The Exception Handling in Automatic Translation of Goto from PROMELA Model to *Java* Program

Suprapto, *Member, IAENG*

***Abstract*—A structured programming language does not or should not have an unconditional jump goto statement. Any program that contains a goto is not structured. There had been a serious discussion over the years whether or not goto statements should be eliminated or kept, and there was one of the most discussed papers that spoke for the elimination of goto statements. However, there were still some that spoke for keeping the goto statements, even only for certain cases. The modeling language PROMELA is one of some languages that contain goto – the language creator agreed to keep goto.**

This study proposes an automatic translation of goto that appears in a PROMELA model into *Java* program using exception handling in *Java*. The automatic translation is part of the translator from PROMELA model into *Java* program done in the previous study. There are at least two reasons why used exception handling: (1) it is able to bring program's execution path jumping from one part of program to another. This jump can be addressed to the specified target by exploiting type of exception being captured. (2) it can be assigned with a value - usually as a message about the exception. So that, by assigning different value will distinguish one goto from others in case there were more than one gotos in the same block of codes. At the end, a program was written to show the automatic translation works properly. However, because of the behavior of goto itself and limitations of exception handling in *Java*, the automatic translation developed in this study has not handled all types of goto patterns appeared in PROMELA model.

The result of this study were the automatic translation correspond to three types of goto pattern in a PROMELA model, and a procedure to determine the right type of goto pattern correspond to the automatic translation. The testing was conducted by running the program of automatic translation written in *Java* using 30 cases (PROMELA model containing the goto pattern), 10 for each. In accordance with the testing, the automatic translation worked properly for all cases.

***Index Terms*—Goto, label, exception handling, PROMELA, *Java*, appearance, translation.**

I. INTRODUCTION

THE unconditional jump statement (goto) is the most powerful control statement, it means that all control structures can be built by using only goto statement and one-way selector. The use of goto means that any statement in a program can potentially be executed immediately after any other statements, and frequently jeopardize both readability and maintainability [11]. However, the benefits of using it sometimes outweigh the dangers, almost all "standard" programming languages include it [20] [21].

Even though PROMELA is one of languages that include goto, however, goto is not really statement [7]. Goto is normally not executed, it is only used by the parser to determine the target control state for the immediately preceding statement. The target state is identified by the label $name$ and must be unique within the surrounding proctype declaration or never claim [22] [4].

Java programming language does not have an explicit goto statements. Keywords such as break, continue, exit, and return are restricted forms of goto. The break is a keyword used to "break" out of the innermost switch, for, while, or do while statement body. It can also be used in a label form to jump out of outer nested loop or switch statement. Similarly, the keyword continue can be used only inside loops (i.e., for, while, and do while). It causes the execution to skip over the remaining steps of the loop body in the current iteration (but then continue the loop if its condition is satisfied). All of these keywords attempt to minimize harm of readability than goto by restricting the potential target of the branching. However, *Java* has an exception handling that can be used to implement goto statements [23] [16] [2] [19] [12].

A project proposed to add some new jump statements to C++ made available an explicit-specifier for switch statement. The new jump statements were break $label$, continue $label$ (same as *Java*), goto case $constant - expression$ and goto default (same as C#). An explicit switch statement (same as C#) caused each case block to have its own block scope, and to never flow off the end. That is, each case block must be explicitly exited. The implicit fall through semantic between two consecutive case blocks could be expressed in an explicit switch using a goto case statement instead [3].

The handling of FORTRAN goto statements has been a difficult problem because of *Java*'s lack of goto statement [15]. According to [18], *Java*'s labeled break, and continue statements could be used to translate certain types of FORTRAN gotos. The technique has been using for these gotos was to generate "*placeholders*" in the *Java* source code. The *placeholders* were method calls which specified the target of the goto statements.

In [7] was discussed a strategy of how the goto statements should be translated into *Java* code(s) in case they occurred in a model of PROMELA. The strategy was encapsulating a switch statements inside a while statement. There were four cases of goto statement's appearances observed and solved by this approach. The first case was goto statement appears in one option of selection if..fi and label comes before goto, the second was similar to the first case except label was in atomic construct, the third was one in which both label and goto appear in the same do..od construct and label comes after goto, and the last one was like the third except label appears in the outside of repetition do..od.

The association between some constructs in PROMELA and some in *Java* had been defined, and its correctness had already been proven, except for goto statement [24]. In PROMELA, the goto construct could be any where in a model. Therefore, the translation of goto that appears in

Manuscript received August 31, 2020; revised July 10, 2021.

Suprapto is a lecturer and researcher of Computer Science and Electronics Department, Universitas Gadjah Mada, Yogyakarta, Indonesia, 55281. e-mail: sprapto@ugm.ac.id.

a model PROMELA to a *Java* program require a special handling and treatment.

Most mainstream or modern programming languages, such as Java and C#, typically provide features that handle exceptions. They typically involve constructs to throw and handle error signals, and separate error-handling code from regular source code and aim to assist in the practice of software comprehension and maintenance [8] [13]. The policy of exception handling of a system consists of design rules set that specify its exception handling behavior (i.e., how exceptions should be handled and thrown in a system). Such policy is usually undocumented and implicitly defined by the system architect [26] [12]. Previous studies on exception handling stated the suboptimal practices of the flows and prevalence of their anti-patterns. In addition, some studies do not agree that exception handling can make the programming languages simple and promote the introduction of subtle bugs in programs [8] [17]. Although the majority of exception handling anti-patterns are not significant in the models, but they can provide a significantly explanation about power to the probability of post-release defects. This paper presents the study of exception handling usage in *Java* to reduce the complexity of flows in program as a result of translation from PROMELA model that contain goto. The result of this study is an automatic translation of goto in PROMELA model into *Java* program.

II. Discussion

There are some difficulties found in translating goto that occurres in a model of PROMELA into a *Java* program, because *Java* does not support goto's mechanism directly. Fortunately, *Java* has an exception handling mechanism's that can be used to translate goto occurring in a PROMELA model. Even so, there were still some occurrences of goto in a model impossible or could not even be translated into *Java* program. This was because the exception's behavior that only bring an execution path out of a code block, while goto is able to jump either into a code block or out of it. For more detail, the following are some factors that determine the translatability of goto's construct.

A. Position level of goto and label

The difference of position level (or depth) between goto and label may cause goto in a PROMELA model could not be translated into a *Java* program using exception handling. In accordance with the exception's behavior, if the jump occurs from a position level into another of code block with one or more levels deeper, goto could not be translated automatically. Otherwise, the translation can be carried out. An example of jump in the same level is shown in Listing 1.

Listing 1: An example of jump in the same level

```
...
swap:
  a = a + b;
  b = a - b;
  a = a - b;
  log ! a, b;
goto swap;
...
```

It is seen from Listing 1, the position of label swap has the same level as goto, in other word they are parallel. While an example in Listing 2 shows the case where an occurrance of goto and label are in different level. Because the label swap appears inside atomic, and goto is outside atomic.

Listing 2: An example of jump into the inner level, atomic code block

```
...
atomic {
  flag = false;
swap:
  a = a + b;
  b = a - b;
  a = a - b;
}
log ! a, b;
goto swap;
...
```

The label swap appears in the position one level deeper than the goto does. In this example the atomic construct consists of four statements, and block of label starts from the second statement: a = a + b;. Since the label is inside the atomic construct, and the goto is parallel with it, so that the direction of execution is from outside to inside. Because of the limitations, exception handling in *Java* can't simulate it. Therefore, this appearance pattern of goto can't be translated.

B. Occurrence Order of goto and label

A relative position of goto to label also influences the pattern or structure of goto's translation result. If goto comes before label, there would be a block of codes (i.e., ones in between goto and label) skipped or ignored. Otherwise, if goto comes after label, it would be repeated. An example of the pattern in which label come before goto is shown in Listing 1. While Listing 3 shows an example of one in which label come after goto.

Listing 3: A case where label comes after goto

```
...
goto next;
  a = a + b;
  b = a - b;
  a = a - b;
  log ! a, b;
next:
...
```

The example in Listing 3 shows the case where label next come after goto, and both goto and label have the same level of position.

C. The appearance of goto or label in some constructs

In PROMELA, both goto and label might appear any where in a model, either as independent appearance or appears in other constructs. If goto or label arises in a construct with special behavior such as, if..fi, do..od, atomic, and unless, the translation of goto must consider the execution flow of corresponding construct. This is done in order to maintain the behavior of construct, so that, their semantics are preserved. For example, if a jump goes from inside an atomic construct to outside, the lock was applied to variable and process must have released first to allow other processes become active again. Listing 4 shows an example of the pattern where a jump leave from inside an atomic code block.

Listing 4: A case where the execution's flow jumped out of the inner level, atomic code block

```
....
atomic{
  log ! a, b;
  if
    :: flag -> goto next;
    :: !flag -> skip;
  fi;
a = a + b;
b = a - b;
a = a - b;
}
next:
log ! a, b;
....
```

In this example, goto is inside if..fi construct, while the if..fi is inside atomic construct. Since label next is outside the atomic, it means that jump goes to not only outside of if..fi but also the construct containing it. In other word, the jump goes to outer level of code block.

D. *Number of goto respect to label*

It is obvious that the number of goto's address to label will influence the translation. In practice, one label could be addressed by more than one goto's. However, this form (or structure) will cause some difficulties in translation, and even impossible in the automatic one. For this reason, the study only cover the case where one label is associated with at most one goto. An example is shown in Listing 5.

Listing 5: One label one goto

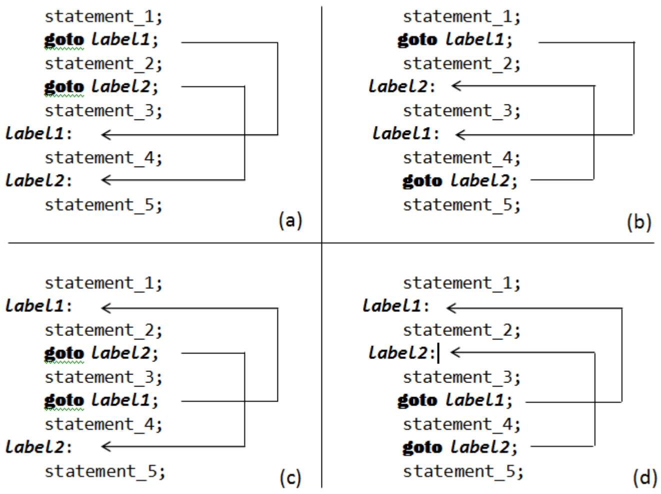
```
...
atomic{
log ! a, b;
if
  :: flag -> goto swap;
  :: !flag -> goto next;
fi;
swap:
  a = a + b;
  b = a - b;
  a = a - b;
}
next:
log ! a, b;
...
```

In Listing 5, there are two gotos, both are inside if..fi construct, and this if..fi is inside atomic. The associate label, however, has different position level. Label swap inside atomic has the same level as the associate goto, while next is outside atomic.

On the other hand, Listing 6 shows an example of the form in which one label is addressed by two goto. The first is inside if..fi that is in atomic, and the second is outside the atomic. Because the label swap is inside the atomic, the flow of the first goto goes out, while the flow of the second goto goes in, and this is not possible to be performed automatically. Therefore, it will not be covered in this automatic translation.

Listing 6: One label two goto

```
...
atomic{
  log ! a, b;
  if
    :: flag -> goto swap;
    :: !flag -> skip;
  fi;
  swap:
  a = a + b;
  b = a - b;
  a = a - b;
 }
goto swap;
log ! a, b;
...
```



```
   statement_1;
   goto Label1;
   statement_2;
   goto Label2;
   statement_3;
Label1:
   statement_4;
Label2:
   statement_5;
                    (a)
```

```
   statement_1;
   goto Label1;
   statement_2;
Label2:
   statement_3;
Label1:
   statement_4;
   goto Label2;
   statement_5;
                    (b)
```

```
   statement_1;
Label1:
   statement_2;
   goto Label2;
   statement_3;
   goto Label1;
   statement_4;
Label2:
   statement_5;
                    (c)
```

```
   statement_1;
Label1:
   statement_2;
Label2:
   statement_3;
   goto Label1;
   statement_4;
   goto Label2;
   statement_5;
                    (d)
```

Fig. 1: The four classes of untranslatable goto patterns.

E. *Structure of goto and label pair*

Any number of goto and label pairs can appear anywhere in any model of PROMELA. However, the structure of their appearances also influence the possibility of translation. Only a certain number of structures would be automatically translated. The parallel appearances of pairs of goto and label are structures that usually easy to translate. On the other hand, two or more pairs that cross each other are ones that impossible or could not be automatically translated using exception handling. The pattern is then called by **untranslatable**. It is indicated by the occurrences of more than one pairs of either goto and label or label and goto in the same model, and their flows cross each other or their scopes intersect each other. By the scope means a block of codes between label and goto or vice versa. Based on the order of goto or label occurrences, the structures could be classified into four: (a) both goto come before the label with the same order of associate goto, (b) in the first pair, goto come before the label and in the second pair label come before goto, but the label of the first pair come after the label of the second pair, (c) in the first pair, label come before the goto and in the second pair goto come before label, but the goto of the first pair come after the goto of the second pair, and (d) both gotos come after labels with the same order of associate label. Fig. 1 (a), (b), (c) and (d) shows the four classes of structures that cannot be automatically translated - then they are refered as **untranslatable**.

As an illustration for Fig. 1, Listing 7 shows the case where flows of two pairs of goto and label cross each other.

Listing 7: One label one goto cross each other

```
  ...
  atomic{
     log ! a, b;
     if
       :: flag -> goto swap;
       :: !flag -> skip;
     fi;
     next:
     a = a + b;
     b = a - b;
     a = a - b;
  }
  swap:
  log ! a, b;
  goto next;
  ...
```

It is seen from Listing 7 the pattern has two kinds of codes block, first is one that will be skipped; i.e., it consists of $a = a + b$; $b = a - b$; and $a = a - b$, and second is one that will be repeated; i.e., it consists of $a = a + b$; $b = a - b$; $a = a - b$; and $log\ !\ a, b$;. Moreover, these two blocks are overlapped. So that, in accordance with the scheme in Fig. 1 this pattern of goto is untranslatable.

F. Automatic Translation of goto

The automatic translation of goto that appears in a PROMELA model into a *Java* program using exception handling requires a derived class exception with type of GotoException. An object created from class GotoException is then thrown when find goto. It could be carried out also using the existing class of exception, but it must be distinguishable from exception created or required by program. To make sure that goto is captured at the appropriate place, the stored message in exception at goto should be the name of label addressed by goto. Because when there are some pairs of gotos, the place to capture the first goto might not be right. If it does, exception would be thrown again in order to be captured at the wrong place. Hence, the exception handling in *Java* can't cover all cases of goto occurrence in PROMELA model.

There are only three patterns of goto that is possible to be translated automatically using exception handling. They are then refered as Type1, Type2 and Type3 respectively.

1. Label comes before Goto

In Type1 pattern, if goto is unconditional jump, the block of codes (i.e., all statements are in between goto and label) would be repeatedly executed. On the other hand, they will be executed if the condition of goto is satisfied. Moreover, when position level of label and goto are equal, the translation is straight forward.

The translation will still work, even their level of positions are different, as long as the position level of goto is deeper than the one of label. For instance, goto appear inside the block code of *statement_k* - it means the position level of goto is deeper than the one of label, an automatic translation can be performed. The illustration of this situation is shown in Fig. 2.

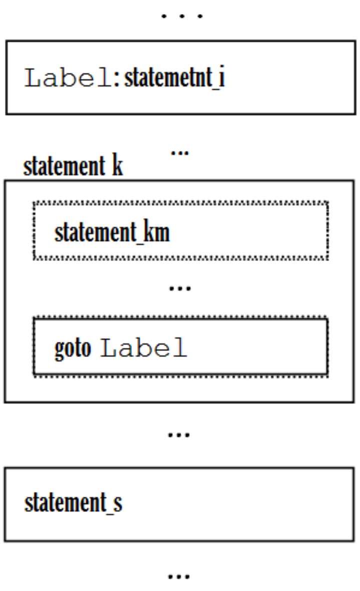
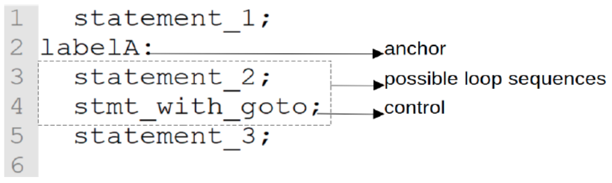


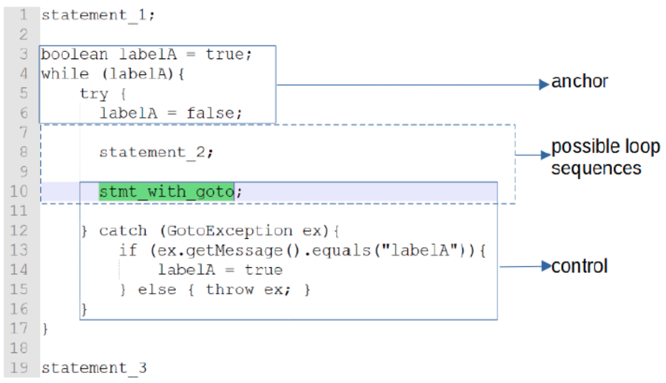
Fig. 2: The translatable structure of different position level between label and goto

In real case, when goto is in any statement such as, an option of if..fi or do..od, atomic, and unless constructs, the schemes will be formulated as in Fig. 3.



```
1    statement_1;
2  labelA:
3    statement_2;
4    stmt_with_goto;
5    statement_3;
6
```

Fig. 3: label comes before goto, and goto appears in any other statements



```
statement_1;

boolean labelA = true;
while (labelA){
    try {
      labelA = false;

      statement_2;

      stmt_with_goto;

    } catch (GotoException ex){
       if (ex.getMessage().equals("labelA")){
          labelA = true
       } else { throw ex; }
    }
}

statement_3
```

Fig. 4: Translation of structure where label come before goto and the level of label is outer than goto

In this structure of occurrance, goto causes a loop. labelA is called anchor since it becomes starting point in which the loop happens. *stmt_with_goto* can be any statement with goto, such as goto in *if..fi* or *if..fi* by itself, and it is called control since it can change path of program's execution when gpto is executed. In Fig. 3, it change the execution path that is supposed to line 5 from line 4, but to line 2 instead, and creates the loop. Additionally, it also has function as scope delimiter of loop. So that, the next statement (*statement_3* in line 5) is not part of loop sequence. *possible_loop_sequences* (i.e., the statements insides dottet-line box) is a group of statements that will be repeated whenever goto is called. As shown in Fig. 3, it begin at statement after label until control. Therefore, it will be automatically translated in Java program as seen in Fig. 4.

It is straight forward that lines 1, 8 and 19 are translation of line 1, 3 and 5 in Fig. 3 respectively. An anchor labelA (line 2 in Fig. 4) is translated into these parts:

- a boolean variable labelA, since Java does not have label. So that, a variable is used to substitute as well as an identifier of label or anchor.
- while(labelA), as it is explained earlier, an anchor functions as indicator of the beginning of loop, and it is translated into loop statement.
- labelA = $false$, used to indicate that the label has been passed. This assignment also used to make sure that loop will not happen without goto's calling. try does not have any function, it only to meet the syntax of $try..catch$.

Statement *stmt_with_goto* is not translated, since it could be sequence such as, goto in $if..fi$. On the other hand, if *stmt_with_goto* is a single statement, it would be translated as in Listing 8. As mentioned previously, *stmt_with_goto* had two functions: moves execution path (control) and limits scope of looping. As a control, it was translated into throw new GotoException("labelA"). So that, if there is a goto inside *stmt_with_goto*, throw would produce exception that will change execution path as control (or goto) functioning. The execution will move to section of catch and change the value of $labelA$ into $true$ - $labelA = true$. The existence of variable $labelA$ in catch is used also as identifier of the label, so that, goto will not be captured by wrong catch in case there are more than one label in the model. Additionally, because catch is an end part of loop, it will guarantee executing the loop. catch(...) is a part of translation of *stmt_with_goto* that functions to limit scope of looping. If there is no goto inside *stmt_with_goto*, catch would not be executed, and $labelA = true$ was not executed. Hence, it become end of loop or loop's sequence.

Listing 8: Translation template of goto pattern where label come before goto and the level of label is outer than the level of goto

```
...
statement_1;

boolean labelA;
while (labelA)
  try{
   labelA = false;
   statement_2;
   throw new GotoException("labelA");
  } catch (GotoException ex){
      if (ex.getMessage().equals("labelA"))
        { labelA = true; }
      else { throw ex }
    }
  statement_3
  ...
```

The translation result in Listing 8 only satisfy for goto pattern in which the position level of label must be either the same as or at least one level outer than the position level of goto, otherwise there is no automatic translation. The goto pattern in which the position level of label was at least one level inner than the position level of goto is shown in Fig. 5.

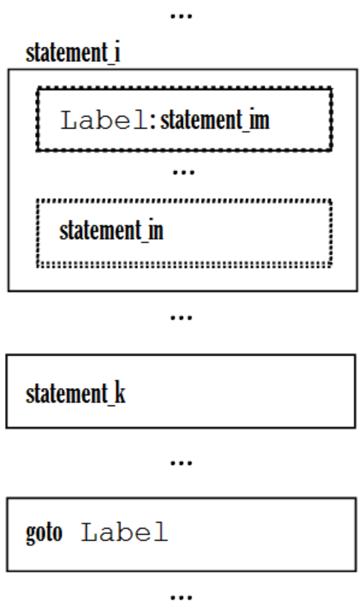


Fig. 5: An untranslatable goto pattern where position level of label was one level inner than its goto

In addition, if goto is inside the constructs whose special behavior, such as, atomic, unless, etc., translation requires a special handling to avoid behavior changes of the corresponding construct. For those of the pattern in which goto is inside atomic and label is outside, the lock must be released from atomic right before throwing an exception to allow other processes back to run. The substitute of goto labelA; is shown in Listing 9.

Listing 9: An additional releaseLock()

```
...
statement_1;

boolean labelA;
while (labelA)
try{
     labelA = false;
     statement_2;

     releaseLock();
      // allowing other processes bact to run

     throw new GotoException("labelA");
   } catch (GotoException ex){
       if (ex.getMessage().equals("labelA"))
         { labelA = true; }
       else { throw ex }
     }
   statement_3
   ...
```

In order to make the concept clearer, the following are some examples that illustrate the usage of translation pattern shown in Listing 8.

Example II.1.

This example shows the codes fragment in a PROMELA model where both label **labelOne** and goto are in the same level, as in Listing 10.

Listing 10: The code fragment in PROMELA containing a pair of label-goto

```
  ...
  bool b;
labelOne:
  int a = 4;
  a += 5;
goto labelOne;
  a = 3;
  b = true;
  ...
```

In accordance with the translation pattern in Listing 8, the result (i.e., a *Java* program) is shown in Listing 11.

Listing 11: *Java* translation of the pattern where label come before goto

```
.....
PromelaBool b;
boolean labelOne = true;
while (labelOne) {
  try {
    labelOne = false;
    PromelaInt a = new PromelaInt(4);
    a.assign(a.getValue()+5);
    throw new GotoException("labelOne");
  } catch ( GotoException x ) {
      if (x.getMessage().equals("labelOne"))
        labelOne = true;
      else throw x; }
  }
a.assign(3);
b.assign(1);
.....
```

that Since this automatic translation is part of translator from PROMELA model into *Java* program, any variable used in PROMELA models is translated into objects in *Java* [25]. For instance, *bool b* in PROMELA model is translated into $PromelaBool\ b$ istead of using *boolean b*, etc.

Notice that, not only the translation of goto, but also for statement. For instance, $PromelaBool\ b$; is a translation result for *bool b*; in PROMELA model. Instead of using boolean in *Java*, a defined data type PromelaBool used as a translation of bool in PROMELA, since the values between bool in PROMELA and boolean in *Java* is not exactly the same. This also applies to some other data types, such as int, bit, unsigned, etc. [25]. However, the statement such as $boolean\ labelOne = true$; is a new boolean variable declared and initialized in *Java*, that is why does not use $PromelaBool$.

Example II.2.

It is different than one in Example II.1, the occurrence levels of goto and label are different (i.e., goto is nested in if..fi construct, see Listing 12). The position level of label is outer than one of goto, this pattern is translatable. For the shake of simplicity, however, the if..fi construct used in this example has only one *sequence* which is $a < 100 \to$ goto **labelOne;**. Besides, the goto has a condition, then goto would be executed if $a < 100$ is satisfied.

Listing 12: *Java* translation of pattern where label came before goto

```
  ....
  bool b;
labelOne:
  int a = 4;
  a += 5;
  if
   :: a < 100 -> goto labelOne;
  fi;
  a = 3;
  b = true;
  ....
```

In accordance with the translation pattern in Listing 8, the result of translation is shown in Listing **??**.

Listing 13: *Java* translation result where label is outer than goto

```
.....
PromelaBool b;
boolean labelOne = true;
while (labelOne) {
  try {
    labelOne = false;
    PromelaInt a = new PromelaInt(4);
    a.assign(a.getValue()+5);
    boolean if_flag = false;
    while (if_flag) {
      PromelaInt choice = -1;
      header.lock();
      if ((a.getValue() < 100) == true)
    {
  choice = 1;
}
switch( choice ) {
  case 1:
    if_flag = true;
    throw new GotoException("labelOne");
    if (x.getMessage().equals("labelOne"))
      labelOne=true;
    break;
  }
 } catch (GotoException x) {
    labelOne = true;
   }
}
a.assign(3);
b.assign(1);
.....
```

2. Label comes after Goto

This section discusses the **Type2** pattern of goto which is one where label come after goto. With this order of occurrence, there is a block of codes skipped. Similar to the previous case, the simplest structure of this case is when the position level of label and goto are the same. The scheme for this pattern is shown in Fig. 1 (b).

On the other hand, if the position level of goto is one or more levels inner than its label, it would be translatable. The illustration of this situation is shown in Fig. 6.

The general pattern of scheme in Fig. 6 is shown in Listing 14, where each $statement\_1$, $statement\_2$, and $statement\_3$ can be a single statement or a block of statements.

Listing 14: Goto pattern where label comes after goto

```
  ...
  statement_1;
```

```
goto labelA;
  statement_2;
labelA:
  statement_3;
  ...
```

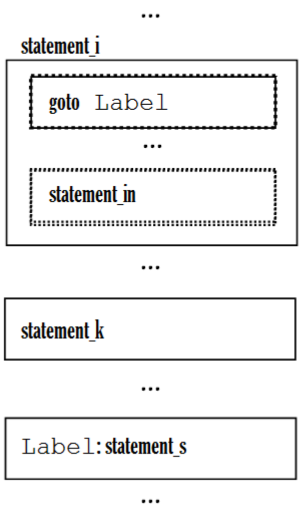


Fig. 6: Goto pattern in which label come after goto

This pattern is even simpler than the previous one, since the execution flow is only one-way down. It is like an ordinary sequential statements. Therefore, the pattern in Listing 14 is easily translated into *Java* program by using exception handling. The result of translation is shown in Listing 15.

Listing 15: Translation template of goto pattern in which label comes after goto and the level of label is outer than the level of goto

```
  ...
  statement_1;
  try{
    throw new GotoException("labelA");
    statement_2;
  } catch(GotoException ex){
      if (ex.getMessage().equals("labelA")
        { } else {throw ex}
    }
  statement_3;
  ...
```

Similar to the previous form, the translation template in Listing 15 only satisfies to the goto pattern where the position level of label is one outer than that of goto. Hence, the pattern whose scheme is shown in Fig. 7 does not work.

When goto is inside the special behavior construct such as atomic, the translation template is similar with one of the previous patterns. The only different is the lock must be released from atomic before throwing an exception to allow other processes back to run. Therefore, the translation template will look like one in Listing 16.

Listing 16: Translation template of goto pattern in which label comes after goto, and goto is inside atomic while label is outside

```
  ...
  statement_1;
  try{
```

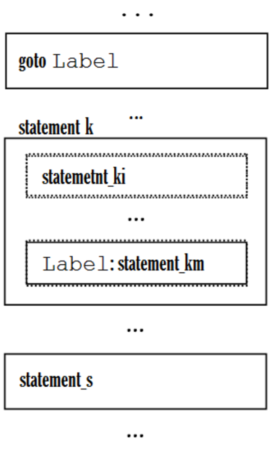


Fig. 7: Goto pattern where the position level of label is deeper than one of goto

```
    releaseLock();
    throw new GotoException("labelA");
    statement_2;
  } catch(GotoException ex){
      if (ex.getMessage().equals("labelA")
        { }
      else {throw ex}
    }
  statement_3;
  ...
```

The following are some examples of the goto pattern in which label comes after goto, the level of label is outer than goto or goto is inside atomic.

Example II.3.

In this example goto is inside atomic construct, while label is outside. So that, it is translatable. The codes fragment is shown in Listing 17.

Listing 17: An example of goto pattern where the level of label is inner than the level of goto

```
  ....
  int a;
  atomic {
     b += 4;
     a += b;
   goto labelThree;
     b = 5;
  }
labelThree:
  a = 25;
  ....
```

In accordance with the translation template in Listing 15 and Listing 9, the translation result is shown in Listing 18.

Listing 18: Translation result of goto pattern where label comes after goto and the level of label is outer than the level of goto

```
  ...
  PromelaInt a = new PromelaInt();
                                // int a;
  try {
   int stmt_number;
   while (stmt_number <= 4) { // atomic
     header.lock();
     switch (stmt_number) {
      case 1 :
        if (!true) break;
     b.assign(b.getValue()+4);   // b+=4;
        stmt_number++;
       case 2 :
        if (!true) break;
    a.assign(a.getValue()+b.getValue());
                                  // a+=b;
        stmt_number++;
       case 3 :
        if (!true) break;
        releaseLock();
    throw new GotoException("labelThree");
                         // goto labelThree;
        stmt_number++;
       case 4 :
        if (!true) break;
        b.assign(5);   // b = 5;
        stmt_number++;
      }
    }
  } catch (GotoException x) {
                           // labelThree:
   if (x.getMessage().equals
                           ("labelThree")) {}
      else throw x;
  }
  a.assign(25);   // a = 25;
  .....
```

Notice that the translation tempalte in Listing 16 does not apply for every pattern where goto is inside a construct with special behavior. For example, it does not apply when pairs of goto-label is inside the unless construct. Especially when goto is in the main sequence of unless and label is in the escape sequence.

3. goto in unless main sequence and label in unless escape sequence

This section presents the Type3 of goto pattern. Suppose there are m statements in the main sequence, and there are n in the escape sequence. The goto appears in between $main\_stmt\_(i-1)$ and $main\_stmt\_i$ for some i, $2 \leq j \leq m$, while label does in between $escape\_stmt\_(j-1)$ and $escape\_stmt\_j$ for some j, $2 \leq j \leq n$. The schema of appearance is depicted in Listing 19.

Listing 19: The pattern in which goto is inside unless construct

```
  .....
  { main_stmt_1;
      ...
      main_stmt_(i-1);
  goto labelx;
      main_stmt_i;
      ...
      main_stmt_m;
  } unless { escape_stmt_1;
      ...
      escape_stmt_(j-1);
  labelx:
      escape_stmt_j;
      ...
      escape_stmt_n;
  }
  ...
```

Based on the control flow of unless construct, goto will be executed only if while executing the first $i-1$ statements in the main sequence, the first statement in the escape sequence is not executable. Once goto is executed, all statements in the main block after goto - $main\_stmt\_i, \ldots, main\_stmt\_m$, and all statements in the escape block before the label - $escape\_stmt\_1, \ldots, escape\_stmt\_(j-1)$ will be skipped. In *Java*, this pattern has translation template as in Listing 20. The $tologic$ is a function used to convert the statement into logic value ($true$, $false$).

Listing 20: The translation template of the pattern in which goto is inside unless construct

```
  ....
  try {
   if (tologic(escape_stmt_1))
     throw new unlessException();
     main_stmt_1;
     ...
   if (tologic(escape_stmt_1))
     throw new unlessException();
     main_stmt_{i-1};
   if (tologic(escape_stmt_1))
     throw new unlessException();
     throw new GotoException("labelx");
   if (tologic(escape_stmt_1))
     throw new unlessException();
     main_stmt_i;
     ....
   if (tologic(escape_stmt_1))
     throw new unlessException();
     main_stmt_m;
  } catch(unlessException x) {
      escape_stmt_1;
      ....
      escape_stmt_(j-1);
      escape_stmt_j;
      ....
      escape_stmt_n;
  }
  catch (GotoException g) {
    escape_stmt_j;
    ....
    escape_stmt_n;
   }
   ....
```

Example II.4 gives an illustration of simple case in Listing 19.

Example II.4.

As in example II.3, this goto also comes before label. However, it appears in unless construct. The codes fragment is shown in Listing 21.

Listing 21: The translation template of the pattern in which goto is inside unless construct

```
   ....
   {
    a = 4; b += 3;
  goto labelX;
   } unless {a == 12;
    labelX:
       b = 7; } a = 30;
  ....
```

In accordance with the translation template in Listing 20, the number of statements in main sequence is two and goto come in the main sequence after the last statement. The label come in the escape sequence ater the first statement. Then the translation result of PROMELA codes fragment in Listing 21 is shown in Listing 22 (i.e., *Java* program).

Listing 22: The translation result of the pattern in which goto is inside unless construct

```
    .....
    try {
     if (a.getValue() == 12)
        throw new unlessException();
        a.assign(4);
     if (a.getValue() == 12)
        throw new unlessException();
        b.assign(b.getValue() + 3);
     if (a.getValue() == 12)
        throw new unlessException();
        throw new GotoException("labelx");
    } catch (unlessException u) {
        while ( (a.getValue() == 12)
                               == false { }
        b.assign(7);  // b = 7;
      } catch(GotoException g) {
     if (g.getMessage.Equals("labelX")){
              a.ssign(5); }
          else throw g
    a.assign(30);
    .....
```

Subsequently, in order to obtain the accurate result of translation, it is required a procedure to select (or determine) template correctly. The following subsection presents a procedure that will guide in selecting the right type of goto statement.

G. Procedure of Type Selection

This procedure is part of DFS process in the translation of PROMELA model [25]. It was specifically designed to select the right type of translation. It guides to select the right type of goto statement. The selection is made based on the occurrance pattern of goto statement in PROMELA model. The procedure is shown in Listing 23.

Listing 23: Procedure for determining type of goto statement

```
Procedure Type Selection

1         Read statement
2         If statement contain goto
2.1       Check the entire statement afterward
2.2       If find label
2.2.1         Type2
3         Else if statement is label
3.1       Check the entire statement afterward
3.2       If find statement containing goto
3.2.1         Type1
4         Else if statement is unless
4.1       Check the entire action
4.2       If contain goto
4.2.1     Check guard
4.2.2     If find label
4.2.2.1       Type3
```

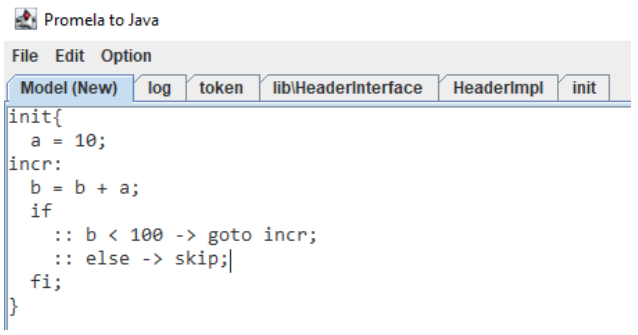
A statement contains goto if and only if either in its action or guard there is a statement with type of goto. While Type1 is the form of goto in which goto statement come before label, Type2 is the form of goto in which label come before goto statement, and Type3 is the form of goto in which goto statement is in action of an unless, and label is in the guard of the same unless. Among these three, they might have nested each other, such as Type1 is inside Type2 or vice versa, Type1 and Type2 are parallely inside Type3, Type1 is inside Type2 and Type2 is inside Type3, etc.

Since the procedure is part of DFS process, the translation is carried out from the innermost pairs of goto and label until the outermost in the nested form. For instance, the translation will begin with Type1 and its result will be used in translation process of Type2 for the case of Type1 is inside Type2. Similarly, the case of Type1 and Type2 are parallely inside Type3 will be carried out by first translating Type1 and Type2 sequentially, and the results will be used together in translation process of Type3, etc.

H. Results of Testing

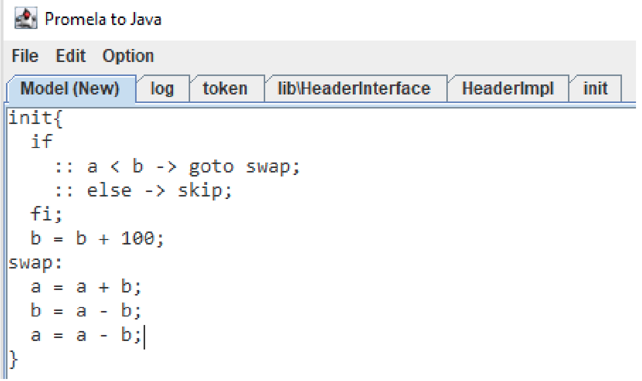
To show that the automatic translation of goto's that appear in PROMELA model into *Java* program work properly for each type, this section presents the results of running program with some various inputs for each type. The code fragment of PROMELA models that contain goto were manually generated, and they would be used as inputs of the automatic translator. The testing was conducted with 10 models for each type and run one per each, so that, there were 30 results. However, to minimize the use of space (number of pages), only one selected results were displayed for each type.

1) Type1: The input sample used in this case representing Type1 of goto pattern, which is label come before goto. As shown in Fig. 8, the label incr lies outside if..fi construct, while the goto lies inside. The label incr come before the statement goto. So that, when the guard $b < 100$ is executable, the control is sent back to the label incr and the statement $b = b + a$ is executed, otherwise the control is sent to the statement after if..fi construct. This flow of process will be repeated until $b >= 100$. The output of running program for this input (Fig. 8) is shown in Fig. 10.



```
init{
  a = 10;
incr:
  b = b + a;
  if
    :: b < 100 -> goto incr;
    :: else -> skip;
  fi;
}
```

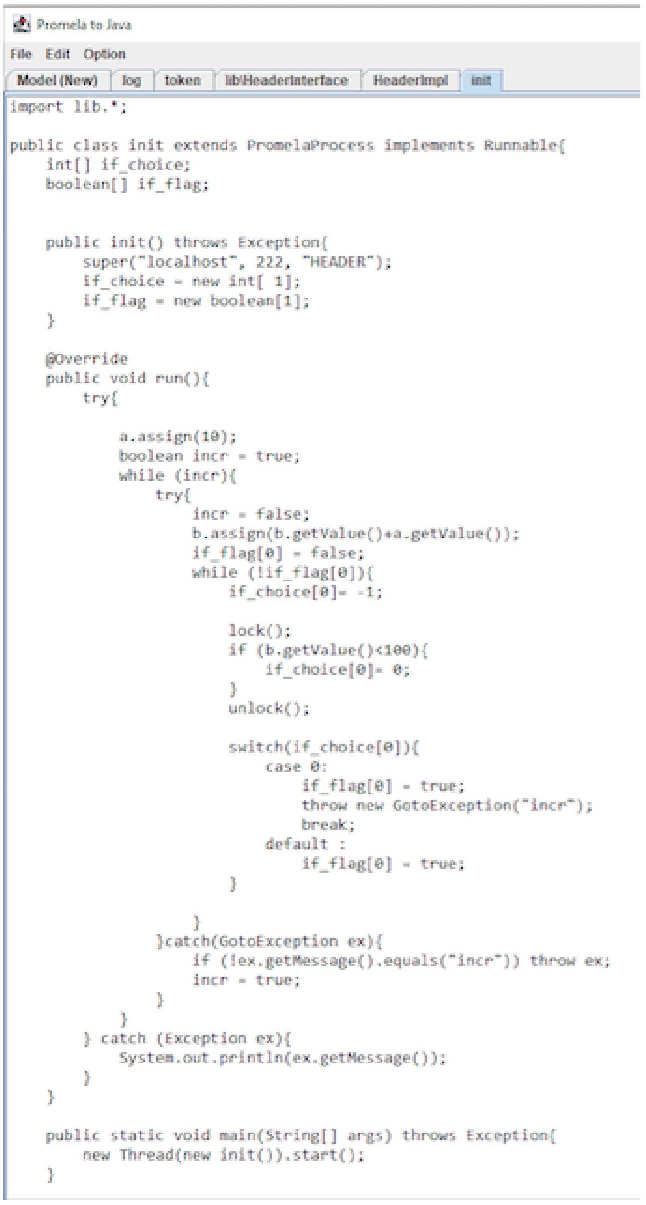
Fig. 8: The selected input (model) representing Type1



```
init{
  if
    :: a < b -> goto swap;
    :: else -> skip;
  fi;
  b = b + 100;
swap:
  a = a + b;
  b = a - b;
  a = a - b;
}
```

Fig. 9: The selected input (model) representing Type2

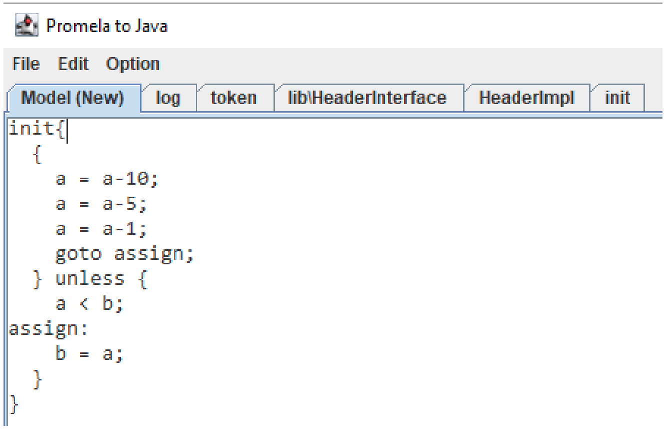
2) Type2: As it was mentioned previously, in Type2 statement of goto comes before label. Therefore, all statements are in between will be skipped. For instance, in Fig. 9, the statement of goto lies in an action with the guard $a < b$ - one of the options in if..fi construct, and the corresponding label (i.e., swap) lies outside of if..fi construct and comes after it. When the guard $a < b$ is executable (not blocked), the statement of goto will be executed, and the control program is delivered to location of swap, and the only statement $b = b + 100$ is skipped. Otherwise, the control is delivered to statement $b = b + 100$, and the execution is performed normally. The result of translation for this example is shown in Fig. 12.



```
import lib.*;

public class init extends PromelaProcess implements Runnable{
    int[] if_choice;
    boolean[] if_flag;


    public init() throws Exception{
        super("localhost", 222, "HEADER");
        if_choice = new int[ 1];
        if_flag = new boolean[1];
    }

    @Override
    public void run(){
        try{

            a.assign(10);
            boolean incr = true;
            while (incr){
                try{
                    incr = false;
                    b.assign(b.getValue()+a.getValue());
                    if_flag[0] = false;
                    while (!if_flag[0]){
                        if_choice[0]= -1;

                        lock();
                        if (b.getValue()<100){
                            if_choice[0]= 0;
                        }
                        unlock();

                        switch(if_choice[0]){
                            case 0:
                                if_flag[0] = true;
                                throw new GotoException("incr");
                                break;
                            default :
                                if_flag[0] = true;
                        }

                    }
                }catch(GotoException ex){
                    if (!ex.getMessage().equals("incr")) throw ex;
                    incr = true;
                }
            }
        } catch (Exception ex){
            System.out.println(ex.getMessage());
        }
    }

    public static void main(String[] args) throws Exception{
        new Thread(new init()).start();
    }
```

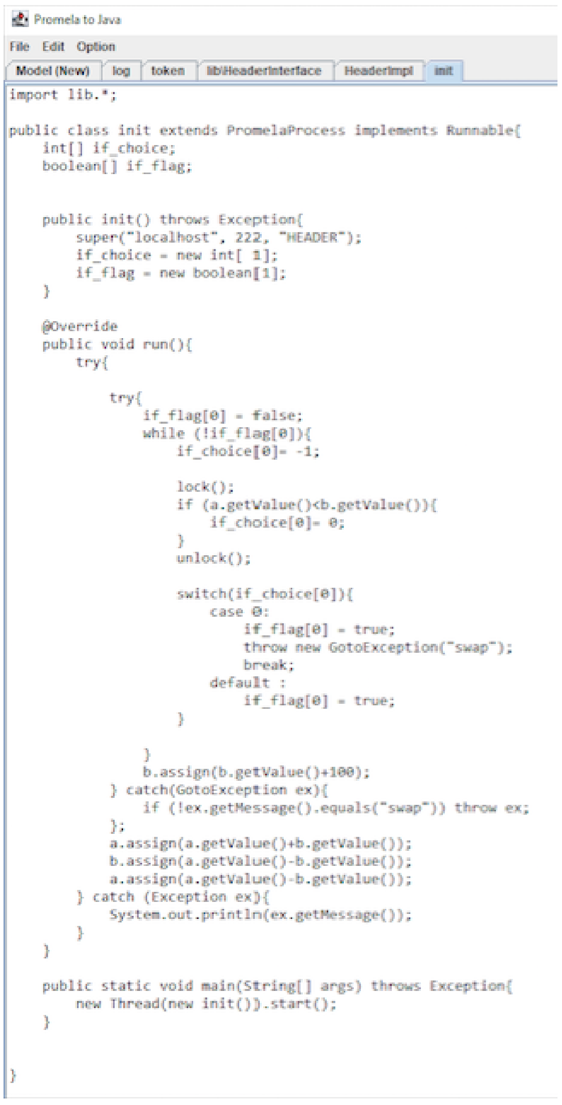
Fig. 10: The result of Type1 associated with the selected input of Fig. 8

3) Type3: In Type3, goto statement is in action part of an unless, and label is in the guard of the same unless. From Fig. 11 is seen that between goto and label are in the same unless - goto is in main sequence, while label is in an escape sequence. In this case, all statements before goto (i.e., $a = a - 10$, $a = a - 5$, and $a = a - 1$) will be executed while the first statement in escape sequence ($a < b$) is unexecutable. Because goto is the last statement in the main sequence, after goto is being executed there will be no statement skipped, and the control is sent to the label assign. The result of translation - the output of running program is shown in Fig. 13.



```
init{
  {
    a = a-10;
    a = a-5;
    a = a-1;
    goto assign;
  } unless {
    a < b;
assign:
    b = a;
  }
}
```

Fig. 11: The selected input (model) representing Type3

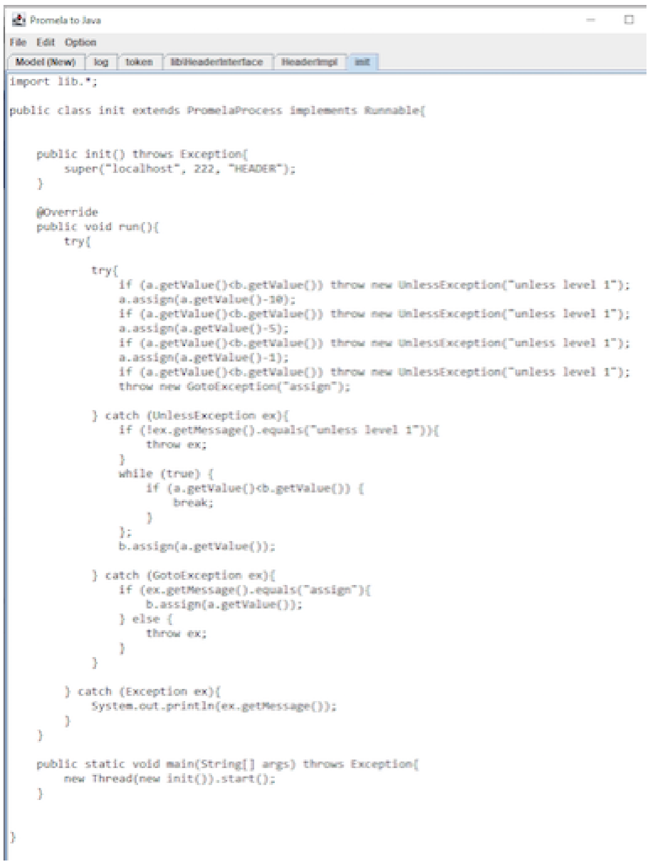


```
import lib.*;

public class init extends PromelaProcess implements Runnable{
    int[] if_choice;
    boolean[] if_flag;


    public init() throws Exception{
        super("localhost", 222, "HEADER");
        if_choice = new int[ 1];
        if_flag = new boolean[1];
    }

    @Override
    public void run(){
        try{

            try{
                if_flag[0] = false;
                while (!if_flag[0]){
                    if_choice[0]= -1;

                    lock();
                    if (a.getValue()<b.getValue()){
                        if_choice[0]= 0;
                    }
                    unlock();

                    switch(if_choice[0]){
                        case 0:
                            if_flag[0] = true;
                            throw new GotoException("swap");
                            break;
                        default :
                            if_flag[0] = true;
                    }

                }
                b.assign(b.getValue()+100);
            } catch(GotoException ex){
                if (!ex.getMessage().equals("swap")) throw ex;
            };
            a.assign(a.getValue()+b.getValue());
            b.assign(a.getValue()-b.getValue());
            a.assign(a.getValue()-b.getValue());
        } catch (Exception ex){
            System.out.println(ex.getMessage());
        }
    }

    public static void main(String[] args) throws Exception{
        new Thread(new init()).start();
    }


}
```

Fig. 12: The result of Type2 associated with the selected input of Fig. 9

III. Conclusion

The study has identified three types of goto pattern in PROMELA model that could be automatically translated into *Java* program using the exception handling mechanism in *Java*. On the other hand, there were several pattern could not be handled using the exception handling. Most of them were nested patterns of goto, and among of them cross each other. Those of three types were Type1 was the pattern where label came before goto, Type2 was the pattern where goto came before label, and Type3 was one where goto was in action of an unless, and label was in the guard of the same unless. Further more, it has been successfully developed the translation template of each type that will be used to translate the corresponding type of goto pattern into *Java* program, and a procedure to determine the right type of goto pattern correspond to the automatic translation. Subsequently, the testing was conducted using 10 cases (i.e., PROMELA model containing goto) for each type, and showed that the automatic translation worked properly. In order to handle those patterns that still could not be covered, there should be a further study that reduce the limitations of an existing exception handling mechanism to improve its capability.



```
import lib.*;

public class init extends PromelaProcess implements Runnable{


    public init() throws Exception{
        super("localhost", 222, "HEADER");
    }

    @Override
    public void run(){
        try{

            try{
                if (a.getValue()<b.getValue()) throw new UnlessException("unless level 1");
                a.assign(a.getValue()-10);
                if (a.getValue()<b.getValue()) throw new UnlessException("unless level 1");
                a.assign(a.getValue()-5);
                if (a.getValue()<b.getValue()) throw new UnlessException("unless level 1");
                a.assign(a.getValue()-1);
                if (a.getValue()<b.getValue()) throw new UnlessException("unless level 1");
                throw new GotoException("assign");

            } catch (UnlessException ex){
                if (!ex.getMessage().equals("unless level 1")){
                    throw ex;
                }
                while (true) {
                    if (a.getValue()<b.getValue()) {
                        break;
                    }
                };
                b.assign(a.getValue());

            } catch (GotoException ex){
                if (ex.getMessage().equals("assign"){
                    b.assign(a.getValue());
                } else {
                    throw ex;
                }
            }

        } catch (Exception ex){
            System.out.println(ex.getMessage());
        }
    }

    public static void main(String[] args) throws Exception{
        new Thread(new init()).start();
    }


}
```

Fig. 13: The result of Type3 associated with the selected input of Fig. 11

References

[1] A. Luvisi, "The History, Controversy, and Evolution of the Goto Statement," Sonoma State University, 2008.

[2] Anurag, Akankasha, and A. Saxena, "Implementation of Custom Exception and Its Optimization in Java," IEEE 3rd International Conference on Computing for Sustainable Global Development (INDIACom), 2016.

[3] A. Tomazos, "Explicit Flow Control: break label, goto case and explicite switch," Project of Programming Language C++ Evolution Working Group, 2014.

[4] C. Pronk, "Promela to Java - Automatic translation," Slides of TU Delft course IN4023: Advanced Software Engineering, 2007.

[5] C. Wimberger, "Source to Source Translator from C# to Java and Action Script," Journal of Kepler University Linz (KJU), 2008.

[6] D. A. Plaisted, "Source-to-Source Translation and Software Engineering," Journal of Software Engineering and Applications, 2013.

[7] E. Vielvoije, "Promela to Java, Using an MDA Approach," Thesis, 2008.

[8] F. Ebert, F. Castor, and A. Serebrenik, "A Reflection on *An Exploratory Study on Exception Handling Bugs in Java Programs*, IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 18-21 Feb. 2020, London, ON, Canada. ISSN: 1534-5351, DOI: 10.1109/SANER48275.2020.9054791, IEEE 2020, 2020.

[9] G. J. Holzmann, "The Model Checker SPIN," IEEE Transactions on Software Engineering, 1997.

[10] G. J. Holzmann, "The SPIN Model Checker: Primer and Reference Manual," Addison-Wesley Professional, ISBN 0 321 228628, 2003.

[11] H. Kondoh, and K. Futatsugi, "To use or not to use the goto statement: Programming styles viewed from Hoare Logic," Science of Computer Programming 60, pp 82-116, 2006.

[12] H. Melo, R. Coelho, and C. Treude, "Unveiling Exception Handling Guidelines Adopted by Java Developers," IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019.

[13] J. Oliveira, H. Macedo, N. Cacho, and A. Romanovsky, "DroidEH: An Exception Handling Mechanism for Android Applications," IEEE 29th International Symposium on Software Reliability Engineering (ISSRE), 15-18 Oct. 2018, Memphis, TN, USA. DOI: 10.1109/ISSRE.2018.00030, IEEE 2018, 2018.

[14] K. Jiang, "Model Checking C Programs by Translating C to PROMELA," 2009.

[15] K. Seymour. and J. Dongarra, "Automatic Translation of FORTRAN to JVM Bytecode," Concurrency and Computation Practice and Experience, vol. 15, pp 207-222, 2003.

[16] M. Asaduzzaman, M. Ahasanuzzaman, C. K. Roy, and K. A. Schneider, "How Developers Use Exception handling in Java ?," IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), 2016.

[17] M. Mehrabi, N. Giacaman, and O. Sinnen, "Unobtrusive Asynchronous Exception Handling with Standard Java Try/Catch Blocks," IEEE International Parallel and Distributed Processing Symposium (IPDPS), 21-25 May 2018, Vancouver, BC, Canada. ISSN: 1530-2075, DOI: 10.1109/IPDPS.2018.00095, IEEE 2018, 2018.

[18] M. Ceccato, P. Tonellia, and C. Matteotti, "Goto Elimination Strategies in the Migration of Legacy Code to Java," 12th European Conference on Software Maintenance and Reengineering, Athens, Greece, DOI: 10.1109/CSMR.2008.4493300, IEEE, Print ISBN: 978-1-4244-2157-2, Print ISSN: 1534-5351, 2008.

[19] M. Kechagia, T. Sharma, and D. Spinellis, "Toward a Context Dependent Java Exceptions Hierarchy," IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), 2017.

[20] M. Nagappan, R. Robbes, Y. Kamei, E. Tanter, S. McIntosh, A. Mockus, and A. E. Hassam, "An Empirical Study of goto in C code," PeerJ PrePrint—http://dx.doi.org/10.7287/peerj.preprints.826v1—CC-BY 4.0 Open Access—rec: 11 Feb 2015, publ: 11 Feb 2015, 2015.

[21] M. Nagappan, "Reconsidering Whether GOTO Is Harmful," IEEE Software, vol. 35, Issue: 3, DOI: 10.1109/MS.2028.2141020, 2018.

[22] R. Gerth, "Concise PROMELA Reference," 2012. http://www.cse.msu.edu/ cse470/PromelaManual/Quick.html.

[23] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of Exception handling Pattern in Java Projects: An Empirical Study," IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), 2016.

[24] Suprapto, R. Wardoyo, R. Pulungan, and B. Wijaya, "A Scheme of Construct Association from PROMELA Model to Java Program," Proc. 4th FTRA, 2013.

[25] Suprapto, R. Wardoyo, R. Pulungan, and B. Wijaya, "A Formal Proof of Correctness of Construct Association from PROMELA to Java," IAENG International Journal of Computer Science, vol. 42, no.4, pp313-331, 2015.

[26] T. Montenegro, H. Melo, R. Coelho, and E. Barbosa, "Improving developers awareness of the exception handling policy," IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 20-23 March 2018, Campobasso, Italy. DOI: 10.1109/SANER.2018.8330228, IEEE 2018, 2018.

[27] W. Nabialek, A. Janowska, and P. Janowski, "Translation of Timed Promela to Timed Automata with Discrete Data," John Wiley Sons, Ltd, 2008.