Model Checking MSVL Programs Based on Dynamic Symbolic Execution

Zhenhua Duan, Kangkang Bu, Cong Tian(✉), and Nan Zhang

Institute of Computing Theory and Technology, and ISN Laboratory,
Xidian University, Xi'an 710071, China
ctian@mail.xidian.edu.cn

Abstract. In this paper, we propose a DSE based model checking approach (DSE-MC) for verifying programs written in Modelling, Simulation and Verification Language (MSVL) [1,3]. For doing so, we adopt a DSE method to execute an MSVL program to generate a symbolic execution tree (SEtree) which is used as the abstract model of the program. Further, a property to be verified is specified by a Propositional Projection Temporal Logic (PPTL) formula [8,13]. To check whether or not the program satisfies the property, first the SEtree and the negation of the property are both described in Labelled Normal Form Graphs (LNFGs) [21], then the product of two LNFGs is produced. As a result, a counter example is encountered if the product is not empty. Otherwise, we cannot determine if the program satisfies the property. In this case, the verification process could be restarted with new inputs. In this way, a software system written in C can also be verified since the C program can be transformed to an MSVL program automatically by using toolkit MSV [19] developed by us.

Keywords: DSE · Constraints · Temporal logic · MSVL · Model checking

1 Introduction

Symbolic execution (SE) is first proposed for program testing [2] that uses symbolic values as inputs instead of real data. It represents the values of program variables as symbolic expressions which are functions of the input symbolic values. Subsequently, SE has been used for bugs finding [4] and verification condition generation [5,6], among others. Dynamic symbolic execution (DSE) [7,9] enhances traditional symbolic execution [2] by combing concrete execution and symbolic execution together. Essentially, DSE repeatedly runs a program both concretely and symbolically. After each run, all the branches off the execution path are collected, and then one of them is selected to generate new inputs for the next run to explore a new path. A symbolic execution tree (SEtree) depicts

The research is supported by the NSFC under Grant No. 61133001, 61322202, 61420106004 and 91418201.

all executed paths during the symbolic execution. A path condition is maintained for each path and it is a formula over the symbolic inputs built by accumulating constraints satisfied by those inputs in order for execution to follow that path. A path is infeasible if its path condition is unsatisfiable. Otherwise, the path is feasible.

Recently, symbolic execution has been used for software verification [10–12] as an alternative to the existing model checking techniques based on Counter Example Guided Abstraction Refinement (CEGAR) [14,15] method. Essentially, the general technique followed by symbolic execution-like tools starts with the concrete model of the program and then, the model is checked for the desired property via symbolic execution by proving that all paths to certain error nodes are infeasible (i.e., error nodes are unreachable). For instance, when DSE is applied to checking a program against a regular property specified by an automaton [16] , an execution path satisfies the property if the sequence of the events in the path is accepted by the automaton, and this path is an accepted path. DSE improves the coverage through symbolic execution, and avoids false alarms by actually running the program. More importantly, DSE can use the information of the concrete execution to simplify complex symbolic reasonings and handle the environment problem. However, the disadvantages of DSE based verification are obvious. The first challenge is the exponential number of symbolic paths. The approaches of [10–12] tackle successfully this fundamental problem by eliminating from the concrete model those facts which are irrelevant or too-specific for proving the unreachability of the error nodes. However, the problem is still remained. The second problem is that different notations are used to specify programs, symbolic execution trees and properties of programs to be verified. For instance, usually, a program is written in C or Java language, an SEtree is described by a graph and a property of the program is specified by an automaton [16] or temporal logic formula [17]. This makes the verification much complicated. The third problem is that some probes need to be inserted in the context of the program, and they are normally done manually.

In this paper, we are motivated to propose a DSE based model checking approach (DSE-MC) for verifying programs written in Modelling, Simulation and Verification Language (MSVL) [1,3]. For doing so, we adopt a DSE method to execute an MSVL program to generate an SEtree which is later used as the abstract model of the program. Further, a property to be verified is specified by a Propositional Projection Temporal Logic (PPTL) formula [8,13]. To check whether or not the program satisfies the property, first the SEtree and the negation of the property are both described in Label Normal Form Graphs (LNFGs) [21], then the product of two LNFGs is produced. As a result, a counter example is encountered if the product is not empty. Otherwise, we do not know if the program satisfies the property. In this case, the verification process could be restarted with new inputs. This means that the proposed model checking approach is incomplete. In this aspect, it is some what similar to Bounded Model Checking (BMC) [18]. In this way, a software system written in C can also be verified since the C program can be transformed to an MSVL program automatically by toolkit MSV [19] developed by us. However, DSE-MC is an abstract model checking and is suitable for verifying software systems. Comparing with CEGAR approach, the advantages of DSE-MC lie in four aspects: (1) the abstract model (i.e., symbolic execution tree) is automatically generated by means of dynamic symbolic execution of MSVL programs and SMT solver (i.e., Z_3); (2)we do not need to insert probes into the context of an MSVL program as other DSE methods do; (3) a program and a property of the program are both in the same logic system (PTL) [8]; (4) the speed of DSE-MC is relatively quicker than CEGAR method since an SEtree is smaller than an abstract model generated using CEGAR.

The paper is organized as follows: the next section describes the Dynamic Symbolic Execution of MSVL programs including DSE algorithm, extended MSVL interpreter, search strategy and constraint solving, and generating symbolic execution tree. The DSE based model checking approach is formalized in detail in section 3. In section 4, a case study is given to show how DSE based model checking works. Finally, conclusion is drawn in section 5.

2 Dynamic Symbolic Execution of MSVL Programs

Dynamic symbolic execution of MSVL programs is an improvement of symbolic execution where symbolic execution is performed simultaneously with concrete execution. DSE includes three main phases. The first phase is to execute programs symbolically and concretely to record the satisfied path constraints. The second phase is to select a constraint to negate from the path constraints according to the search strategy. The third phase is to solve the modified path constraints by using SMT solver Z_3 to generate new inputs and to execute the program with the new inputs. By repeating this process, almost all feasible execution paths are swept through.

2.1 Dynamic Symbolic Execution Algorithm

Essentially, to implement the dynamic symbolic execution of a program, the most frequently used method is the instrumentation-based approach. This approach instruments a program by inserting probes which perform symbolic execution to extract symbolic path constraints, then executes the instrumented code on a standard compiler or an interpreter which is determined by the type of the programming language. With this approach, the instrumentation code performs symbolic execution, while the original code performs concrete execution, then, as a result, the whole program is executed both symbolically and concretely at the same time.

Unlike the instrumentation-based approach, the dynamic symbolic execution of MSVL programs does not require such instrumentation, instead, implements a non-standard interpreter of MSVL programs which extends the original MSVL interpreter[3]. We extends the standard, concrete execution semantics of MSVL programs with symbolic execution semantics. The symbolic information is stored in *symbolic memory map* $\boldsymbol{S}$ and *symbolic path constraints* $\boldsymbol{PCs}$, and is propagated as needed, during the symbolic execution. The *symbolic memory map* $\boldsymbol{S}$ is defined as a mapping from program variables to symbolic expressions which record the symbolic values of all variables that are handled symbolically. The *symbolic path constraints* $\boldsymbol{PCs} = < \boldsymbol{pc_1}, \boldsymbol{pc_2}, \cdots, \boldsymbol{pc_n} >$ are defined as a list of formulas over symbolic input values which record the symbolic path constraints the current execution satisfies. Further, we need to define *Input map* $\boldsymbol{I}$, which maps input variables to its initial values before the execution of the program. One advantage of our approach is that we can exploit all execution information at runtime, since the MSVL Interpreter possesses all necessary information.

We now give the algorithm DSE-MC for dynamic symbolic executing MSVL programs. The basic idea of this algorithm is similar to the traditional one. Given an MSVL program P as input, we first initialize I, S and PCs to be $\emptyset$. Next, P is executed with input I by the extended MSVL Interpreter both symbolically and concretely. During the execution, the symbolic values of program variables and the path constraints along the path of the execution are collected and stored in S and PCs respectively. Once an MSVL program is executed, a symbolic path is generated and added to the SEtree of P subsequently. The symbolic path represents the models of P generated by the inputs which satisfy the path constraints of the current execution. Then, we select one constraint from PCs and negate the selected constraint, and solve the modified path constraints, to generate further inputs which possibly direct the program along alternative paths. The selection of the constraints to be negated depends on a search strategy. Finally, we execute P with the new inputs repeatedly until the termination conditions are met. Actually, if all the feasible paths are executed or the pre-defined bound is reached, the termination conditions are met. The pseudo-code for DSE-MC is shown in Algorithm 1.

Algorithm 1. Dynamic Symbolic Execution of MSVL Programs

```
Function DSE-MC($P$)
/* Input: $P$ is an MSVL program*/
/* Output: The Symbolic Execution Tree of $P$ */
begin function
  1:    $I = \emptyset$; $S = \emptyset$; $PCs = \emptyset$;
  2:    while termination conditions are not met
  3:        $SymbolicPath$=EXTMSVLINTERPRETER ($P$, $I$, $S$, $PCs$);
  4:        ADDTOSET ($SymbolicPath$);
  5:        $i$ = pick a constraint from $PCs$ ;
  6:        $I$ = SOLVE ($pc_1$,$pc_2$,$\cdots$, $\neg pc_i$ );
  7:    end while
end function
```

Extended MSVL Interpreter: The MSVL programs are executed only concretely by the original MSVL Interpreter. To execute MSVL programs both concretely and symbolically, we extend the MSVL Interpreter to implement the symbolic execution semantics. There are three main extensions of the MSVL Interpreter.

Algorithm 2. Dynamic Symbolic Execution of MSVL Programs using BDFS

```
Function DSE-MC-BDFS($P$, $max\_depth$)
/* Input: $P$ is an MSVL program, $max\_depth$ is the max execution times of $P$ */
/* Output: The Symbolic Execution Tree of $P$ */
begin function
  1:     $I = \emptyset$; $S = \emptyset$;$PCs = \emptyset$;
  2:     $SymbolicPath$=ExtMSVLInterpreter ($P$, $I$, $S$, $PCs$);
  3:     AddToSET ($SymbolicPath$);
  4:     BDFS (0, $max\_depth$, $PCs$, $P$, $I$, $S$);
end function
function BDFS($pos$, $depth$, $PCs$, $P$, $I$, $S$)
/* Input:$pos$ is the index of the path constraint picked to negate, $depth$ is the max execution times of $P$
          $PCs$ is the path constraints collected in previous execution
          $P$ is a MSVL program, $I$ is the input of $P$, $S$ is the symbolic memory map */
begin function
  1:     if $depth \geq 0$ do
  2:         for $i = pos$ to $PCs.size()$ do
  3:             $I$ = Solve($pc_1$,$pc_2$,$\cdots$, $\neg pc_i$);
  4:             $SymbolicPath$ =ExtMSVLInterpreter ($P$, $I$, $S$, $PCs$);
  5:             AddToSET ($SymbolicPath$);
  6:             $depth - -$;
  7:             BDFS ($i + 1$, $depth$, $PCs$, $P$, $I$, $S$);
  8:         end for
  9:     end if
end function
```

An MSVL program can be written as a logically equivalent conjunction of two programs *Present* and *Remains* (i.e., normal form) by the reduction process. *Present* part consists of immediate assignments to program variables, output of program variables, true, false or empty, and is executed at the current state. *Remains* part is what is executed in the subsequent state. In other words, any MSVL statements such as *Sequential statements*, *Projection statements*, *Parallel statements* and *while statements* can be reduced to their normal forms (NFs). In particular, the *while statements* can be transformed to *if statement* according to its definition $while\ b\ do\ p \overset{\text{def}}{=} (p \wedge q)^* \wedge \Box(empty \rightarrow \neg b)$. Therefore, to execute an MSVL program, we first transform it to its NF, then to interpret the *present* part, and further to execute its next part (*Remains*) recursively. In the extended interpreter, we focus on the dynamic symbolic execution as follows.

(1) For each input statement such as *input() and empty*, the symbolic memory map $\boldsymbol{S}$ introduces a mapping $x \longmapsto s_x$ from variable x to a fresh symbolic constant s_x which is regarded as the symbolic value of variable x. When the MSVL program is first executed, the input map $\boldsymbol{I}$ is $\emptyset$. Thus, we need to generate the inputs randomly. When the MSVL program is executed again, the inputs are generated by SMT solver Z_3.

(2) For each assignment statement such as $x <== e$ where e is an arithmetic expression that contains symbolic variables, the symbolic memory map $\boldsymbol{S}$ updates the mapping of x to $\boldsymbol{S}(e)$ which is obtained by evaluating e in the current symbolic memory map. For example, suppose that the symbolic memory map is $S = [x \longmapsto s_x, y \longmapsto s_y, z \longmapsto s_z]$, after the symbolic execution of $x <== 2*y+z$, the symbolic memory map becomes $S = [x \longmapsto 2*s_y+s_z, y \longmapsto$

$s_y, z \longmapsto s_z]$. For each assignment statement such as $x <== e$ where e is an arithmetic expression that contains no symbolic variables, the symbolic memory map is not updated, the statements are just executed concretely.

(3) For each conditional statement such as *if b then p else q*, if the concrete execution takes the *then branch*, the symbolic constraint $\boldsymbol{S}(b)$ is appended to $\boldsymbol{PCs}$. Otherwise, the symbolic constraint $\boldsymbol{S}(\neg b)$ is added.

Formally, we can construct a function called $ExtMSVLInterpreter$ (P, I, S, PCs) to fulfil the above functions. In fact, this function is the program that implements the new interpreter. It is omitted here.

Search Strategy and Constraint Solving: To execute all the paths in MSVL programs, we implement a commonly used bounded-depth-first(BDFS) strategy. In the BDFS, each run is executed based on path constraints collected in the previous run. The pseudo-code implementing BDFS strategy is shown in Algorithm 2.

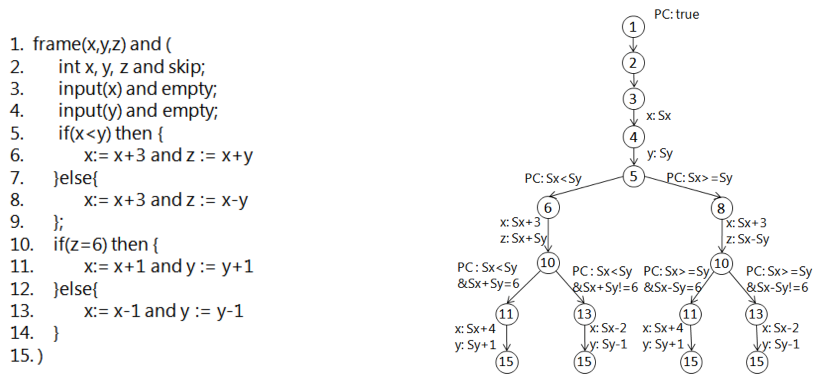
The procedure Solve in Algorithm 2 transforms the new path constraints into the format that SMT solver Z_3 can deal with and then calls Z_3 to produce a solution that satisfies the new path constraints as the new inputs. If the new path constraints are unsatisfiable, another constraint is negated until satisfiable path constraints are found. We choose Z_3 as our constraint solver, because it provides more extensive supports to solve linear arithmetic and non-linear arithmetic constraints.

Symbolic Execution Tree: The SEtree characterizes the execution paths of a program and is generated at the end of DSE. In general, in an SEtree, a node represents the statements executed, labeled with the statement number, and a directed arc represents a transfer of control between statement executions, labeled with the changes of the *symbolic memory map* $\boldsymbol{S}$ and *symbolic path constraints* $\boldsymbol{PCs}$, if any, caused by the execution of the preceding statement. The traditional SEtree is modified to be the system model of MSVL programs. On one hand, each node is changed to specify a program in MSVL. On the other hand, each directed arc is changed to specify a state formula. The modified SEtree can be regarded as the LNFG of the MSVL program. For example, the corresponding traditional and modified SEtrees of the MSVL program in Fig.1(a) are shown in Fig.1(b) and Fig.2(d) respectively.

2.2 An Example

We use a simple example to illustrate how an MSVL program performs DSE to generate its SEtree. Consider the MSVL program P shown in Fig. 1(a). In P, there are three variables that can be handled symbolically and four paths introduced by the two *if statements* in lines 5-9 and lines 10-14. According to the values of the three variables and the relationships among them, P runs following different paths. To construct the SEtree of P, we need to execute it at least four times to explore all the existing four paths in P.

There are two points we should pay attention to when executing P symbolically. The first is that input variables x and y are given symbolic values s_x and



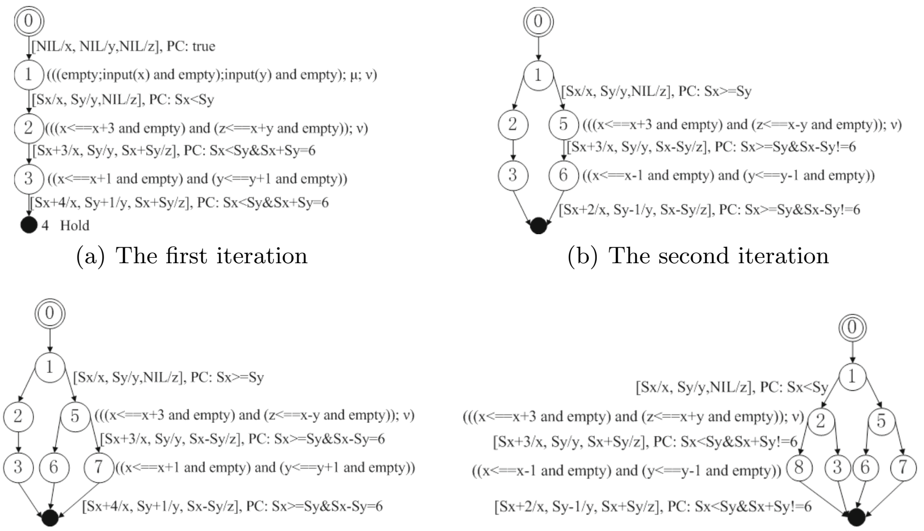
```
1.  frame(x,y,z) and (
2.      int x, y, z and skip;
3.      input(x) and empty;
4.      input(y) and empty;
5.      if(x<y) then {
6.          x:= x+3 and z := x+y
7.      }else{
8.          x:= x+3 and z := x-y
9.      };
10.     if(z=6) then {
11.         x:= x+1 and y := y+1
12.     }else{
13.         x:= x-1 and y := y-1
14.     }
15. )
```

(a) An MSVL Program P (b) Traditional SEtree of P

Fig. 1. An MSVL program and its traditional SEtree

s_y respectively when the input statements in line 3 and line 4 are executed. The second is that the ***PCs*** are initialized to ∅ before the execution, and updated after the execution of *if statements* in lines 5-9 and lines 10-14.

Now we describe the dynamic symbolic execution of P. In the first iteration of the DSE, it is assumed that the initial input is $\boldsymbol{I_1 = (x = 1, y = 5)}$ which is generated randomly. Executing P both concretely and symbolically by the extended MSVL Interpreter, the path this execution followed is recorded in *symbolic path constraints* ***PCs***, the details of the execution such as the changes of symbolic memory map ***S*** and ***PCs*** are shown in Fig. 2(a), where μ and ν in the figure represent statements *if(x < y) then* {*x:=x+3 and z:=x+y*}*else*{*x:=x+3 and z:=x-y*} and *if(z=6) then* {*x:= x+1 and y := y+1*}*else*{*x:= x-1 and y:=y-1*} respectively. $[s_x/x, s_y/y, NIL/z]$ in Fig. 2 means the values of variables x, y and z are s_x, s_y and NIL respectively. The symbolic path constraints collected in the first iteration are $\boldsymbol{PCs\_1 =< s_x < s_y, s_x + s_y = 6 >}$, which mean P takes the *then branch* of the first and the second *if statement*. With BDFS search strategy, we negate a constraint in the current execution and remove the subsequent constraints to obtain a modified symbolic path constraints $PCs\_1' =< s_x \geq s_y >$. Next, we solve PCs_1' to obtain input I_2 that drives the next execution along an alternative path. Suppose that the new input generated by Z_3 is $\boldsymbol{I_2 = (x = 3, y = 2)}$. Then, in the second iteration, P takes the *else branch* of the first and the second *if statement*. The path $< 0, 1, 5, 6, 4 >$ in Fig. 2(b) is the path this execution followed. For this execution, the path constraints $\boldsymbol{PCs\_2 =< s_x \geq s_y, s_x - s_y \neq 6 >}$ are gathered. We next solve the $PCs\_2' =< s_x \geq s_y, s_x - s_y = 6 >$, obtained by negating the last constraint and generate I_3 for the third execution. The third and fourth iterations that run with inputs $\boldsymbol{I_3 = (x = 8, y = 2)}$ and $\boldsymbol{I_4 = (x = 1, y = 2)}$ are similar to the second iteration. The paths $< 0, 1, 5, 7, 4 >$ in Fig. 2(c) and $< 0, 1, 2, 8, 4 >$ in Fig. 2(d) are the paths the third and fourth execution take respectively. After the


0
[NIL/x, NIL/y,NIL/z], PC: true
1 (((empty;input(x) and empty);input(y) and empty); μ; ν)
[Sx/x, Sy/y,NIL/z], PC: Sx<Sy
2 (((x<==x+3 and empty) and (z<==x+y and empty)); ν)
[Sx+3/x, Sy/y, Sx+Sy/z], PC: Sx<Sy&Sx+Sy=6
3 ((x<==x+1 and empty) and (y<==y+1 and empty))
[Sx+4/x, Sy+1/y, Sx+Sy/z], PC: Sx<Sy&Sx+Sy=6
4 Hold
[Sx/x, Sy/y,NIL/z], PC: Sx>=Sy
5 (((x<==x+3 and empty) and (z<==x-y and empty)); ν)
[Sx+3/x, Sy/y, Sx-Sy/z], PC: Sx>=Sy&Sx-Sy!=6
6 ((x<==x-1 and empty) and (y<==y-1 and empty))
[Sx+2/x, Sy-1/y, Sx-Sy/z], PC: Sx>=Sy&Sx-Sy!=6
[Sx+3/x, Sy/y, Sx-Sy/z], PC: Sx>=Sy&Sx-Sy=6
7 ((x<==x+1 and empty) and (y<==y+1 and empty))
[Sx+4/x, Sy+1/y, Sx-Sy/z], PC: Sx>=Sy&Sx-Sy=6
[Sx+3/x, Sy/y, Sx+Sy/z], PC: Sx<Sy&Sx+Sy!=6
8 ((x<==x-1 and empty) and (y<==y-1 and empty))
[Sx+2/x, Sy-1/y, Sx+Sy/z], PC: Sx<Sy&Sx+Sy!=6

(a) The first iteration (b) The second iteration

(c) The third iteration (d) The fourth iteration

Fig. 2. Dynamic Symbolic Execution of MSVL Program P

fourth execution of P, the dynamic symbolic execution algorithm is terminated because all the four paths have been explored. At last, the whole SEtree of P is constructed as shown in Fig. 2(d).

3 Model Checking MSVL Programs

The DSE based model checking approach is formalized in detail in this section. This approach can be divided into three major parts:the system model, the property and the model checking algorithm.

3.1 System Model

A path of a program execution is a model of the program, the set of all the paths of program executions is called all the models of the program, or *program models* for short. A path in an SEtree is an abstract path with constraints, which is the representative of all the paths satisfying the constraints. In other words, each path in the SEtree represents an equivalence class of paths (or inputs). Thus, the SEtree can be regarded as an abstract program model of an MSVL program.

With the method of program verification based on model checking, a system model means all the models of the program to be verified. Thus, the correctness of program verification can be guaranteed only if the SEtree completely covers the program models. However, we may not obtain complete abstract models in some cases. To combat the problem, we should increase the coverage degree of the program models as much as possible. Here we give a formal definition of the coverage degree, if $A \subsetneqq B \subsetneqq C$ where A, B and C are sets, we say, C includes B and A, B includes A. On the contrary, A and B cover part of C, and further B has a higher coverage degree over C than A.

3.2 Property

Propositional projection temporal logic (PPTL) [8,13] with the full regular expressive power is employed as the property specification language in our offline model checking approach. Thus, any MSVL programs with regular properties can be automatically verified with our offline model checking approach.

Let *Prop* be a countable set of atomic propositions and $B = \{true, false\}$ the boolean domain. Usually, we use small letters, possibly with subscripts, like *p,q,r* to denote atomic propositions and capital letters, possibly with subscripts, like *P, Q, R* to represent general PPTL formulas. Then the formula P of PPTL is defined by the following grammar:

$$P ::= p \mid \neg P \mid P_1 \wedge P_2 \mid \bigcirc P \mid (P_1, \ldots, P_m)\,\mathsf{prj}\,P \mid P^+$$

where $p \in Prop$, $\bigcirc$ (next), $+$ (chop-plus) and prj (projection) are temporal operators, and $\neg, \wedge$ are similar as that in the classical propositional logic. We define a *state* s over *Prop* to be a mapping from *Prop* to$B$, $s : Prop \rightarrow B$. We use $s[p]$ to denote the valuation of p at state s. Intervals and interpretations can be defined in the same way as in the first order case PTL [8]. The definitions for NFs and NFGs as well as LNFGs of PPTL formulas are the same as that presented in [21]. If the property to be verified is specified by a PPTL formula P, we can construct the LNFG of $\neg P$ for the subsequent verification according to its normal form [21].

3.3 Model Checking Algorithm

With our offline model checking algorithm for an MSVL program, the MSVL program M is modeled as its SEtree A_m, and property to be verified is specified by a PPTL formula P. To check if the MSVL program satisfying P is valid, we first obtain SEtree A_m by the dynamic symbolic execution of M. Then, we transform $\neg P$ to an LNFG A_φ, the transformation algorithm is given in [21]. Finally, we calculate the product of A_m and A_φ. If the product LNFG is not empty then a counter example is found, otherwise we cannot determine if the MSVL program satisfying the property is valid. In this case, the verification process could be restarted with new inputs. The algorithm is shown in Fig. 3.

Since model A_m and property P of a system are both described in the same logic framework PTL, it enables us to improve the efficiency of verification because of the avoidance of some transformations. However, there exists a problem that we should pay attention to, that is, it cannot be guaranteed that the SEtree we obtain covers all the models of an MSVL program. If we find a counter example during verification, the MSVL program satisfying the property is not valid. However, if there exist no counter examples, whether or not the

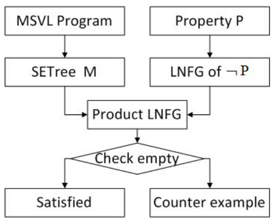


Fig. 3. Offline Model Checking MSVL

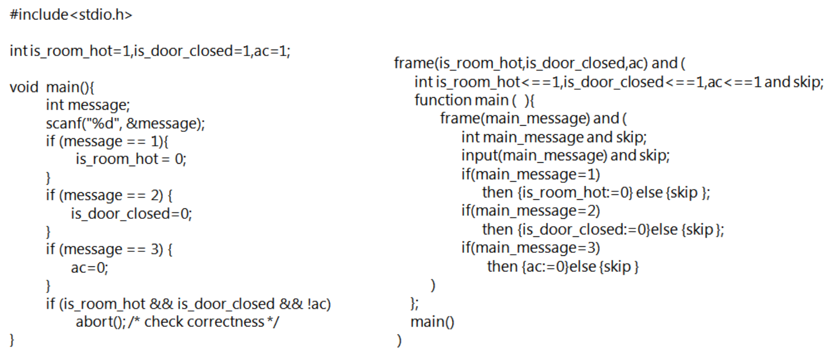
MSVL program satisfies the property cannot be determined. Thus, we should improve the coverage degree of the program models in the corresponding SEtree to improve the completeness of DSE-MC approach.

4 A Case Study

In this section, a typical example is presented to illustrate how the offline model checking approach can be utilized in the verification of real-world programs.

4.1 AC-controller Example

We use the AC-controller example presented in [7], which is used to compare the efficiency of traditional testing approach with the testing approach based on dynamic symbolic execution. Fig. 4(a) shows the AC-controller example simulating a controller for an air-conditioning system which is implemented by C. Initially, the room is hot and the door is closed, so the ac_controller is on. We can send a message to control the system. Different messages have different meanings and operations. From the code, we can also find when *is_room_hot&&is_door_closed&&!ac* is true, the system will be collapsed.



```
#include<stdio.h>

int is_room_hot=1,is_door_closed=1,ac=1;

void main(){
      int message;
      scanf("%d", &message);
      if (message == 1){
           is_room_hot = 0;
      }
      if (message == 2) {
          is_door_closed=0;
      }
      if (message == 3) {
          ac=0;
      }
      if (is_room_hot && is_door_closed && !ac)
           abort(); /* check correctness */
}
```

(a) A C program

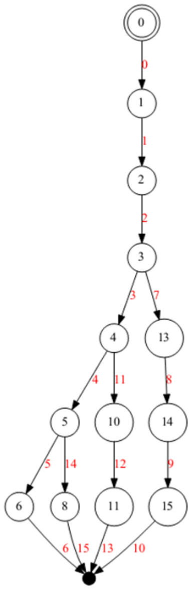
```
frame(is_room_hot,is_door_closed,ac) and (
   int is_room_hot<==1,is_door_closed<==1,ac<==1 and skip;
   function main ( ){
        frame(main_message) and (
            int main_message and skip;
            input(main_message) and skip;
            if(main_message=1)
               then {is_room_hot:=0} else {skip };
            if(main_message=2)
               then {is_door_closed:=0}else {skip };
            if(main_message=3)
                then {ac:=0}else {skip }
        )
   };
   main()
)
```

(b) A MSVL program

Fig. 4. AC- controller example

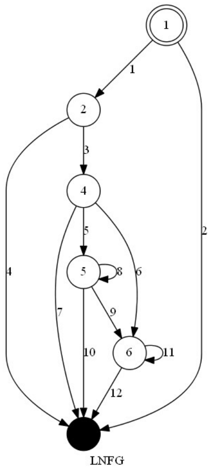
4.2 Verification

To verify AC-controller example, we first transform the original C program (remove the statement $if(is\_room\_hot \&\& is\_door\_closed \&\& !ac)abort();$) to an equivalent MSVL program as shown in Fig. 4(b) using toolkit MSV developed



Edge0:$[1/h, 1/d, 1/a, S_m/m]$, true
Edge1:$[1/h, 1/d, 1/a, S_m/m]$, true
Edge2:$[1/h, 1/d, 1/a, S_m/m]$, true
Edge3:$[1/h, 1/d, 1/a, S_m/m]$, $S_m \neq 1$
Edge4:$[1/h, 1/d, 1/a, S_m/m]$, $S_m \neq 1 \& S_m \neq 2$
Edge5:$[1/h, 1/d, 1/a, S_m/m]$, $S_m \neq 1 \& S_m \neq 2 \& S_m \neq 3$
Edge6:$[1/h, 1/d, 1/a, S_m/m]$, $S_m \neq 1 \& S_m \neq 2 \& S_m \neq 3$
Edge7:$[1/h, 1/d, 1/a, S_m/m]$, $S_m = 1$
Edge8:$[0/h, 1/d, 1/a, S_m/m]$, $S_m = 1 \& S_m \neq 2$
Edge9:$[0/h, 1/d, 1/a, S_m/m]$, $S_m = 1 \& S_m \neq 2 \& S_m \neq 3$
Edge10:$[0/h, 1/d, 1/a, S_m/m]$, $S_m = 1 \& S_m \neq 2 \& S_m \neq 3$
Edge11:$[1/h, 1/d, 1/a, S_m/m]$, $S_m \neq 1 \& S_m = 2$
Edge12:$[1/h, 0/d, 1/a, S_m/m]$, $S_m \neq 1 \& S_m = 2 \& S_m \neq 3$
Edge13:$[1/h, 0/d, 1/a, S_m/m]$, $S_m \neq 1 \& S_m = 2 \& S_m \neq 3$
Edge14:$[1/h, 1/d, 1/a, S_m/m]$, $S_m \neq 1 \& S_m \neq 2 \& S_m = 3$
Edge15:$[1/h, 1/d, 0/a, S_m/m]$, $S_m \neq 1 \& S_m \neq 2 \& S_m = 3$

Fig. 5. The symbolic execution tree of AC- controller example



Node ID	Corresponding PPTL formula	Label
Node1	!((len(2)) ; [](!(p) || !(q) || (r)))	
Node2	!(()EMPTY ; [](!(p) || !(q) || (r)))	
Node3	EMPTY	
Node4	<>((p) && (q) && !(r))	
Node5	(fin(l1)) ; (p) && (q) && !(r)	l1
Node6	TRUE	

Nodes of LNFG

Edge ID	state formula	->
Edge1	TRUE	Node1->Node2
Edge2	TRUE	Node1->Node3
Edge3	TRUE	Node2->Node4
Edge4	TRUE	Node2->Node3
Edge5	TRUE	Node4->Node5
Edge6	(q) && (p) && !(r)	Node4->Node6
Edge7	(q) && (p) && !(r)	Node4->Node3
Edge8	TRUE	Node5->Node5
Edge9	(q) && (p) && (l1) && !(r)	Node5->Node6
Edge10	(q) && (p) && (l1) && !(r)	Node5->Node3
Edge11	TRUE	Node6->Node6
Edge12	TRUE	Node6->Node3

Edges of LNFG

Fig. 6. The LNFG of the negation of Property

by us [19]. Then, we construct the SEtree of the MSVL program using Algorithm 2. The SEtree is shown in Fig. 5. In Fig. 5, letters h, d, a and m represent the variables is_room_hot, is_door_closed, ac and $message$ respectively, and s_m represents the symbolic value of $message$. Next, we specify the property to be verified by a PPTL formula. It is easy to find out that the property needs to be verified is "After a message is sent, AC-Controller should never be off when the room is hot and the door is closed". By employing propositions p, q and r to denote $is\_room\_hot = 1$, $is\_door\_closed = 1$ and $ac = 1$ respectively, this property can be specified by $P = len(2); \Box((p \wedge q) \rightarrow r)$ in PPTL. The LNFG of $\neg P$ is shown in Fig. 6. Finally, the product of the SEtree and the LNFG of $\neg P$ is calculated. We can find a counter example that violates P when the input value is 3. The path$< 0, 1, 2, 3, 4, 14, 15 >$ in Fig. 5 is the counter example.

5 Conclusion

We proposed a DSE based model checking approach for verifying MSVL programs. This approach can be used to verify software systems since C/Verilog programs can be transformed to MSVL programs by using toolkit MSV developed by us. Basically, the approach is based on abstract models because an SEtree can be viewed as an abstract model. The advantage is that the proposed method is a unified, automatical and quick approach. However, this method is an incomplete model checking approach like BMC in some sense because a counter example is the real one for the original program if the product of the SEtree and the LNFG of the negation of property is not empty. However, if there are no counter examples to be encountered we cannot determine if the original program satisfies the property. Therefore, in the future, we intend to further investigate how to improve the coverage degree of symbolic execution trees to the models of a program.

References

1. Ma, Q., Duan, Z., Zhang, N., Wang, X.: Verification of distributed systems with the axiomatic system of MSVL. Formal Aspects of Computing **27**(1), 103–131 (2015)
2. King, J.C.: Symbolic Execution and Program Testing. Journal of ACM, 385–394 (1976)
3. Ma, Y., Duan, Z., Wang, X., Yang, X.: An Interpreter for framed tempura and its application. In: Proceedings TASE 2007, pp. 251–260 (2007)
4. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. In: Proceedings of CCS 2006, pp. 322–335 (2006)
5. Beckert, B., Hahnle, R., Schmitt, P.H.: Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
6. Jacobs, B., Piessens, F.: The Verifast Program Verifier (2008)
7. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of PLDI 2005, pp. 213–223 (2010)
8. Duan, Z.: Temporal Logic and Temporal Logic Programming. Science Press, Beijing (2006)

9. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proceedings of ESEC FSE 2005, pp. 263C–272 (2005)
10. Jaffar, J., Santosa, A.E., Voicu, R.: An interpolation method for CLP traversal. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 454–469. Springer, Heidelberg (2009)
11. McMillan, K.L.: Lazy annotation for program testing and verification. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 104–118. Springer, Heidelberg (2010)
12. Harris, W.R., Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program analysis via satisfiability modulo path programs. In: Proceedings of POPL 2010, pp. 71–82 (2010)
13. Duan, Z.: An extended interval temporal logic and a framing technique for temporal logic programming, Ph.D. thesis, University of Newcastle upon Tyne (1996)
14. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: CounterrExample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
15. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proceedings of PLDI 2001, pp. 203–213 (2001)
16. Zhang, Y., Chen, Z., Wang, J., Dongy, W., Liu, Z.: Regular property guided dynamic symbolic execution. In: Proceedings of ICSE 2015 (2015)
17. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems-specification. Springer (1992). ISBN 978-3-540-97664-6
18. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Bounded model checking. Advances in computers **58**, 117–148 (2003)
19. Yang, K., Duan, Z., Tian, C.: Modeling and Verification of REC Handover Protocol. Electronic Notes Theoretical Computer Science **309**, 51–62 (2014)
20. Duan, Z., Tian, C.: A unified model checking approach with projection temporal logic. In: Proceedings of ICFEM 2008, pp. 167–186 (2008)
21. Duan, Z., Tian, C.: A practical decision procedure for Propositional Projection Temporal Logic with infinite models. Theoretical Computer Science **554**, 169–190 (2014)