal verification, the three methods all generate verification conditions manually and then perform manual verification. You et al\(^\text{(14)}\) enhanced the PAR method by replacing manual verification with mechanical verification via Isabelle. Table 1 compares the above mentioned automatic algorithm programming models in terms of program refinement and formal verification.

<table>
<thead>
<tr>
<th>Method</th>
<th>Program refinement</th>
<th>Formal verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra’s weakest precondition method</td>
<td>the weakest precondition transformation rules</td>
<td>manually generated, manually verified</td>
</tr>
<tr>
<td>Morgan’s refinement calculus</td>
<td>Morgan’s refinement calculus rules</td>
<td>specification to abstract program, manually generated, manually verified</td>
</tr>
<tr>
<td>PAR method</td>
<td>partition and recur rules + C++ generation system</td>
<td>specification to executable program, manually generated, manually verified</td>
</tr>
<tr>
<td>Improved PAR method</td>
<td>partition and recur rules + C++ generation system</td>
<td>specification to executable program, mechanically verified by Isabelle</td>
</tr>
</tbody>
</table>

The current automatic programming model has two flaws. The first is the incompleteness of program refinement, and the second is the inadequate automation of formal verification. This paper presents an automatic algorithm programming model based on improved Morgan’s refinement calculus to address these two issues. For the first deficiency, Morgan’s refinement calculus is extended by adding quantifier specification transformation rules, and the C++ generation system is designed and implemented to complete the refinement process from specification to executable program. For the second deficiency, verification conditions are generated automatically based on VCG and mechanically verified by Isabelle. This is a significant step toward achieving high reliability and automation of the verification process.

1 Automatic Algorithm Programming Model

The flow chart of automatic algorithm programming model proposed in this paper is shown in Fig. 1.

1.1 Specification Generation

By analyzing user requirements, we generate an initial formal specification with pre- and post-conditions in specification generation. In this model, the specification is written as \(w:\[P,Q]\)\(^\text{(15)}\), where \(P\) is the precondition, \(Q\) is the postcondition, and \(w\) is the list of frame variables. The variables whose values may change, i.e. those in the list \(w\) above, are called the frame variables (or, more concisely, simply the frame) of the specification. The code that satisfies the specification \(w:\[P,Q]\) may only change the value of one or more variables in \(w\).

1.2 Specification Simplification

The generated specification could always be simplified to a lower complexity equivalent specification based on the analysis of concrete features of the specific field. We adopt the methods of problem division and specification transformation to simplify the specification.
1.3 Abstract Program Refinement Method

**Definition 1(Sat)** $\text{Sat}(C, w: [P, Q])$ is a predication that asserts the IMP(Imperative Programming Language) program $C$ satisfies the specification $w: [P, Q]$.

To make this idea more formal, the assertion that means $X$ is refined by $Y$ (or, equivalently, $Y$ is a refinement of $X$) is denoted by $X \subseteq Y$, which is defined as follows:

**Definition 2(Refinement)** $w: [P, Q] \subseteq w: [P', Q']$ if and only if

\[
\forall C. \text{IMP-Sat}(C, w: [P', Q']) \implies \text{Sat}(C, w: [P, Q]).
\]

Thus $w: [P, Q]$ is refined by $w: [P', Q']$ if every IMP program that satisfies the specification $w: [P', Q']$ also satisfies the specification $w: [P, Q]$.

The process of abstraction program refinement begins with the specification and ends with the abstract program (IMP program). Each step of the process adheres to the rules for demonstrating correctness. If these rules are applied strictly, we could end up with a correct IMP program.

In this model, the refinement of the abstract program is based on Morgan’s refinement calculus. Morgan’s refinement calculus provides refinement calculus rules for transforming one specification into another specification or an IMP statement. However, appropriate rules must be chosen by analyzing the specification. In one word, the specification should always be transformed in an equivalent manner to match the appropriate rules.

The Morgan’s refinement calculus rules used in this model are shown in Table 2.

### Table 2 Morgan’s refinement calculus rules

<table>
<thead>
<tr>
<th>Rule name</th>
<th>Proviso and rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strengthen Postcondition Rule</td>
<td>if $Q' \Rightarrow Q$ then $w: [P, Q] \subseteq w: [P, Q]$</td>
</tr>
<tr>
<td>Sequential Composition Rule</td>
<td>$w: [P, Q] \subseteq w: [P, R], w: [R, Q]$</td>
</tr>
<tr>
<td>Assignment Rule</td>
<td>if $P \Rightarrow Q[E]$ then $w: [P, Q] \subseteq w: E$</td>
</tr>
<tr>
<td>Repetition Rule</td>
<td>if $GG = G_1 \lor G_2 \cdots \lor G_s$ and $V$ is a variant then $w: [P, P \land \neg GG] \subseteq$</td>
</tr>
<tr>
<td></td>
<td>$\text{do } G_i \rightarrow w: [P \land G_i \land V = V_o, P \land (0 \leq V &lt; V_o)]$</td>
</tr>
<tr>
<td></td>
<td>$\cdots$</td>
</tr>
<tr>
<td></td>
<td>$G_s \rightarrow w: [P \land G_s \land V = V_o, P \land (0 \leq V &lt; V_o)]$</td>
</tr>
<tr>
<td>od</td>
<td></td>
</tr>
<tr>
<td>Following Assignment Rule</td>
<td>$w, x: [P, Q] \subseteq w, x: [P, Q[x/E]]; x := E$</td>
</tr>
<tr>
<td>Weaken Precondition Rule</td>
<td>if $P \Rightarrow P'$ then $w: [P, Q] \subseteq w: [P', Q]$</td>
</tr>
<tr>
<td></td>
<td>if $P \Rightarrow G_i \lor G_2 \cdots \lor G_s$ then $w: [P, Q] \subseteq$</td>
</tr>
<tr>
<td></td>
<td>$G_i \rightarrow w: [P \land G_i, Q]$</td>
</tr>
<tr>
<td></td>
<td>$\cdots$</td>
</tr>
<tr>
<td></td>
<td>$G_s \rightarrow w: [P \land G_s, Q]$</td>
</tr>
<tr>
<td>Selection Rule</td>
<td>fi</td>
</tr>
</tbody>
</table>
Quantifiers are used in specification to describe the preconditions and postconditions, but the refinement rules outlined above are insufficient to refine such specifications. In this model, we add some quantifier specification transformation rules, as shown in Table 3. A quantifier, \( q_a \) is the binary operation corresponding to \( Q_a \), which are used to transform the precondition or postcondition equivalently. The formalization of program refinement is optimized by expanding the available formal rules.

**Table 3 Quantifier specification transformation rules**

<table>
<thead>
<tr>
<th>Rule name</th>
<th>Proviso and rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope Split Rule</td>
<td>((Q \cdot i = r(i)) \Rightarrow (Q \cdot i = b(i) \lor Q \cdot i = \neg b(i)))</td>
</tr>
<tr>
<td>Single Scope Rule</td>
<td>((Q \cdot i = k(f(i))) = f(k))</td>
</tr>
<tr>
<td>Constraint Variable Renaming Rules</td>
<td>((Q \cdot i = r(i)) \Rightarrow (Q \cdot k = r(k) \land k \text{ is not a free variable of } r(i) \lor f(i)))</td>
</tr>
<tr>
<td>Cross Product Property Rule</td>
<td>((Q \cdot i, J = r(i)) = (Q \cdot r(i), J = f(i, J)))</td>
</tr>
</tbody>
</table>

\( J \) represents any non-empty set of constraint variables.

**1.4 Formal Verification**

Artificial analysis during the refinement of an abstract program may result in program errors. Therefore, formal verification is required to ensure the correctness of the IMP program. Formal verification is divided into two steps. First, based on VCG, verification conditions are generated automatically by the IMP program. Second, the verification conditions are mechanically verified by Isabelle.

### 1.4.1 VCG

Hoare logic provability could be reduced to assertion language provability. In particular, if we want to prove a Hoare logical triple \( \{P\} c Q\), we can compute an assertion \( A \) from it, and then \( \{P\} c Q\) is provable if \( A \) is provable. \( A \) is referred to as a verification condition, and the function that computes \( A \) is referred to as a verification condition generator, or VCG.[10]

The precondition and postcondition are provided as annotations to the VCG for all algorithm programs, and the invariant is additionally provided for WHILE loops. Figure 2 depicts the VCG code template for generating verification conditions.

The advantage of using VCG is that no human or machine is required to know the Hoare logic when trying to prove the generated validation conditions.[10]

#### 1.4.2 Isabelle theorem prover

Isabelle[16-19] is an auxiliary theorem prover which provides theorem proving with strong theoretical and technical support. Compared with other traditional theorem provers, the Isabelle theorem prover has the following distinguishing features. 1) Isabelle can handle multiple object logics. 2) Isabelle is in favor of new logic, which could be implemented by defining object logic’s concretesyntax, abstract syntax, and inference rules. 3) Isabelle’s verification language is Isar. It could accurately describe the system under test. Furthermore, Isar benefits from a wide range of data types, logic systems, and so on. 4) The editing tool jEdit could verify each step in the theorem proof, and display the proof status in the output panel.

**1.5 IMP Program Refinement**

We propose IMP program refinement to improve the completeness of program refinement. This refinement allows for the conversion of formally verified IMP programs to executable C++ programs. As a result, an automatic system known as the C++ generation system[20-22] is created, and the overall structure of the system is depicted in Fig. 3.

The system mainly consists of four modules: lexical analysis, syntactic analysis, semantic analysis, and program transformation.

Lexical analysis transforms the IMP program character sequences into equivalent internal symbol sequences using IMP’s lexical rules. Syntactic analysis...
identifies IMP syntactic items from internal symbol sequences using IMP syntax rules. Semantic analysis applies semantic processing to the syntactic items that have been identified. If the results of the above three analyses are incorrect, error messages will be reported in error handling. On the other hand, if the result is correct, the IMP program will be converted into a C++ program.

2 Example: A Stock’s Maximum Income

2.1 Generating Problem Specification

2.1.1 Problem description

Suppose you could buy one share or sell your share per day at the stock’s closing price. A stock’s income is equal to the selling price minus the buying price. The problem with the maximum income of a stock is to calculate the maximum sum of the incomes from all possible trades over n days.

Figure 4 is an example of the volatility of a stock's closing price over n days.

2.1.2 Specification generation

The closing price of this stock in n days is stored in the array \(P[0..n-1](n=p.len, n > 1)\). According to the problem description, the essence of solving the problem is to find all the ascending segments \(P[i..j](P[i] and P[j] are the minimum value and the maximum value of the ascending segment respectively), then calculate the income \(P[j]–P[i]\), finally add up the income of each ascending segment and the result of addition is maximum income.

Therefore, the maximum income of a stock in n days could be expressed by \(\text{maxincome}(n, P)\) as:

\[
\text{maxincome}(n, P) = \sum_{i=1}^{n-1} \text{ income } \left(P[i] \to P[j] \to P[j+1] \to \ldots \to P[i-1]\right)
\]

The expression of the \(\text{maxincome}(n, P)\) is required when describing the specification of this problem. However, the specification would be complicated due to the complexity of this expression.

2.1.3 Specification simplification

In order to reduce the complexity of the specification, we could simplify the expression of the maximum income \(\text{maxincome}(n, P)\).\n
\[
P[j]–P[i] = (P[i+1]–P[i]) + (P[i+2]–P[i+1]) + \ldots + (P[j]–P[j-1])\]

For example, the income from buying on day n–2 and selling on day n–5 is \((P[n-4]–P[n-5]) + (P[n-3]–P[n-4]) + (P[n-2]–P[n-3])\). Therefore, we could divide the array \(P[0..n-1]\) into the n–1 segments \(P[0,1], P[1,2], P[2,3], \ldots, P[n-2, n-1]\), i.e. \(P[i-1,i] (1 \leq i \leq n-1)\). The maximum income could then be calculated by adding the incomes of all ascending segments \([23]\). It is obviously that whether \(P[i-1,i]\) is an ascending segment or not is determined by the truth value of \(P[i-1] < P[i]\).

Therefore, \(\text{maxincome}(n, P)\) could be simplified to:

\[
\text{maxincome}(n, P) = \sum_{i=1}^{n-1} \text{ income } \left(P[i-1] < P[i]\right)
\]

We introduce a variable \(s\) to represent the maximum income calculated at some point in the process, then the postcondition of this problem could be expressed as: \(s = \text{maxincome}(n, P)\). Because the income of a stock is calculated from the second day, the precondition is \(n > 1\). In conclusion, the specification is as follows:

\[
s[n > 1, s = \text{maxincome}(n, P)]
\]

2.2 Refining Abstract Program

Figure 5 shows the whole process of refining abstract program, where \(\text{①, ②, \ldots, ④}\) represent program specifications and \(1, 1, \ldots, 1\) represent refined IMP statements. The abstract program refinement ends when all of the specifications have been refined into IMP statements.
2.2.1 Choosing the invariant

We introduce a variable $m(0 < m \leq n)$ to traverse the array $P_{[0..n-1]}$ from left to right. When the subarray $P_{[0..m-1]}$ is traversed, the maximum income $\text{maxincome}(m, P)$ could be calculated, as shown in Fig. 6. Therefore the loop invariant could be defined as $\text{inv}(s, n) = \text{maxincome}(m, P) \leq n - m$, which is similar to the postcondition $\text{post}(s, m) = \text{inv}(s, n) = n - m$, so the program specification could be replaced equivalently by $m, s; [n > 1, \text{inv} \land m = n]^{10}$.

![Fig. 5 Process diagram of abstract program refinement](image)

![Fig. 6 Traversal process of calculating the maximum income](image)

2.2.2 Establishing the invariant

Before we can refine it to a loop, the general pattern for a specification should be something like $w; [\text{inv}, \text{inv} \land \neg G]$. The following refinement process is required to get our refined specification into this shape.

According to Sequential Composition Rule, specification $\Box$ could be refined as:

$m, s; [n > 1, \text{inv} \land m = n]$ (Sequential Composition Rule)

Thus, the result of "Establishing the Invariant" is:

$m, s; [n > 1, \text{inv}]$; $m, s; [\text{inv}, \text{inv} \land m = n]$ (Assignment Rule) (2)

According to Repetition Rule, specification $\Box$ could be refined to the following loop statement:

$m, s; [\text{inv} \land m < n \land V_0 = V, \text{inv} \land m < n \land V_0 = V']$ (4)

where $V_0$ is the previous value of $V$.

2.2.3 Refining to create a loop

$V$ is defined as $n - m$, and $m_0$ is defined as the previous value of $m$, so we could get:

$m, s; [\text{inv} \land m < n \land V_0 = V, \text{inv} \land 0 < V < V']$

$= m, s; [\text{inv} \land m < n \land m_0 = m, \text{inv} \land m_0 < m \leq n]$(5)

All that remains is to refine the body of the loop.
into code. Intuitively, we could see that $m = m + 1$ is a good choice. Therefore specification $\circled{6}$ could be refined as follows:

\[ m, s[x \in m \land n \land m_0 = m \land \text{inv} \land m_2 < m \leq n] \]

\[ \text{(Following Assignment Rule)} \]

\[ \subseteq m, s[x \in m \land n \land m_0 = m \land \text{inv} \land m_2 < m \leq n]\{m[m+1]\}; m := m + 1 \]

After the above refinement, we have obtained the IMP statement (IMP statement $I_4$) that changes $m$, so we remove $m$ from the frame variable. The specification is then updated as follows:

\[ s[x = \{(\sum i : 1 < i < m - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1] \land P[m] > P[m - 1],
\]

\[ s = \{(\sum i : 1 < i \leq (m + 1) - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1]\}

\[ \text{(Selection Rule)} \]

\[ \text{if } P[m] > P[m - 1] \rightarrow \ldots \]

\[ s[x = \{(\sum i : 1 < i < m - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1] \land P[m] > P[m - 1],
\]

\[ s = \text{maxincome}(m + 1, P) \]

\[ \text{fi} \]

Specification $\circled{11}$ could be updated as follows by substituting maxincome($m, P$) with its definition:

\[ s[x = \{(\sum i : 1 < i < m - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1] \land P[m] > P[m - 1],
\]

\[ s = \{(\sum i : 1 < i < m - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1]\}

\[ \text{(Scope Split Rule)} \]

\[ \sum i : 1 < i < m \land P[i - 1] < P[i] \land i \neq m; P[i] - P[i - 1] \land P[m] > P[m - 1],
\]

\[ + \{(\sum i : 1 < i < m \land P[i - 1] < P[i] \land i = m; P[i] - P[i - 1]\}

\[ s = \{(\sum i : 1 < i \leq (m + 1) - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1]\}

\[ \text{(Single Scope Rule)} \]

\[ s = \{(\sum i : 1 < i \leq (m + 1) - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1]\}

\[ + \{(\sum i : P[m + 1] < P[m] \land P[m] - P[m - 1]\}

\[ \text{It is easy to find:} \]

\[ (\sum i : P[m + 1] < P[m] \land P[m] - P[m - 1])
\]

\[ \sum i : P[m + 1] < P[m] \land P[m] - P[m - 1]
\]

\[ = \{(\sum i : P[m + 1] < P[m] \land P[m] - P[m - 1] \land P[i] - P[i - 1] \land P[m] > P[m - 1],
\]

\[ s = \{(\sum i : 1 < i \leq (m + 1) - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1]\}

\[ \text{The specification could then be transformed to:} \]

\[ s[x = \{(\sum i : 1 < i \leq (m + 1) - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1] \land P[m] > P[m - 1],
\]

\[ s = \{(\sum i : 1 < i \leq (m + 1) - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1]\}

\[ \text{We could prove that the following derivation is true.} \]

\[ s = \{(\sum i : 1 < i \leq (m + 1) - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1] \land P[m] > P[m - 1],
\]

\[ s = \{(\sum i : 1 < i \leq (m + 1) - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1] \land P[m] - P[m - 1]\}

\[ \sum i : 1 < i \leq (m + 1) - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1] \land P[m] - P[m - 1]\]

\[ + \{(\sum i : P[m + 1] < P[m] \land P[m] - P[m - 1]\}

\[ \text{Therefore, specification } \circled{10} \text{ would be refined as follows:} \]

\[ s[x = \{(\sum i : 1 < i < m - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1] \land P[m] > P[m - 1],
\]

\[ s = \text{maxincome}(m, P), s = \text{maxincome}(m + 1, P) \]

\[ \text{(Weaken Precondition Rule)} \]

\[ \subseteq s[x = \{(\sum i : 1 < i < m - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1] \land P[m] > P[m - 1],
\]

\[ s = \text{maxincome}(m, P), s = \text{maxincome}(m + 1, P) \]

\[ \text{After observing the pre- and post-conditions of specifications } \circled{10}, \text{ we analyze the relationship between maxincome}(m, P) \text{ and maxincome}(m + 1, P): \]

\[ s = \text{maxincome}(m, P) \Rightarrow s = \text{maxincome}(m + 1, P), \text{ if } P[m] > P[m - 1]
\]

\[ s = \text{maxincome}(m, P), \text{ if } P[m] < P[m - 1] \]

\[ \text{Therefore, specification } \circled{10} \text{ would be refined as follows:} \]

\[ s[x = \{(\sum i : 1 < i < m - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1] \land P[m] > P[m - 1],
\]

\[ s = \text{maxincome}(m, P), s = \text{maxincome}(m + 1, P) \]

\[ \text{(Selection Rule)} \]

\[ \text{if } P[m] > P[m - 1] \rightarrow \ldots \]

\[ s[x = \{(\sum i : 1 < i < m - 1 \land P[i - 1] < P[i]; P[i] - P[i - 1] \land P[m] > P[m - 1],
\]

\[ s = \text{maxincome}(m + 1, P) \]

\[ \text{fi} \]
Therefore the specification could be refined as follows:

\[
\begin{align*}
\{ & s = \text{maxincome}(m, P) \land P[m] > P[m-1], \\
& s = \text{maxincome}(m + 1, P) \}
\end{align*}
\]

(Assignment Rule) (18)

\[
\subseteq \{ s = s + P[m] - P[m-1]. \} \ \ \ \ ... \ \ I_1
\]

In the same way, specification (2) could be refined as:

\[
\begin{align*}
\{ & s = \text{maxincome}(m, P) \land P[m] \leq P[m-1], \\
& s = \text{maxincome}(m + 1, P) \}
\end{align*}
\]

(Skip Rule) (19)

\[
\subseteq \text{SKIP} \ \ \ \ ... \ \ I_6
\]

In conclusion, we could obtain the refinement of specification (2) (IMP statement I1) by integrating IMP statement I1, I2:

\[
\begin{align*}
\{ & s = \text{maxincome}(m, P), s = \text{maxincome}(m + 1, P) \\
& \subseteq \text{if } P[m] > P[m-1] \rightarrow s = s + P[m] - P[m-1]. \} \ \ \ \ I_7
\end{align*}
\]

(20)

2.2.4 Putting IMP statements all together

We could now glue all of the pieces (IMP statement I1, I2, I3, I4) together to get the code in Fig. 7 as a refinement of specification (1).

2.3 Formal Verification through Isabelle Based on VCG

The refinement process of Section 2.2 is accompanied by mathematical proof, ensuring manual reliability. However, it is not automatic and highly reliable. If the IMP program in Section 2.2.4 could be verified using Isabelle and VCG, the reliability of the automatic algorithm programming model could be ensured mechanically.

2.3.1 Automatically generating verification conditions based on VCG

Based on the IMP program obtained in Section 2.2.4, we insert the precondition and postcondition before and after the program respectively, and the loop invariant before the loop statement, as shown in top half of Fig. 8. The verification conditions could then be obtained (3 subgoals in bottom half of Fig. 8).

Fig. 8 Verification conditions generation based on VCG

2.3.2 Mechanically verifying the generated conditions through Isabelle

The second subgoal in Fig. 8 generates two recursions, which can be transformed into two lemmas to assist us in verifying the second subgoal. The verification of the two lemmas is depicted in Fig. 9.

Fig. 9 Verification of the two recursions

The next step is to verify the three subgoals in Fig. 8. After we call the Isabelle code "apply auto", there is only one subgoal to verify. This subgoal could be demonstrated automatically using the tool sledgehammer\cite{24,25}. Figure 10 depicts the outcome of applying sledgehammer for current verification. We can select the Isar proof code and click on it.

Figure 11 depicts the overall verification process for all the verification conditions.

2.4 Refinement of the IMP Program

Through the C++ generation system, an IMP program could be refined to a C++ program. Figure 12 depicts the refinement process for the problem of a stock’s
maximum income, with the IMP program successfully verified on the left and the C++ program generated automatically on the right. Figure 13 depicts the testing of generated C++ program.

3 Conclusion

This paper proposes an automatic algorithm programming model based on the improved Morgan’s refinement calculus to improve the automation and reliability of algorithm programming. The model generates the program specifications according to user requirements. It refines step by step with formal verification and generates executable programs automatically with...
the help of the C++ generation system. An example of a stock’s maximum income demonstrates the model’s effectiveness. The model in this paper has the following characteristics in general.

1) Morgan’s refinement calculus is extended by adding of quantifier specification transformation rules. Therefore, the formalization of program refinement is enhanced.

2) The manual generation of verification conditions for algorithm programs is replaced by the automatic generation. Therefore, formal verification automation is improved.

3) Based on the C++ generation system, abstract IMP programs are refined to executable programs. Therefore, the program refinement is more complete.

The automatic algorithm programming model allows for efficient programming while also ensuring the correctness of the generated program. However, two aspects of future work must be improved. One is to expand the refinement rule library to further optimize Morgan’s refinement calculus, and the other is to apply the model to more complex data structures.

References


