Improvement of Equivalent Mutant Detection Using Loop Count Restriction

Wang Weitao , Hirohide Haga
Graduate School of Science and Engineering, Doshisha University
1-3 Miyakotani, Tatara, Kyotanabe, 610-0321, Japan
wwang@ishss10.doshisha.ac.jp , hhaga@mail.doshisha.ac.jp

ABSTRACT

Software testing is an indispensable part of the software development process. Mutation analysis is regarded as an effective software testing method. By adopting mutation operators on the original program, mutation analysis generates mutants to simulate possible bugs, and then it creates a test case suite working on both mutants and the original program to analyze the difference in the results and find these bugs. Mutation analysis improves the test case suite until it is strong enough to find all possible bugs. If the result of running a mutant is different from the result of running the original program for at least one test case in the input test case suite, the seeded fault denoted by the mutant is detected. However, in the process of seeding into the fault, there is the equivalent mutant that has exactly the same behavior as the original program generated. These equivalent mutants obstruct the precise assessment of the quality of the test case suite. Therefore, it is necessary to remove them. In this article, we improve a previously proposed novel approach to equivalent mutant detection that uses symbolic computation. When we apply symbolic computation to a statement with loop description, the path explosion problem occurs. To overcome this path explosion problem, this article proposes a method that restricts the number of iterations to improve the detection ratio of equivalent mutants.

KEYWORDS

Software Testing, Fault-based Testing, Mutation Analysis, Structured Programing, Test Cases

1 INTRODUCTION

With the rapid progress of information technology, the dependence of human society on information systems has grown steadily. To meet people's growing demand for software functions, the scale and complexity of software will also continue to expand. The stability and security of the software are important attributes and constitute an important foundation to ensure the integrity of software functions. However, the software-building process, including design, development and maintenance, may contain some defects that can lead to software vulnerabilities. Such defects in various types of software are unavoidable, and once they have been discovered by hackers, they can allow unauthorized access or system damage. Therefore, software testing is an indispensable part of the software development process.

Mutation analysis[1] is regarded as an effective software testing method. By adopting mutation operators on the original program, mutation analysis generates mutants to simulate the possible bugs, and then it creates a test case suite working on both mutants and the original program to analyze the difference in the results and find these bugs. Mutation analysis improves the test case suite until it is strong enough to find all possible bugs. Thus far, there have been few studies on the evaluation method of the software test case suite. Unlike other software testing methods, mutation analysis is a structural testing method aimed at improving the adequacy of the test case suite. In mutation analysis, the faulty programs are generated artificially by making syntactic changes in the original program. The faulty program is called a mutant, and each mutant contains only one syntactic change. To assess the quality of a given test case suite, these mutants are executed against the input test case suite. If the result of running a mutant is different from the result of running the original program for at least one test case in the input test case suite, the seeded fault denoted by the mutant is detected. However, in the process of seeding into the fault, there is an equivalent mutant[2] with

exactly the same behavior as the original program generated. These equivalent mutants obstruct the precise assessment of the quality of the test case suite. If large numbers of equivalent mutants are included in the set of mutants, the mutation score remains low, and testers cannot decide whether to add more test cases. Therefore, it is necessary to remove them. In this article, we improve a previously proposed novel approach[3] to equivalent mutant detection that uses symbolic computation[4]. When we apply symbolic computation to a statement with loop description, the path explosion problem occurs. To overcome this path explosion problem, this article proposes a method that restricts the number of iterations to improve the detection ratio of equivalent mutants.

2 BRIEF SUMMARY OF MUTATION ANALYSIS

2.1 Mutation analysis overview

Mutation analysis is a method of white box testing in software testing[1], with its purpose to measure and improve the quality of the test case suite instead of testing the program itself. In mutation analysis, program faults are intentionally injected into a program. The error-injected faulty program is called a mutant. The syntactic modifications responsible for a mutant are determined by a set of mutant operators. This set is determined by the language of the program being tested and the mutation system used for testing. Since there are several possibilities in error injection, one original code will generate several different mutants. Consider the example shown in Fig. 1: the mutation operator *Arithmetic Operator Replacement* (**AOR**) replaces each occurrence of one of the arithmetic operators ($+, -, \star, /$) with another arithmetic operator. Figure 2 shows the program before being modified. We call it the original. Figure 3 shows a typical example of a mutant. In this program, the assignment operator was changed from "+=" to "−=".

```
                  a - b
a + b   ⟹        a * b
                  a / b
```

Figure 1. Mutation operator **AOR**

```
int sum (int x) {
   int sum = 0;
   for (int i = 1; i <= x; i++) {
      sum += i;
   }
   return sum;
}
```

Figure 2. Original (`Sum`)

2.2 Mutation operator

A mutation operator is a rule that specifies syntactic variations of strings generated from a grammar. Mutation operators are created with one of two goals: to induce simple syntax changes based on errors that programmers typically make (for example, using the wrong variable names) or to force common testing goals (for example, higher coverage)[6]. In our system, we used the following mutation operator shown in Tab.1: each of the mutation operators is represented by a three-letter acronym.

2.3 Equivalent Mutant

In the process of mutant creation, some mutants with specific features may be created. Consider the following mutant shown in Fig.4. In the case of Fig.4, the statement of the return value was modified by changing "`return sum`" to "`return Math.abs (sum)`". If the statements within the `for` loop do not modify the value of `i`, the original program and

```
int sum (int x) {
   int sum = 0;
   for (int i = 1; i <= x; i++) {
      sum -= i;
   }
   return sum;
}
```

Figure 3. Example mutant of `Sum`

ISBN: 978-1-941968-18-5 ©2015 SDIWC

Table 1. Mutation operators used in this article

Operator	Description
ABS	Absolute Value Insertion
AOR	Arithmetic Operator Replacement
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
ASR	Assignment Operator Replacement
UOI	Unary Operator Insertion
UOD	Unary Operator Deletion
SVR	Scalar Variable Replacement
BOM	Bomb Statement Replacement
BVI	Border Value Increment

```
int sum(int x){
   int sum = 0;
   for(int i=1; i<=x; i++){
   sum += i;
  }
   return Math.abs(sum);
}
```

Figure 4. Equivalent Mutant of `Sum`

its mutant produce identical outputs. Therefore, no test case can detect this kind of mutant. Such a mutant is called an ***equivalent mutant***[3]. In other words, an equivalent mutant has a syntactically different part but is semantically identical to the original program.

2.4 Mutation Score(MS)

Mutation score is an objective measure to evaluate the test case suite adequacy against mutation testing. From one original source program, several mutants are generated. After the generation of sufficient mutants, the original program and its mutants are executed using the same test case suite (set of test cases). A mutant is said to be killed when the output of the original and mutant are different. When the mutant is killed, this mutant is considered to be dead. A mutant is an equivalent to the given program if it always produces the same output as the original program. The quality of the test case suite can be computed by the following ratio called the ***mutation score***:

$$MS = \frac{D}{M - E} \times 100 \qquad (1)$$

where

- D: the number of killed mutants,
- M: the number of all mutants,
- E: the number of equivalent mutants.

A mutation score for a set of test cases is the percentage of non-equivalent mutants killed by the test case suite. The test case suite is said to be mutation-adequate if its mutation score is 100%.
Equivalent mutants reduce the mutation score (MS) because no test case can detect them. MS is an important indicator of the decision about test case suite quality. To estimate the test case suite 's quality, equivalent mutants must be removed from the set of mutants. As for detecting whether a program and one of its equivalent mutant programs is theoretically undecidable, additional human efforts are necessary to find and remove the equivalent mutants from the set of mutants.

2.5 Symbolic Computation

Symbolic computation is a natural extension of conventional computation[4, 7]. It is used, for example, to transform original programs into more effective programs[9] or to generate test cases automatically[7]. Symbolic computation uses an input variable as a symbol value to simulate programs and various operations on the symbol. Unlike conventional program computation based on actual data values, the symbolic computed value of the variable is composed of symbols and constant expression. In our system, symbolic computation is adopted to detect equivalent mutants. Symbolic computation uses input values as symbolic values instead of actual concrete values. Therefore, the output of symbolic computation is not an actual value but a function or expression of symbolic inputs. Symbolic computation generates a specific condition (path condition) to get a specific symbolic output[3, 4].
Consider the program "min" in Fig. 5 as an example. In normal computation, the concrete values of x and y are given. Then, the function

returns smaller values of x and y. However, in symbolic computation, the result shown in Table 2 is obtained.

```
int min(int x, int y){
    int min = x;
    if (x > y) {
        min = y;
   }
    return min;
}
```

Figure 5. Function min

Table 2. Symbolic computation result

Computation result	Path condition
y	x>y
x	!(x>y)

In this table, "**Computation result**" is the symbolic result of the computation and "**Path condition**" means the condition of the symbolic inputs to get the corresponding result. In this program, there are two computation result possibilities: y and x. To get the y result, the corresponding path condition (x>y) must hold. Symbolic computation generates all of the possible output results and their path condition. Since it generates more details of computation information than normal concrete computation, equivalent mutants may be detected by this rich information.

2.6 XML and JavaML

XML is a markup language describing the structural information of documents. It is a subset of SGML, the Standard Generalized Markup Language. The XML language is designed to represent the set of data in an easily understandable format both for machine and human being. JavaML[8] is one instance of XML (Extensible Markup Language). JavaML is used for describing Java programming language. It provides a complete self-describing representation of Java source code in XML. In this article, the converter **JJmlt.jar**[5] is applied to convert Java source code into JavaML. Fig.6 shows a simple XML document which is sufficient for containing the syntax of XML document. XML document consists of XML declaration, element and attribute. Here we simply introduce the JavaML, the detail content will be provided in Section 3.

```
<? xml version="1.0" encoding="utf−8"?>
<Root>
  <user id= "001 ">
    <admin>
      <name>wwang</name>
      <password>islab</password>
      <age>27</age>
    </admin>
    <admin>
      <name>zcui</name>
      <password>123456</password>
      <age>28</age>
    </admin>
  </user>
</Root>
```

Figure 6. Sample of XML document

3 IMPLEMENTATION OF THE DETECTION FUNCTION OF EQUIVALENT MUTANTS

Without detecting all equivalent mutants, the mutation score will never reach 100%. Thus, the tester will not have complete confidence in the program and the test data. Also, the tester will be left wondering whether the remaining mutants are equivalent or the test case suite is insufficient. Detecting equivalent mutants by hand is very time-consuming, which contributes to making the cost of mutation analysis prohibitively high. Therefore, we proposed a method for automatically detecting equivalent mutants using symbolic computation[3]. Symbolic computation executes a given program by providing symbolic values instead of concrete values. The final result of symbolic computation is a pair of expressions; one is a computation result and the other is a path condition, both of which may contain symbols. The result of equivalent mutants is identical to that of its original for any input. Thus, a mutant is proved to be an equivalent mutant if the symbolic output of the mutant is identical to that of the original[3].

The implemented system is for Java programming language. However, the concept of the

proposed system can be applied to any programming language. First, the system converts an original and its mutants represented in Java to JavaML. Then, both programs represented in JavaML are executed symbolically, and the outputs for arbitrary inputs are generated. Then, if the output of the mutant is equivalent to that of the original, the system determines that the mutant is an equivalent mutant. The evaluation experiments proved that the proposed method detects more equivalent mutants than conventional methods in simple programs with assignment instructions and branch instructions.

3.1 JavaML as an intermediate representation of source code

Machine code is not suitable to symbolically execute a program. No information about such symbols as names is stored in the machine code. Nor is the textual form of the original source code appropriate for symbolic computation. This is partly because no structural information, such as "if then structure", is explicitly represented in textual form. To extract the necessary information for symbolic computation, such additional processing as both lexical and syntax analysis is necessary. Since these require additional computational cost, appropriate intermediate representation should be used for symbolic computation[3].

The prototype system uses the XML form as an intermediate representation. More specifically, it uses JavaML. With JavaML, both humans and machines can recognize sufficient information of the source program[8]. JavaML representation holds all the syntactical information and some additional semantic information. Since JavaML is an application of general XML representation, various kinds of software libraries are available for XML processing that help develop programs for processing JavaML. Appendix A is the JavaML representation of the function `sum` in Fig. 2.

3.2 Symbolic computation on JavaML

We implemented a symbolic computation program for Java using C#. LINQ to XML is also used as an XML processing tool. Since JavaML can be seen as tree representation of source code, a symbolic computation program traverses a JavaML tree in a post-order and invokes an appropriate routine to execute program components[3].

When symbolic computation starts, one specific object called the "symbolic computation status" is created. This object, which contains all the status information of each execution path, consists of three fields: (1) evaluation result field, (2) path information field, and (3) all the stored values of the local variables. All the fields are updated when each node in the JavaML tree is executed.

Consider the JavaML program given in Appendix A. This is an XML representation of the original program shown in Fig. 2. For example, after traversing this JavaML representation in post-order, we could get the computation result by using the symbolic computation method we proposed. Appendix B is a symbolic computation result of the JavaML-based program. By comparing the computation results of the original and the mutant, we can find the equivalent mutant.

3.3 Symbolic computation problem

In practice, symbolic computation always has a path explosion problem for large programs. The increase of line of code (LOC) causes an exponential growth of the feasible paths, which would even be infinite when an unbounded loop exists in the program. This article focuses on this problem of symbolic computation, which contains iterations implemented by loop statements. Such a problem will lead the symbolic computation path to become infinite, and computation cannot be stopped. Consider the following mutant shown in Fig. 7.

In this case, this mutant was generated by modifying line 7 from "`return v`" to "`return Math.abs (v)`". `v` is substituted for 1 as the initial value because the factorial computa-

ISBN: 978-1-941968-18-5 ©2015 SDIWC

```
int factorial(int x) {
    int i;
    int v = 1;
    for (i = x; i > 0; i--) {
        v *= i;
    }
    return Math.abs(v);
}
```

Figure 7. Program factorial

tion results of `v` and `Math.abs(v)` in line 7 are always identical; so this kind of mutant is an equivalent mutant.

Since the location of the change of equivalent mutant is line 7, symbolic computation must reach line 7. The computation result is shown in Table 3. Because the number of iterations depends on input parameter "`x`", the symbolic computation program cannot determine how many times the iteration part of the program is executed. Features of this kind of equivalent mutant are as follows: the mutation point is after the iteration statement, and it contains an iteration implemented with a non-fixed number of iterations; the number of iterations cannot fit in a finite number of times; and the symbolic computation result cannot be constructed.

3.4 Implementation of proposed method

The above explosion problem may be overcome by restricting the number of iterations, which can improve the detection ratio of an equivalent mutant. Specifically, we set an upper bound of iteration. When the number of iterations reaches the upper bound, even if the target statement has not been covered, symbolic computation will be stopped immediately.

There are mainly three types of loops in Java: `for`, `while` and `do-while`. Consider the following example shown in Fig. 8. All of them could be divided into four parts: init, test, update and loop body.

In symbolic computation, the process to handle loop statements is shown in Table 4.

As the process in Table.4 shows, each output corresponds only to one execution path. Suppose there are N execution paths gener-

```
for (inin; test; update)
{
   loop-body;
}
```

(a) `for` loop

```
init;
while(test) {
   loop-body;
}
```

(b) `while` loop

```
init;
do {
   loop-body;
} while (test);
```

(c) `do-while` loop

Figure 8. Three types of loop statement

ated after the symbolic computation. Then, N becomes constant when it has no relation to loop-index and method-parameter; otherwise N would be infinite and path explosion would occur.

If N is constant, only N execution paths need to be covered; thus all of the equivalent mutants could be detected. When N is infinite, only the first k paths could be compared. Consider the example shown in Fig.9; we assume that the set value of k is 2. When the mutant part exists within the loop, we cannot detect all of the equivalent mutants and *Detection Error* may occur.

When k equals 2, there are two computation paths being handled. The computation results of the above two programs are shown in Table 5.

From this example, we can see that the mutant would be determined as an equivalent mutant incorrectly. Obviously, by increasing the value of k, we could improve the accuracy of equivalent mutant detection. In conclusion, the accuracy and detection ratio could be improved by increasing as much as possible the computation paths in the loop condition.

Table 3. Symbolic computation result of `factorial`

Result	Condition
1	`x <= 0`
1	`x > 0 && !(x - 1 > 0)`
2	`x > 0 && x - 1 > 0 && !(x - 2 > 0)`
...	...
`n*(n-1)*...* 1`	`x > 0 && x - 1 > 0 && ⋯ && !(x - n > 0)`
...	...

Table 4. Process of loop statement

Step	Content
1	initial $\longrightarrow$ output 0
2	test
3	loop body
4	update $\longrightarrow$ output1
5	test
6	loop body
7	update $\longrightarrow$ output1
...	.....
N	update $\longrightarrow$ output1
...	.....

Table 5. Symbolic computation result

Computation result	Path condition
2	`x<=0`
4	`0<x<=1`

```
int method(int x){
  int n=2;
  for (int i=0;i<x;i++){
    n*=2;
  }
  return n;
}
```

(a) Original Program

```
int method(in
  int n=2;
  for (int i=0;i<x;i++){
    n+=2;
  }
  return n;
}
```

(b) Mutant Program

Figure 9. Original and mutant

4 EXPERIMENT

In this section, we describe the experiments we conducted. Using the proposed method, we conducted experiments assessing how many generated equivalent mutants can be detected. By restricting the number of iterations, compared to the previously proposed approach, we expect to see improvement in the detection ratio. In the experiment, we used Windows 7 Ultimate, running on an Intel Core i7-2600 CPU 3.40 GHz with 12GB RAM.

4.1 Experimental contents

The following are steps of the experiment.

1. Apply the mutation operators to the origi-

Table 6. Experimental results

Program Name	N_{tm}	N_{em}	N_{dm}	r
Sum	17	2	2	100%
Pi	39	3	3	100%
Factorial	18	2	2	100%
MaxAbs	44	13	13	100%
NewtonSqrt	69	13	5	38%
FactRec	35	4	2	50%
MoonAge	569	46	12	26%

nal program to generate mutants.

2. Execute all programs (original program and its mutants) on test case suite.

3. Select candidates for equivalent mutant by comparing the computation results of the original and the mutants. Mutants with identical results are possibly equivalent mutants.

4. For all equivalent mutant candidates, apply the proposed method to check if they are equivalent mutants or not.

5. Determine whether the equivalent mutants are detected by the proposed method.

4.2 Experimental results

Table 6 shows the results of the experiment. In Table 6,

- N_{tm}: number of total mutants
- N_{em}: number of equivalent mutants
- N_{dm}: number of detected equivalent mutants
- r: detection ratio ($= N_{dm}/N_{em}$).

4.3 Discussion

This article mainly overcomes the path explosion problem for loop description, and thus all programs to be tested contain loop iteration. From the experimental results, compared to the previously proposed approach, we can see the proposed method has made great progress.

```
...  ......
38     int m;
39     for (m=1; m<month; m++)
40     {
41      saday += calcM(m, year);
42     }
43     return saday;
44     }
```

Figure 10. A part of MoonAge program

Using the proposed method, the detection ratios of Factorial and MaxAbs were 100%; all of the equivalent mutants were automatically detected. The previous method provided the detection ratios of MoonAge 26%, MaxAbs 38%, Factorial 50%, and MoonAge 13%[3]. This experiment proved that the detection ratio is improved greatly. Because symbolic computation needs to compare all the path conditions and corresponding computation results, when encountering the programs containing iteration statements with a non-fixed number of iterations, the number of path conditions is infinite, and symbolic computation cannot be stopped in the previous method. By restricting the number of iterations for a small-scale program, all of the equivalent mutants were detected.

However, several programs' detection ratios were still not ideal. These programs usually contained complex iteration statements and had a fairly large scale. Consider the program shown in Fig. 10.

Executing such programs by symbolic computation will lead to the stack overflow exception problem. Therefore, in order to effectively alleviate the symbolic computation path explosion problem, we should make use of large computing power, such as distributed multiprocessor, multi-core, and cloud computing, which enable the traditional symbolic computation to execute in parallel.

ISBN: 978-1-941968-18-5 ©2015 SDIWC

5 CONCLUSION

This article proposed a method to improve the previous approach to equivalent mutant detection that uses symbolic computation. Several studies exist on the automatic detection of equivalent mutants. Our proposed method involves restricting the number of iterations and generating shorter and simpler computation results and path conditions to get results. By comparing these two pieces of information, equivalent mutants can be detected more effectively than by the conventional method. Also, by restricting the number of iterations, the detection ratio shows obvious improvement.

By restricting the number of iterations for a small-scale program, all of the equivalent mutants were detected. However, several programs ' detection ratios were still not ideal. These programs usually contained complex iteration statements and had a fairly large scale. Due to the huge number of branches and iterations, there exists exponential growth of the execution path; thus, the practical application of symbolic computation will encounter the potential path explosion problem, which has become the bottleneck of symbolic computation applications. To effectively alleviate the symbolic computation path explosion problem, in the future, we should make use of large computing power, such as distributed multiprocessor, multi-core[11], and cloud computing, which enable the traditional symbolic computation to execute in parallel.

REFERENCES

[1] P. Ammann, J. Offutt, *Introduction to software testing*, Cambridge University Press, Cambridge, UK, 2008.

[2] J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths, " *Journal of Software Testing, Verification and Reliability*, **7**(3), pp.165-192, 1997.

[3] T. Ueshiba, H. Haga, "Detecting Equivalent Mutants Using Symbolic Computation," *Proceedings of the International Conference on Electrical, Electronics, Computer Engineering and their Application*, pp.6-11, 2014.

[4] J.C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, **19**(7), pp.385–394, (1976)

[5] H. Aman, "JJmlt, A Java-JavaML Translator,", `http://se.cite.ehime-u.ac.jp/tool/JJmlt/`

[6] J. M. Bieman, S. Ghosh and R. T. Alexander. "A Technique for Mutation of Java Objects," *Proceedings.16th Annual International Conference on. Automated Software Engineering (ASE)*, pp. 337–340, 2001.

[7] Y. Futamura "Partial evaluation of computation process – an approach to a compiler-compiler," *Higher-Order and Symbolic Computation*, **12**(4), pp.381–391, 1999.

[8] G. Badros, "JavaML: A Markup Language for Java Source Code," *Proceedings of 9th International World Wide Web Conference on Computer networks*, pp.159–177, 2000.

[9] A. Coen-Porisini, "Software specialization via symbolic execution," *IEEE Transactions on Software Engineering*, **SE-17**(9), pp.884–891, 1991.

[10] C Cadar, K Sen, "Symbolic execution for software test: three decades later," *Communications of the ACM*, **56**(2), pp.82–90, 2013.

[11] M. Staats, C. Păsăreanu, "Parallel Symbolic Execution for Structural Test Generation," *Proceedings of the 19th international symposium on Software testing and analysis*, pp.183–194, 2010.

A JAVAML REPRESENTATION OF THE FUNCTION SUM

Following is a JavaML representation of the function `Sum` in Figure.2.

```
<java-source-program>
  <java-class-file name="Test\Sum\Test.
      java">
   <class name="Test">
     <superclass name="java.lang.Object" />
     <method name="test" visibility="public
         " static="true" id="Test:mth-1">
       <type name="int" primitive="true" />
       <formal-arguments>
         <formal-argument name="x" id="
             Test:frm-1">
           <type name="int" primitive="true
               " />
         </formal-argument>
       </formal-arguments>
       <block>
         <local-variable name="sum" id="
             Test:var-1">
           <type name="int" primitive="true
               " />
           <literal-number kind="integer"
               value="0" />
         </local-variable>
         <local-variable name="i" id="Test:
             var-2">
           <type name="int" primitive="true
               " />
           <literal-number kind="integer"
               value="1" />
         </local-variable>
         <loop kind="for">
           <test>
             <binary-expr op="&lt;=">
               <var-ref name="i" idref="
                   Test:var-2" />
               <var-ref name="x" idref="
                   Test:frm-1" />
             </binary-expr>
           </test>
           <update>
             <unary-expr op="++" post="
                 true">
               <var-ref name="i" idref="
                   Test:var-2" />
             </unary-expr>
           </update>
           <block>
             <assignment-expr op="+=">
               <lvalue>
                 <var-set name="sum" />
               </lvalue>
               <var-ref name="i" idref="
                   Test:var-2" />
             </assignment-expr>
           </block>
         </loop>
         <return>
           <var-ref name="sum" idref="
               Test:var-1" />
         </return>
       </block>
     </method>
   </class>
  </java-class-file>
</java-source-program>
```

B SYMBOLIC EXECUTION RESULT

Following is an execution result represented in XML.

```
<output>
  <value>
    <binary-expr op="+">
      <literal-number kind="integer" value=
          "0" />
      <literal-number kind="integer" value=
          "1" />
    </binary-expr>
  </value>
  <constraint>
    <binary-expr op="&amp;&amp;">
      <binary-expr op="&amp;&amp;">
        <literal-boolean value="true" />
        <binary-expr op="&lt;=">
          <literal-number kind="integer"
              value="1" />
          <var-ref name="x" />
        </binary-expr>
      </binary-expr>
      <unary-expr op="!" post="false">
        <binary-expr op="&lt;=">
          <binary-expr op="+">
            <literal-number kind="integer"
                value="1" />
            <literal-number kind="integer"
                value="1" />
          </binary-expr>
          <var-ref name="x" />
        </binary-expr>
      </unary-expr>
    </binary-expr>
  </constraint>
</output>
<output>
  <value>
    <binary-expr op="+">
      <binary-expr op="+">
        <binary-expr op="+">
          <literal-number kind="integer"
              value="0" />
          <literal-number kind="integer"
              value="1" />
                </binary-expr>
        <binary-expr op="+">
          <literal-number kind="integer"
              value="1" />
          <literal-number kind="integer"
              value="1" />
        </binary-expr>
      </binary-expr>
      <var-ref name="i" />
    </binary-expr>
  </value>
  <constraint>
    <binary-expr op="&amp;&amp;">
      <binary-expr op="&amp;&amp;">
        <binary-expr op="&amp;&amp;">
          <binary-expr op="&amp;&amp;"
              >
            <literal-boolean value="true"
                />
            <binary-expr op="&lt;=">
```

```
                <literal−number kind="integer
                    " value="1" />
                <var−ref name="x" />
              </binary−expr>
            </binary−expr>
            <binary−expr op="&amp;lt;=">
              <binary−expr op="+">
                <literal−number kind="integer
                    " value="1" />
                <literal−number kind="integer
                    " value="1" />
              </binary−expr>
           <var−ref name="x" />
            </binary−expr>
          </binary−expr>
          <binary−expr op="&amp;lt;=">
            <var−ref name="i" />
          <var−ref name="x" />
        </binary−expr>
      </binary−expr>
      <unary−expr op="!" post="false">
          <binary−expr op="&amp;lt;=">
            <binary−expr op="+">
            <var−ref name="i" />
              <literal−number kind="integer"
                  value="1" />
            </binary−expr>
            <var−ref name="x" />
          </binary−expr>
        </unary−expr>
      </binary−expr>
    </constraint>
</output>
```