# Automatic Test Cases Generation for C Written Programs Using Model Checking

Daniset González Lima
*Computer Science Area*
*CINVESTAV unidad Guadalajara*
Jalisco, México
Daniset.Gonzalez@cinvestav.mx

Raúl E. González Torres
*Computer Science Area*
*CINVESTAV unidad Guadalajara*
Jalisco, México
raul.gonzalez@cinvestav.mx

Pedro Mejía Alvarez
*Computer Science Area*
*CINVESTAV unidad Guadalajara*
Jalisco, México
pedro.mejia@cinvestav.mx

*Abstract*—**The present work focuses on the development of a tool that automatically generates coverage criteria based test cases from a C written program. For accomplishing this, the tool translates the C code into PROMELA and generates specifications based on the wanted coverage criteria. Once the model (PROMELA code) and specifications are obtained, it uses SPIN model checker for executing the verification and generating counterexamples which can be used as test cases.**

*Keywords*—**model checking, compiler, linear temporal logic, PROMELA, SPIN, software testing, test cases, coverage criteria**

## I. Introduction

Model Checking (MC) is an algorithmic method that takes as input a model and a specification in the form of temporal logic formula, and computes whether the model satisfies or not the given specification. If the model does not satisfies the specification, it returns a counterexample as an execution trace going from the initial state of the model, to the state where the specification is evaluated false.

The counterexample generation feature has been exploited for more than 20 years, in the search of techniques that allow to automate the model based tests generation [1]. As a consequence, different approaches have been obtained, differing, mainly, in the way to obtain the model, how to represent it and how to select the test cases. In general, the final idea of all approaches is the same: using counterexamples obtained by MC as test cases.

The idea of using MC for test cases generation was first proposed by Callahan et al. [2] in 1996. These authors described a strategy about the use of MC for analyzing a program execution traces, dividing them in equivalence partitions, and using counterexamples for creating new test cases for equivalence partitions with few traces.

In 1997 Engels et al. [3] presented the idea used as a basis in most of later approaches, the present work included. It consists in how to use a formal model describing a system behaviour, and the negation of formulas describing a test purpose, for generating counterexamples that can be used as test cases.

In 1999 Gargantini and Heitmeyer [4] introduced the coverage criteria based test purposes automatic generation, for SCR systems (Software Cost Reduction, [5]).

This general idea is used in different later works, differing mainly in the type of model to use and the coverage criteria to analyze. Some examples are the works of Heimdahl et al. [6], focused on transition systems coverage; Hong and Lee [7] focused on control and data flow based coverage; Gargantini et al. [8] focused on abstract state machines coverage.

All of this mentioned works, either assumed the existence and correctness of a model, or they built it from system formal specifications.

The first tools that focus on the extraction of the model from the program code, were Modex and Java Path Finder (JPF), first developed in 1999. They both represent the model as PROMELA code for later use of MC with SPIN tool. Model focused on model extraction from C code [9], while JPF did it from Java code [10].

In 2000 Corbett et al. [11] developed the tool Bandera, which, like JPF, extracts the formal model from Java code, representing it not only in PROMELA, but also in other model checkers languages.

In 2009 Ke Jian, in his masters degree thesis [12], takes the idea from these mentioned tools, and develops his own tool for translating from C to PROMELA, adding characteristics to the C subset that were not supported by previous tools. Some of these characteristics are work with pointers, functions calls and one dimension arrays.

The main contribution of the present work is to put together all these strategies red into the first tool which covers the complete process, from the translation of the source code into a model (PROMELA code), to automatic test case generation according to some coverage criteria. For achieving this, we translated all the studied temporal logic representations of coverage criteria into LTL logic, in order to use them in a single framework.

## II. The Preliminaries

### A. Software Testing

Software testing consists on the dynamic verification of a program behaviour, versus its expected behaviour, with a finite test cases set, selected from the execution domain [13].

Because the execution domain can be extremely big (usually infinite), executing all possible cases is not usually a practical or possible solution. This brings the need of developing criteria for the selection of finite test sets which cover the maximum possible cases in regarding what you want to test. In this work we will explore two criteria groups:

1) Control flow oriented coverage criteria: It covers cases related with the program control flow. Some of the criteria it includes are:
    - Decision coverage (DC).
    - Condition coverage (CC).
    - Modified condition/decision coverage (MC/DC).
2) Data flow oriented coverage criteria: It covers the paths containing definitions and/or uses of program variables. Some of the criteria it includes are:
    - All definitions coverage (All-defs).
    - All uses coverage (All-uses).

Even knowing the different coverage criteria, manually generating all possible cases according to them, could be quite complicated. Besides, it can introduce the always possible human error. Then, we need to develop techniques which help us automate the process.

### *B. Temporal Logic*

Temporal logic is an extension of propositional logic, where truth values change in terms of time.

There are different classifications of temporal logic according to various visions of time. Two of these are Computational Tree Logic (CTL), and Linear Temporal Logic (LTL) [14].

LTL logic is characterized by being propositional temporal logic in which time is linear and discrete, and future extends infinitely or is not bounded. It extends propositional logic by taking its boolean operators and adding temporal operators.

**Basic boolean operators**

- Logical AND ($\wedge$).
- Logical OR ($\vee$).
- Negation ($\neg$).
- Implication ($\Rightarrow$).
- Double implication ($\Leftrightarrow$).

**Basic temporal operators**

- Globally ($\Box$): When applied to a formula $\varphi$, $\Box\varphi$ means that "$\varphi$ is always true".
- Eventually ($\Diamond$): When applied to a formula $\varphi$, $\Diamond\varphi$ means that "eventually a state will be reached, where $\varphi$ is true".
- In the next state ($\bigcirc$): When applied to a formula $\varphi$, $\bigcirc\varphi$ means that "in the next state $\varphi$ is true".
- Until ($U$): It takes two formulas, $\varphi$ and $\psi$, as parameters, so that $\varphi U \psi$ means that "$\varphi$ will be true until a state is reached where $\psi$ is true".

### *C. Model Checking*

MC is a formal verification automatic method, used to test the correctness of system models, both software and hardware.

As figure 1 shows, the algorithm takes as input a system model $M$ and a property $\varphi$ and verifies if $\varphi$ is satisfied or not by $M$. In case of being false, it also returns a counterexample.

In LTL, both the system model and the property to verify can be represented as Büchi automata. This is how it's done (internally) by SPIN tool.

For executing the verification, the algorithm needs, in the worst case, to exhaustively explore all of the model states space, which results in the still unresolved MC problem, the state explosion.

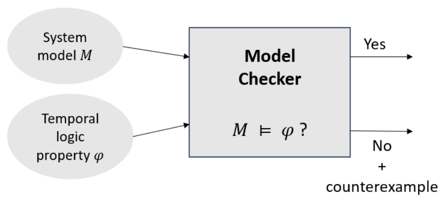


Fig. 1: Basic MC diagram

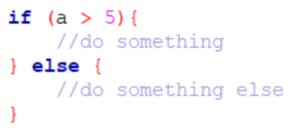
Even if the state explosion problem is still unresolved, some tools, like SPIN and SMV, have been developed, implementing optimized techniques to attack it. Examples of these techniques are: symbolic algorithms, partial order reduction and abstraction [14].

### *D. Automatic test case generation via MC*

If we have a formal model which describes the behaviour of the system under test, then the resulting counterexamples generated traces when applying MC, will match with real program execution traces. Then, we can interpret these traces and generate the corresponding test cases.

In order to generate the test cases regarding some coverage criteria, we need properties that, when verifying them against the model, it returns the counterexamples we need. This can be done by generating what we know as test purposes.

A test purpose is a property expressed as temporal logic, which describes something we want to test from the program.



```
if (a > 5){
    //do something
} else {
    //do something else
}
```

Fig. 2: If-else sentence example in C

For example, let's suppose we have the code fragment shown in figure 2 and we want a test case which makes the if condition to be true. For this, we create the test purpose: $\Diamond(a > 5)$, which tells us the condition $a > 5$ will eventually be true. Therefore, an execution trace satisfying this property can be used as a test case where the condition is true. The problem is MC doesn't return a counterexample if the condition is true, but only when it is false. Nevertheless, if instead of using MC with the previous property, we do it with its negation, $\neg\Diamond(a > 5)$, which tells us is not true that the property $a > 5$ will be eventually true, or equivalently, that the property $a > 5$ will never be true, then the algorithm will return a counterexample (in case of the property being false), which goes from the initial state of the model, to a state where the $a > 5$ is true, therefore obtaining the execution trace we first wanted.

This idea is the base for automatic test case generation via MC. If we need a test set which satisfies some coverage criteria, it's enough with generating test purpose set which covers all cases for the criteria and doing MC with the negation of its logic formulas and the model. This way we obtain a counterexample set which we can later interpret and translated into the wished test cases set.

## *E. SPIN*

SPIN is a model checker developed by Gerard J. Holzmann for the verification of communication protocols, which awarded him with ACM Software Systems Award in 2001 [15].

In order to execute MC, SPIN takes as input a model written in PROMELA (a programming language which has a C-like code). Once obtained the model, verification can be done through assertions or by expressing the wished property as LTL formula. If, while doing the state exploration, a state is found, where the assertions or LTL formula are false, then the tool returns false, and provides a counterexample containing an execution trace containing this state.

## III. System Model

During this work research, some tools were found, as those mentioned in section I, that achieve the automatic model extraction. But none was found, covering the complete process from model extraction to automatic test case generation. This work is intended to be the first tool achieving all this process in an automatic way, also achieving an unification of the studied strategies. For this, the tool consists of three general steps:

- First step consists on a compiler that takes a C written program as input and returns a PROMELA model.
- Second step consists on a specification generator which uses the obtained model from first step and, according to the wished coverage criteria, generates a specifications set, expressed as LTL logic formulas.
- Third step consists on a test cases generator which uses SPIN for implementing MC with the model and specifications, and generating counterexamples that can be later interpreted as test cases.

This whole process can be observed in figure 3.

Given that, as we mentioned before, this is the first tool which unify this whole process, we don't have a way of comparing it with any other tool (as there is no other equal purpose tool) and, therefore, we will only present it as a novel strategy which could be used as a base for creating new tools and strategies.

## IV. Translating C Into Promela

For the tool's first step we need to obtain a model which describes to the best possible the program code behaviour. To accomplish this, we developed a compiler that takes a C written program as input and translates it into a PROMELA written model.

The compiler was developed by following Ghoulum's strategy in [16]: you start with a compiler that accepts a small, trivial set from the source language, and then, incrementally, you add new features from the language step by step. In a first step you only return constants, then you add arithmetic operations and you go on like this, in a way that each step is small enough to be manageable, and at the end of each step you obtain a working compiler.

This section gives an overview of the obtained compiler and the C subset it supports.

### *A. Supported tokens list*

During the lexing step, the compiler must translate the source code into a tokens list.

Token is the smaller unit of a C program. It corresponds to each keyword and special character. A final summary of tokens supported by the tool can be observed in tables IV.1 and IV.2.

In addition, a token can be an identifier (a variable or function name), or an integer constant.

TABLE IV.1: Supported C special characters

| | | | |
|---|---|---|---|
| { | } | ( | ) |
| [ | ] | , | . |
| ; | ” | + | - |
| * | / | % | ∼ |
| & | — | = | < |
| > | ! | : | ? |

TABLE IV.2: Supported C keywords

| | | |
|---|---|---|
| int | bool | void |
| struct | return | if |
| switch | case | default |
| for | while | goto |
| do | break | continue |
| else | printf | scanf |

### *B. Supported C subset*

This work uses Ke Jiang's thesis [12] as the basis for the compiler step. In addition to the supported set in Ke Jiang's tool, we add support for multidimensional arrays and for C $scanf$ for data inputs.

In order to support multidimensional arrays, we simulate an n-dimensional array as a 1-dimensional array. For this, we have to simulate two things: the capacity and the indexes.

**Capacity:** The resulting 1-dimensional array must have as much storage capacity as the n-dimensional array.This can be achieved by calculating the product of the n capacities in the n-dimensional array and assigning the result as the capacity of the 1-dimensional array.

**Array indexes:** We need a way of assigning each cell of the n-dimensional array to a unique cell in the1-dimensional array.

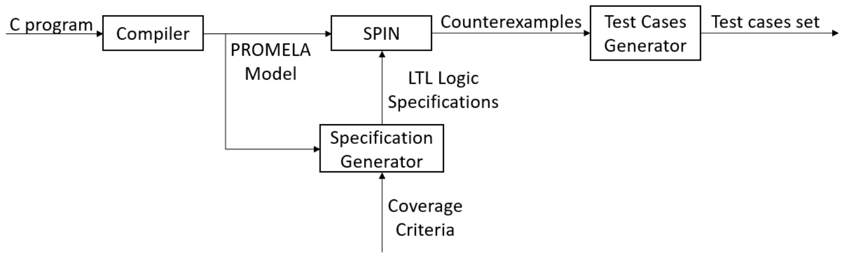


Fig. 3: General tool flow

For this, we concatenate each row of the n-dimensional array in a unique row and calculate the new indexes with equation (1):

$$f(n) = I_n + \sum_{i=1}^{n-1} (I_i \prod_{h=i+1}^{n} C_h) \tag{1}$$

where $n$ is the number of dimensions in the original C array, $I_i$ is the $i$-th index being accessed, and $C_h$ is the total capacity of the $h$-th dimension.

TABLE IV.3: Array translation example

| C code | PROMELA code |
|---|---|
| int a[4][7][5]; | int a[4*7*5]; |
| a[3][2][4] = 27; | a[4 + 3*7*5 + 2*5] = 27; |

Table IV.3 shows an example for a 3-dimensional array.

In order to support data input through $scanf$ function, we simulate data input by taking advantage of PROMELA asynchronous behaviour property for generating random values. This is achieved with the proctype shown in figure 4.

```
proctype input(chan in_input; int min; int max) {
    int value = min;
    do
      ::value < max -> value++;
      ::break;
    od;
    in_input ! value;
}
```

Fig. 4: Proctype for simulating program input.

## V. Automatic Test Cases Generation

Once obtained a PROMELA representation of the formal model of the program, we can use SPIN model checker to verify it with some properties describing a wished behaviour. This way we can use SPIN response for automatically generating test cases.

Given that we want the whole process to be completely automatic, then we need a way of automatically generating the properties to check. This is done by taking into account the wished coverage criteria, and generating a set of properties in order to cover this criteria the maximum possible.

For a better understanding, we will show this whole process through one studied example: the triangle problem. The code used for this problem can be found in [17]. One fragment of the translated PROMELA code, obtained in the first step, is shown in figure 5.

### A. *Control flow Oriented Coverage Criteria*

Control flow oriented coverage criteria covers cases related with program control flow. They are mainly based in truth values of program decisions and its conditions. So, we need to add atomic variables for each condition, so we can use them for creating the logic properties.

Let's look, for example, at the code fragment shown in figure 5. Here, we have an $if$ statement (a decision), with three conditions. So, we need to create the three atomic variables ($c1$, $c2$, and $c3$) shown on the top of the figure. In addition we create the $final$ atomic variable for representing when the execution reach the end of the program. Now, we can create LTL logic formulas in order to cover each of the wished coverage criterias for this group.

**Condition coverage:** For each decision $D$ and each condition $C$ of $D$, the test suite must include a test case where $C$ is evaluated $true$ and a test case where it's evaluated $false$.

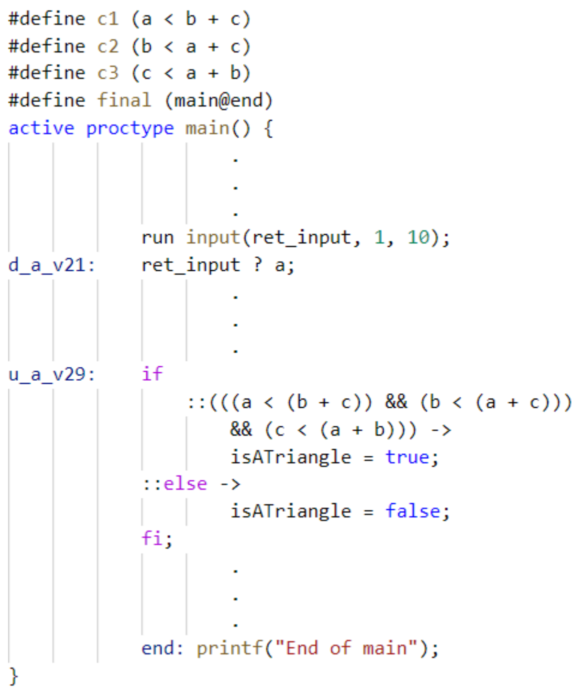
We need to create two LTL formulas for each condition, one for its $true$ value, and one for its $false$ value. Let's take, for example, condition $c1$ from figure 5 example. If we want a test case where it's evaluated $true$, we can use the following LTL formula:

$$\Box\neg(c1 \wedge \Diamond final)$$

This reads the program is never going to reach a state where $c1$ is true and eventually the program reach its end. So, when executing MC with the model and this property, SPIN will return a counterexample where the condition is true, and eventually the program reach its end, therefore providing a counterexample for the $true$ value of $c1$, which can be later use as a test case.

The same procedure can be used for its $false$ value. In this case we need a formula which reads the condition is never going to be $false$. This can be achieved with the following LTL formula:

$$\Box\neg(\neg c1 \wedge \Diamond final)$$

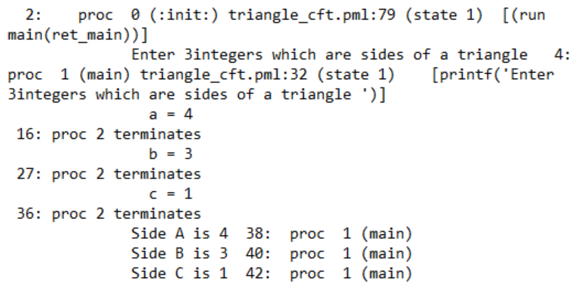


```
#define c1 (a < b + c)
#define c2 (b < a + c)
#define c3 (c < a + b)
#define final (main@end)
active proctype main() {
                          .
                          .
                          .
            run input(ret_input, 1, 10);
d_a_v21:    ret_input ? a;
                          .
                          .
                          .
u_a_v29:    if
                ::(((a < (b + c)) && (b < (a + c)))
                    && (c < (a + b))) ->
                    isATriangle = true;
            ::else ->
                    isATriangle = false;
            fi;
                          .
                          .
                          .
            end: printf("End of main");
}
```

Fig. 5: PROMELA $if$ example with multiple conditions

This way we obtained the two test cases we needed for achieving CC respect to condition $c1$. Now we only need to follow the same procedure for the rest of the conditions in order to obtain a complete coverage of this criteria.

One example of SPIN response for the property $\Box\neg(\neg c1 \wedge \Diamond final)$ is shown in figure 6.



```
  2:    proc  0 (:init:) triangle_cft.pml:79 (state 1)  [(run
main(ret_main))]
              Enter 3integers which are sides of a triangle   4:
proc  1 (main) triangle_cft.pml:32 (state 1)    [printf('Enter
3integers which are sides of a triangle ')]
                a = 4
 16: proc 2 terminates
                b = 3
 27: proc 2 terminates
                c = 1
 36: proc 2 terminates
              Side A is 4  38:  proc  1 (main)
              Side B is 3  40:  proc  1 (main)
              Side C is 1  42:  proc  1 (main)
```

Fig. 6: Fragment of SPIN response for figure 5 example and $\Box\neg(\neg c1 \wedge \Diamond final)$ property

Once we execute the same procedure for the rest of the conditions, and eliminate the test cases redundancy, we obtain the 6 test cases shown in Table V.1.

**Decision coverage:** For each decision $D$, the test suite must include a test case where $D$ is evaluated $true$ and a test case where $D$ is evaluated false.

In this case, we need two LTL formulas for each decision, one for its $true$ value, and one for its $false$ value. This can be achieved by following the same procedure we did for CC, but this time with the whole decision. So, for the previous example, we can use the following LTL formula for its true value:

$$\Box\neg((c1 \wedge c2 \wedge c3) \wedge \Diamond final)$$

and for its false value:

$$\Box\neg(\neg(c1 \wedge c2 \wedge c3) \wedge \Diamond final)$$

Once we execute the same procedure for the rest of the decisions, and eliminate the test cases redundancy, we obtain the 3 test cases shown in Table V.2.

| Case | a | b | c |
|---|---|---|---|
| 1 | 10 | 10 | 10 |
| 2 | 10 | 9 | 1 |
| 3 | 9 | 10 | 1 |
| 4 | 9 | 1 | 10 |
| 5 | 10 | 9 | 10 |
| 6 | 10 | 10 | 9 |

TABLE V.1: Test cases for CC criteria in the triangle problem

| Case | a | b | c |
|---|---|---|---|
| 1 | 10 | 10 | 10 |
| 2 | 10 | 9 | 1 |
| 3 | 10 | 9 | 8 |

TABLE V.2: Test cases for DC criteria in the triangle problem

**Modified condition/decision coverage:** For each decision $D$ and each condition $C$ of $D$, the test suit must contain two test cases where $C$ independently affects the value of $D$. This requires two test cases where we only change the value of $C$ while keeping the value of the remaining conditions, and the resulting value of $D$ is affected.

In order to achieve it, we modify the PROMELA model following the strategy proposed in [18], where a reset button is added to the model. The idea is to first get to the wanted decision, obtain its outcome, and then reset the program propagating the conditions and decision values, so we can get to the decision again but this time one of the conditions has a different truth value and the outcome of the decision is different. This way, instead of obtaining two different test cases for each condition, we obtain only one test case which can be later split into the two wished test cases.

The LTL formula for the previous example, with respect to $c1$ is:

$$\Box\neg(d \wedge \bigcirc(reset \wedge \Diamond(\neg d \wedge \neg inv\_c1 \wedge inv\_c2 \wedge inv\_c3)))$$

where $d = c1 \wedge c2 \wedge c3$ and $inv\_ci = true$ if the outcome of the condition $ci$ remains invariable.

Once we execute the same procedure for the rest of the decisions and conditions, and eliminate the test cases redundancy, we obtain the 6 test cases shown in Table V.3.

| Case | a | b | c |
|---|---|---|---|
| 1 | 10 | 10 | 10 |
| 2 | 10 | 9 | 1 |
| 3 | 9 | 10 | 1 |
| 4 | 9 | 1 | 10 |
| 5 | 10 | 10 | 9 |
| 6 | 10 | 9 | 9 |

TABLE V.3: Test cases for MCDC criteria in the triangle problem

### *B. Data Flow Oriented Coverage Criteria*

Data flow oriented coverage criteria covers the paths containing definitions and/or uses of program variables.

In order to understand this criteria we need to first introduce some basic concepts:

- **Definition node:** Node or sentence containing a definition of a variable (a value is assigned to it). With respect to a variable $x$ and a node $v$ we will use $d_x^v$ for saying there is a definition of $x$ in node $v$.
- **Use node:** Node or sentence where a variable is used. With respect to a variable $x$ and a node $v$ we will use $u_x^v$ for saying there is a use of $x$ in node $v$.
- **Definition-use path:** Path that goes from a definition node to a use node with respect to same variable.
- **Definition-clear path:** It's a definition-use path which does not contain any definition node other than the initial one.

For this group of criteria we will take the idea proposed in [19], by adding labels to the states (sentences in the PROMELA code, for our case), representing if there is a definition or use of a variable. This can be observed in figure 5, respect to variable $a$ with the labels $d\_a\_v21$ (there is a definition of $a$ in sentence 21) and $u\_a\_29$ (there is a use of $a$ in sentence 29).

Now, we need a way of defining a definition-clear path as an LTL formula, so we can later use it for defining the formulas for this criteria group. For this, we took the CTL logic formula proposed in [19] and created the equivalent LTL formula:

$$dcp(d_x^v, u_x^{v'}) = \Diamond(d_x^v \wedge \bigcirc(\neg def(x)\, U\, (u_x^{v'} \wedge \Diamond final))) \quad (2)$$

This reads, there is a definition-clear path respect to $x$, between nodes $v$ and $v'$ if eventually the execution reaches the definition of $x$ in node $v$ and in the next state there is no definitions of $x$ until the program reaches the use of $x$ in node $v'$ and eventually reaches the execution's end.

**All-defs criteria:** For each definition node respect to a variable $x$, the test suite must contain a test case with a definition clear path to at least one use of $x$.

For this criteria we need a test suit which contains all definition nodes and for each definition node, at least one definition-clear path. This can be achieved with the set of LTL formulas:

$$\{\bigvee_{u_x^{v'} \in USES(x)} dcp(d_x^v, u_x^{v'}) | d_x^v \in def(x)\}$$

**All-uses criteria:** For every definition node $d_x^v$ and every use node $u_x^{v'}$, the test suite must contain at least definition-clear path from $v$ to $v'$ respect to $x$.

This time we need a test case containing a definition-clear path, for each $dcp(d_x^v, u_x^{v'})$. This can be achieved with the set of LTL formulas:

$$\{\bigvee dcp(d_x^v, u_x^{v'}) | d_x^v \in def(x), u_x^{v'} \in USES(x)\}$$

For this example, with only one test case we can cover both of the criteria. The test case obtained by SPIN response is shown in Table V.4.

| Case | a | b | c |
|---|---|---|---|
| 1 | 10 | 10 | 10 |

TABLE V.4: Test case for all-defs and all-uses criteria in the triangle problem.

## VI. Conclusions

The whole process described in the previous sections was successfully completely automatized with the developed tool.

Our tool takes as input a C written program and first translates it into a PROMELA written model. Then, following the procedures described in section V, two versions of this first PROMELA model are created, one for the control flow coverage criteria group, and one for data flow coverage criteria group. In the next step, the tool automatically creates the needed LTL logic formulas sets for each of the analyzed coverage criteria and provides them one by one as input to SPIN, along with the corresponding PROMELA model. In the last step, each of the responses obtained from SPIN are stored in separated text files, so they can be later used as test cases for the original C program.

For future work we will consider adding support for more C characteristics in order to accept a wider set of programs.

## References

[1] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: A survey," Competence Network Softnet Austria, Austria, SNA Technical Report SNA-TR-2007-P2-04, 2007.

[2] J. Callahan, F. Schneider, and S. Easterbrook, "Automated software testing using model-checking," in *Proceedings 1996 SPIN Workshop*, vol. 353, 1996.

[3] A. Engels, L. Feijs, and S. Mauw, "Test generation for intelligent networks using model.checking," in *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1217, 1997, pp. 384–398.

[4] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *Software Engineering—ESEC/FSE'99*, vol. 1687, 1999, pp. 146–162.

[5] C. L. Heitmeyer, "Software cost reduction," *Encyclopedia of software engineering*, vol. 2, 2002.

[6] M. P. E. Heimdahl, S. Rayadurgam, and W. Visser, "Specification centered testing," in *Proceedings of the Second International Workshop on Automates Program Analysis, Testing and Verification*, 2000.

[7] H. S. Hong and I. Lee, “Automatic test generation from specifications for control-flow and data-flow coverage criteria,” in *Proceedings of the International Conference on Software Engineering*, 2003.

[8] A.Gargantini, E. Riccobene, and S. Rinzivillo, “Using spin to generate tests from asm specifications,” in *International Workshop on Abstract State Machines*, 2003, pp. 263–277.

[9] G. J. Holzmann and M. H. Smith, “Software model checking: Extracting verification models from source code,” *Software Testing, Verification and Reliability*, vol. 11, 2001.

[10] K. Havelund and T. Pressburger, “Model checking java programs using java pathfinder,” *International Journal of Software Tools for Technology Transfer*, vol. 2, 2000.

[11] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng, “Bandera: Extracting finite-state models from java source code,” in *Proceedings of the 2000 International Conference on Software Engineering*, 2000, pp. 439–448.

[12] K. Jiang, “Model checking c programs by translating c to promela,” M. Eng. thesis, Uppsala University, Sweden, September 2009.

[13] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Elsevier, 2010.

[14] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*. Springer, 2018.

[15] M. Ben-Ari, *Principles of the Spin model checker*. Springer Science & Business Media, 2008.

[16] A. Ghuloum, “An incremental approach to compiler construction,” in *Proceedings of the 2006 Scheme and Functional Programming Workshop*, 2006.

[17] R. W. andYuqiang Luo, J. Dong, and a. X. Q. Shuai Liu, “A heuristic algorithm for solving triangle packing problem,” *Discrete Dynamics in Nature and Society*, vol. 2013.

[18] S. Rayadurgam and M. Heimdahl, “Generating mc/dc adequate test sequences through model checking,” 2003.

[19] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural, “Data flow testing as model checking,” in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 232–242.