Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/091L11

Reduction of Model Checking-based Test Generation using Satisfiability

Gongzheng Lu[1,2,*], *Huaikou Miao*[1] *and Honghao Gao*[1]

[1] School of Computer Engineering and Science, Shanghai University, 200072 Shanghai, China
[2] Department of Computer Engineering, Suzhou Vocational University, 215104 Suzhou, Jiangsu, China

Received: 12 Nov. 2013, Revised: 13 Mar. 2014, Accepted: 14 Mar. 2014
Published online: 1 Feb. 2015

Abstract: Constructing test cases from counterexamples generated by a model checker is an important method to perform test automation. In fact, one counterexample may cover multiple test goals, which leads to unnecessary calls to the model checker, and redundant test cases in test suite such that affect the testing performance. A method to test suite reduction based on satisfiability is proposed. The kripke model is translated in conjunction with test goals (trap properties) into CNFs. And then test goal to generate counterexample is selected according to the hardness of the corresponding CNF, after that, model checking the selected test goal to generate counterexample. The generated counterexample is translated in conjunction with those uncovered test goals into CNFs. If the corresponding CNF is unsatisfiable then the test goal is picked out from the set of test goals. Meanwhile, the new generated test case is winnowed by test suite to reduce the redundancy before it is added into the test suite. Experimental results show that the method proposed in this paper is effective for reducing the model checker calls and the length of the test suite. At the same time, the coverage and error detection capability of the test suite are not declined.

Keywords: Satisfiability, Model Checking, Test Suite Reduction

1 Introduction

Testing is an important and traditional software quality assurance technology. At present, test cases are generated by manually, which is lower efficiency, error-prone and not reusable such that total costs of software development increased dramatically. So testing automation is the inevitable trend. Recent years, model checking has been used to the automatic generation and optimization of test cases. However, model checking is originally an automatic verification technology for finite state model. If the verified property is not hold on the model, the model checker generates a counterexample which explains the reason why the property is violated. Whether test case can be constructed from the counterexample directly becomes the starting point of model checking can be used in testing.

Fraser et al. [1] indicated that there are several problems when using model checking to generate test cases. One of them is that executing a test case may consume some resources, and large numbers of test cases may affect the testing performance greatly, which requires a small test suite to satisfy the coverage criterion. Model checker is not dedicated for generating test cases. It generates one counterexample for each test goal (trap property). Generally, the same counterexample is generated several times for different properties. Similarly, a shorter counterexample may be subsumed by a longer. So it will lead to unnecessary calls to model checker to generate such counterexamples, and decrease the performance of model checking-based testing.

In this paper, we propose a method to reduce the test suite during test generation. The kripke model is translated in conjunction with test goals into CNFs. And then test goal to generate counterexample is selected according to the hardness of their corresponding CNF, after that, model checking the selected test goal to generate counterexample. The counterexample is translated in conjunction with those uncovered test goals into CNFs. If the corresponding CNF is unsatisfiable then the test goal is picked out from the set of test goals. Meanwhile, the new generated test case is winnowed by test suite to reduce the redundancy before it is added into the test suite. This paper is organized as follows: Section

[*] Corresponding author e-mail: tmks0863@sina.com.cn

2 introduces the related works, and then section 3 gives the background about our method. Section 4 shows how satisfiability can be used to reduce the test suite. The effects of this method are empirically analyzed in detail in section 5. Finally, it is the conclusions and future work.

2 Related Works

Many model checking-based test generation methods have been proposed. They can be fall into two categories: 1) Test goal is represented as trap property [2,3,4], and model checking the trap property, the counterexample generated by the model checker is constructed as test sequence which covers the test goal. 2) Model or property is mutated [5,6,7], the inconsistency between the model and the property will generate counterexample which is used to construct test suite satisfying the mutation adequacy criterion.

Test suite generated by model checking may have redundancy which affects the testing performance, so it is necessary to reduce the test suite. The reduction of test suite which uses a smaller test suite to cover the coverage criteria can be divided into reduction after test generation and reduction during test generation. Reduction after test generation denotes that generate test suite in terms of coverage criteria and then eliminate the redundant test cases from the test suite. Reduction during test generation indicates that generate test case for the selected test goal and check whether the test goals remaining in the set of test goals are covered by this test case to avoid to generating test cases for these test goals. Hamon et al. [8] extended test cases iteratively using model checker SAL, the number of the test cases in the resulting test suite is reduced, but the total length of the test suite may not be decreased, and they didn't indicate which test cases can be extended. Ammann et al. [9] represented the test case as model, and model checking the remaining test goals on the model to decide whether they are covered by the test case, it refers to the transition from test cases to models and calls to model checker frequently. Fraser et al. [10] used LTL rewriting to eliminate the test goals covered by existing test cases. Zeng et al. [11] used CTL rewriting to reduce the test goals and test suites. They all did not give in which order to select the test goals to generate test cases, however, the order of test goals selection will affect the effect of the reduction of test suite.

3 Background

Kripke structure [12] is generally used as formal model in model checking.

Definition 1 (Kripke Structure) A Kripke structure K is a tuple K=(S,S_0,T,L), where S is the set of states, $S_0 \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the total transition relation, and $L : S \to 2^{AP}$ is the labeling function maps each state to a set of atomic propositions that hold in this state. AP is the set of atomic propositions. If the model violates the property, model checker generates a counterexample to explain the trace violating the property. Such trace is the prefix of a path in the model.

Definition 2 (Path) A path of $\pi :=< s_0,\cdots,s_n >$ of a Kripke Structure K is a finite or infinite state sequence such that $\forall 0 \leq i < n : (s_i,s_{i+1}) \in T$ for K.

In this paper, the property is specified by temporal logic LTL [13].

The syntax of LTL is: ϕ::=$true|false|p|\neg\phi_1|\phi_1 \wedge \phi_2|\phi_1 \vee \phi_2|\phi_1 \to \phi_2|X\phi|F\phi|G\phi|\phi_1 U \phi_2$, where p is the atomic proposition, ϕ_1 andϕ_2 are LTL formulas, temporal operator X, F, G, U represent the next state, some state in future, all states in future, and until some state respectively.

The semantics of LTL is represented by infinite paths of a Kripke Structure. Model checker computes iteratively whether K satisfies ϕ on path π, denoted as $K,\pi \vDash \phi$. π_i is the suffix of the path π starting from the ith state, $\pi(i)$ denotes the ith state of the path π, where $i \in N$. $\pi(0)$ is the initial state of the path π.

Trap property(test goal) ϕ is the property assumed to be violated by the model and it is used to generate counterexample t such that $K,t \nvDash \phi$, where t is a path. For example, a test goal for state coverage criterion is a state a can finally be reached, the trap property is written as $G\neg(state = a)$. A counterexample to such trap property is any path that contains state a. The test case of test goal ϕ can be constructed directly from the state sequence corresponding to t. Generally, the state sequence corresponding to the counterexample is called a test case. The length of the test case is the number of the transitions in the state sequence. The set of test cases is called test suite, the length of the test suite is the total length of all test cases in the test suite.

4 Using Satisfiability to Reduce Test Suite

Traditionally, test cases are generated by model checking all the trap properties sequentially. It may lead to redundant test cases. And the performance of model checking-based test generation will be degraded because we still generate test cases for trap properties which have covered by existing test cases. Such situation can be avoided by checking whether the trap properties have been covered by existing test cases, so the redundant test cases can be reduced, meanwhile the model checker calls also be eliminated.

Bounded model checking [14] decides whether the model satisfies the property based on satisfiability. Different with the traditional model checking, bounded model checking does not search the state space, but translates the conjunction of the model with the negation of the property into a CNF, and solves the CNF using SAT solver. If it is unsatisfiable, then the model satisfies the property, else the satisfiability assignment of the variables is a counterexample of the property.

Satisfiability can also be used to reduce the model checking-based test generation. After a test case is generated for a trap property, the test case is translated in conjunction with the remaining trap properties into CNFs, and the trap properties covered by the test case is get according to the unsatisfiability of the corresponding CNFs, so it is unnecessary to call model checker to generate test cases for such trap properties.

The method to reduce the test suite using satisfiability is given as follows. It improves the performance of model checking-based test generation greatly.

4.1 Bounded Model Checking and Satisfiability

The main idea of bounded model checking is: the model is modeled as Kripke Structure and the property is specified by LTL formula, and the bound is set to k, then the conjunction of the model with the negation of the LTL formula constitutes BMC formula, the BMC formula is encoded to SAT instance (CNF), finally the CNF is solved by SAT solver. If the CNF is satisfiable, then a counterexample is generated, else it denotes that the model satisfies the property after k steps.

Let M be a Kripke Structure, f be the negation of a LTL formula, k be the bound. $[[M,f]]_k$ is a proposition. A path $\pi =< s_0,\cdots,s_k >$ is a finite state sequence of $[[M,f]]_k$. $[[M,f]]_k$ is satisfiable iff f holds on some path π.

Definition 3 (BMC formula) BMC formula is: $[[M,f]]_k = [[M]]_k \wedge [[f]]_k$.$[[M]]_k$=$I(s_0) \wedge \bigwedge_{l=0}^{k-1} T(s_l,s_{l+1})$, $[[f]]_k$=$(\neg L_k \wedge [[f]]_k^0 \vee \bigvee_{l=0}^{k}({}_lL_k \wedge {}_l[[f]]_k^0)$
${}_lL_k$=$T(s_k,s_l)$,L_k=$\bigvee_{l=0}^{k} {}_lL_k$.

The explanation of BMC formula can refer to [14].

Definition 4 (CNF) A CNF is $F = C_1 \vee \cdots \vee C_m$, where $C_1,\cdots,C_m$ are clauses, each clause has the form $C_i = l_1 \vee \cdots \vee l_n$ for $l_1,\cdots,l_n$ are literals, each literal is the positive or negative form of a boolean variable.

Most SAT tools use DPLL algorithm to solve CNF, the algorithm finds out whether exists some assignment of the boolean variables such that the CNF is true. If existing, F is called satisfiable, or F is called unsatisfiable if F is always false for any assignment of the boolean variables.

4.2 Reduction of the Test Suite

The order for selecting test goals to generate test case is important. Previous works selected the test goals randomly. In satisfiability theory, the shorter the clause is the harder the clause is to be solved. Based on this, we first translate $M \wedge \phi$ into CNFs, and then select the test goal according to the hardness of the CNFs.

Definition 5 (Hardness of CNF) Let the number of the clause in the CNF is n, the number of the literal in the ith clause is l_i. The hardness of CNF is h:
$h = \frac{\sum_{i=1}^{n} l_i}{n}$

For each trap property, we translate the conjunction of the model with it into CNF. Then the hardness of each CNF is computed, we assumed that the set of hardness is H. The test goals are ranked ascending according to the hardness of the CNF, and the resulting set of test goals is RTG. We select the test goal sequentially from RTG. The smaller h will generate the longer test case, so it avoids to generating redundant test cases.

The method to check whether the remaining test goals in the set of test goals are covered by existing test cases using satisfiability is based on the following theorem.

Theorem 1 Let $\pi :=< s_0,\cdots,s_t >$ be the test case for test goal $TG1$. Selecting an uncovered test goal $TG2$ from the set of test goals TG, if the CNF translated from $\pi \wedge G\neg TG2$ is unsatisfiable, then the test case π covers test goal $TG2$.

Because the CNF translated from $\pi \wedge G\neg TG2$ is unsatisfiable, that is $\pi \vDash G\neg TG2$. It can be seen that π is a counterexample of $G\neg TG2$, then π can be the test case covering test goal $TG2$.

Algorithm1 TestSuitReduction(M,TG,TS)

```
1 begin
2   $H$={};
3   for each test goal $\phi$ in $TG$ do
4     begin
5       $i$ = 1;
6       $f$ = GenerateCNF($M \wedge \phi$);
7       $h(i)$= ComputeHardness($f$);
8       $H$= $H \cup h(i)$;
9       $i = i+1$;
10    end
11    $RTG$ = RankAscending($TG,H$);
12    $TS$ ={ };
13    While $RTG$ != empty do
14      begin
15        Select a test goal $\phi$ from $RTG$;
16        $RTG = RTG - \{\phi\}$;
17        $\pi$= Model Checking($M,\phi$);
18        if $TS$ = { };
19        then $TS = TS \cup \{\pi\}$;
20        else $TS$=Winnow($TS,\pi$);
21        for each remain test goal $\varphi$ in $RTG$ do
22          begin
23            $f$ = GenerateCNF($\pi,\varphi$);
24            result = SAT($f$);
25            if result = unsatisfiable
26            then $RTG = RTG - \{\varphi\}$;
27          end
28      end
29 end
```

We select the first test goal from the set of test goals RTG; meanwhile delete the test goal from RTG. Model checking the test goal, and generate test case for the test goal, the test case is added to test suite TS. When the new test case is added to the test suite, it is winnowed by current test suite TS, and the redundant test cases will be eliminated. Then if this test case is not redundant, it is translated in conjunction with the each remaining test goal in the set of test goals into CNF. The CNFs are solved by SAT tool. If the CNF is unsatisfiable, then deleting the test goal in RTG, else solving the next CNF until the CNFs are all solved. After this, the test goals satisfied by the test case are deleted from RTG, and the model checker will not be called to generate test cases for them. We choose test goal sequentially from RTG, and repeat above procedure for each test goal until RTG is empty.

The algorithm of reduction of the test suite using satisfiability is given in algorithm 1.

Function GenerateCNF$(M \wedge \phi)$ is used to translate $M \wedge \phi$ into CNF. ComputeHardness(f) computes the hardness of CNF f. RankAscending(TG, H) sorts the set of the test goals according to the hardness of their corresponding CNFs, and the resulting set is the ranked test goals RTG. ModelChecking(M, ϕ) model checks the test goal ϕ on model M and generates a counterexample π. Winnow(TS, π) winnows test case π by test suite TS. The function Winnow(TS, π) is described in algorithm 2.

Algorithm 2 Winnow(TS, π)

```
1 begin
2   for each test case $\zeta$ in $TS$ do
3     begin
4       $j$ = MinLength$(\pi, \zeta)$;
5       for $i = 1$ to $j$ do
6         begin
7           if ! Match$(\pi(i), \zeta(i))$
8           then $TS = TS \cup \{\pi\}$;
9             exit;
10          else if $(i == j)$ and
11              $(Length(\pi) > Length(\zeta))$
12            then $TS = TS - \{\zeta\} \cup \{\pi\}$;
13        end
14    end
15 end
```

Function MinLength(π, ζ) is used to get the smaller length between the length of test case π and the length of ζ. Match$(\pi(i), \zeta(i))$ checks whether the ith state of π is the same as the ith state of ζ. If they are the same, Match$(\pi(i), \zeta(i))$ returns true. If some test case ζ in TS is subsumed by test case π, then test case ζ is deleted from TS and π is added to.

5 Experimental Study

In this paper, test cases are generated by model checker NuSMV 2.5.4, and the satisfiability of the CNFs are solved by Yices [15]. An example is used to explain how the satisfiability can be used to reduce the test suite generated by model checking. And our method is compared with the non-optimized model checking-based test generation method and LTL Rewriting method.

Test cases for navigation behavior of the simplified version of Student Grade Retrieve System (SGRS) are generated in terms of state coverage criterion, transition coverage criterion and transition pair coverage criterion respectively. The Kripke Structure of the navigation behavior is described in figure 1 [1]. The navigation begins at page *blank*, the home page *main* is consisted of two sub-frame pages *news* and *login*. Users can submit their login information from *login*, then page *studview* is returned for students or the page *adminview* is returned for system administrator. Administrator can maintain the system and retrieve the students' information. Students can look over their personal information from subpage *studinfo* in *studview* and inquire their grade from page *grade*. The page *gradelist* is generated dynamically according to the query conditions provided by *grade*. Page *studinfo* and *grade* can be visited through each other.

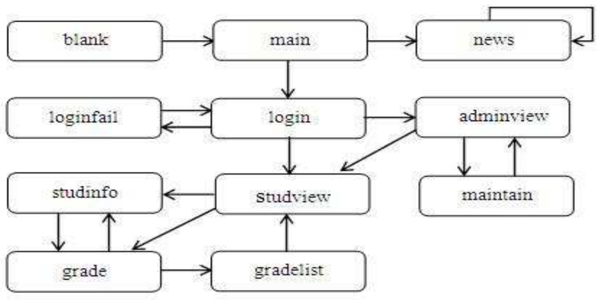

Fig. 1: Kripke Structure of SGRS

Our method first translates the conjunction of the system model M with each test goal ϕ into CNF. Then the hardness of the CNFs are computed, the set of ranked test goals RTG is get according to the hardness of the CNFs. If several CNFs have the same hardness, then we rank the test goals corresponding to these CNFs randomly. Finally, the test goal used to generate test case is chosen sequentially.

For state coverage criterion, the ranking of the hardness of CNFs is showed in table 1.

According to table 1, test goal $TG1$ is first chosen to model checking and generate test case π_1 : $(blank, main, login, studview, grade, gradelist)$. And then the test case π_1 is converted in conjunction with each test goal in $RTG - \{TG1\}$ into CNF. If the CNF is unsatisfiable, then it indicates that π_1 covers the test goal corresponding to the CNF. We delete the test goals covered by π_1 from RTG and get the new RTG. Next we chose the remaining test goals in RTG sequentially, repeat above procedure until RTG is empty.

Finally, we get 6 test cases: $\pi_1 : (blank, main, login, studview, grade, gradelist)$, $\pi_2 : (blank, main, login, adminview)$, $\pi_3 : (blank, main, news)$, $\pi_4 : (blank, main, login, studview, studinfo)$, $\pi_5 : (blank, main, login, adminview, maintain)$, $\pi_6 : (blank, main, login, loginfail)$. During the generation of the test cases, the test goals covered by these test cases are shown in table 2.

Table 1: Ranking of the hardness of CNFs: State Coverage Criterion

TG	L	C	L/C	R
$TG1$	2508	747	3.357429719	1
$TG2$	2401	705	3.405673759	3
$TG3$	2421	706	3.42917847	9
$TG4$	2403	704	3.413352273	5
$TG5$	2439	704	3.464488636	10
$TG6$	2547	704	3.617897727	11
$TG7$	2404	704	3.414772727	6
$TG8$	2404	704	3.414772727	7
$TG9$	2404	704	3.414772727	8
$TG10$	2402	704	3.411931818	4
$TG11$	2390	703	3.399715505	2

$TG : TestGoal\ L : Literals\ C : Clauses\ R : Ranking$
$TG1 : G!gradelist\ TG2 : G!grade\ TG3 : G!studinfo\ TG4 : G!studview$
$TG5 : G!maintain\ TG6 : G!loginfailTG7 : G!login\ TG8 : G!adminview$
$TG9 : G!news\ TG10 : G!main\ TG11 : G!blank$

Table 2: Test goals covered by test cases $\pi_1 \sim \pi_6$

TG	π_1	π_2	π_3	π_4	π_5	π_6
$TG1$	Y					
$TG2$	Y					
$TG3$				Y		
$TG4$	Y					
$TG5$					Y	
$TG6$						Y
$TG7$	Y					
$TG8$		Y				
$TG9$			Y			
$TG10$	Y					
$TG11$	Y					

After winnowing, π_2 is subsumed by π_5, so it is redundant, and the test suite is $TS = \{\pi_1, \pi_3, \pi_4, \pi_5, \pi_6\}$.

Similarly, for the 17 test goals of the transition coverage criterion, we get 7 test cases. And we get 13 test cases for the 29 test goals of the transition pair coverage criterion.

Table 3: Reduction: State Coverage Criterion

Method	MDC	TC	Length
Original	10	10	31
LTL Rewriting	8	5	18
Our Method	6	5	14

$MDC : NumberofModelCheckingcallsTC : NumberofTestCasesLength : LengthofTestSuite$

Our method is compared with the non-optimized model checking-based test generation method (Original) and LTL Rewriting method for these three coverage criteria in table 3∼ 5.

Table 4: Reduction: Transition Coverage Criterion

Method	MDC	TC	Length
Original	17	17	63
LTL Rewriting	12	7	35
Our Method	10	7	32

Table 5: Reduction: Transition pair Coverage Criterion

Method	MDC	TC	Length
Original	29	29	136
LTL Rewriting	22	13	73
Our Method	19	13	73

The original method calls the model checker once for each test goal, and generates a test case for each of them. So the number of model checking calls and the number of test cases both equal to the number of test goals. Our method and LTL Rewriting method both can eliminate the number of the model checking calls well, and reduce the number of test cases and the length of the test suite greatly. The number of test cases generated by our method is the same as the LTL Rewriting, but the number of model checking calls is smaller than LTL Rewriting and the length of the test suite is also shorter (or equal). The reason for our method can reduce more model checker calls than LTL Rewriting is: we select the test goal to generate counterexample according to the hardness of its corresponding CNF, while LTL Rewriting is randomly. The smaller the hardness of the corresponding CNF is, the harder the test goal is satisfiable, the longer the test case is and the more test goals are covered, so the less model checker calls are required. The reason why the length of the test suite is shorter than the LTL Rewriting is: our method use bounded model checking which generate the shortest counterexample, so the length of the test suite may become shorter.

Table 6: Coverage: State Coverage Criterion

Method	TG_S	TG_T	TG_{TP}
Original	100%	58.82%	31.03%
LTL Rewriting	100%	58.82%	31.03%
Our Method	100%	58.82%	31.03%
Our Method(LP)	100%	82.35%	58.62%

Table 7: Coverage: Transition Coverage Criterion

Method	TG_S	TG_T	TG_{TP}
Original	100%	100%	55.17%
LTL Rewriting	100%	100%	55.17%
Our Method	100%	100%	55.17%
Our Method(LP)	100%	100%	75.86%

Table 8: Mutation Scores: Model Mutants

Method	M_S	M_T	M_{TP}
Original	75%	77.68%	68.75%
LTL Rewriting	75%	77.68%	68.75%
Our Method	75%	77.68%	68.75%
Our Method(LP)	71.43%	73.21%	68.75%

In following, in order for presented to the method to be feasible, it is important that the coverage with regard to the criteria used for test generation is not affected. We analyze the coverage of the test suites generated by the original, LTL Rewriting and our method respectively. Table 6 and table 7 show the result of coverage analysis. There is one table for each set of test goals used for test case generation for each coverage criterion.

TG_S, TG_T and TG_{TP} are the set of test goals generated according to state coverage criterion, transition coverage criterion and transition pair coverage criterion respectively. In table 6, the test suite is generated for state coverage criterion, the coverage of the test goals TG_S is 100%. And in table 7, the test suite is generated for transition coverage criterion, the coverage of the test goals TG_T is 100%, and transition coverage criterion is stronger than state coverage criterion, so the coverage of TG_S is also 100%. Our Method (LP) represents that extend the loop in the test cases generated by our method once. The number of test cases is reduced by both our method and LTL Rewriting, but the coverage of the test goals for each criterion is not improved. Our Method (LP) increases the length of test suite a little bit, but the coverage of the test goals is increased obviously. Here, the coverage of the test suite generated for transition pair coverage criterion is not given. Transition pair coverage criterion is stronger than the other two criterions, the test suite satisfying the transition pair coverage criterion also satisfies them. So the coverage of the test suites for these methods is 100%.

Last, we study whether the error detection capability is affected while the number of test cases and length of test suite are reduced. Mutation score which is the percent of mutated models or properties detected by test cases can be as the measurement of the error detection capability. Models or properties can be mutated by mutant operations. In our experiment, we use model mutant and get 112 mutations. The results are in table 8.

M_S, M_T and M_{TP} are the mutation scores of the test suites generated with regard to the three coverage criterions. The error detection capability of our method is the same as the original method and LTL Rewriting while our method reduces more test cases and has shorter test suite. M_S and M_T of Our Method (LP) are declined a little. One possible explanation is: when we use the mutant operation substitution, many states add a transition to themselves. These mutations can be detected by the test cases generated by our method. While Our Method (LP) extends the loop in the test cases generated by our method once. Several states can reach themselves through multistep transitions. These multistep transitions are substituted by one self transition in mutant model. They can't be detected by Our Method (LP), so its mutation scores are declined.

The results of our experimental indicate that: our method can eliminate the number of model checking calls well, and can reduce the number of test cases and the length of test suite greatly. Our method has advantage over LTL Rewriting. At the same time, the coverage of the test goals and the error detection capability of our method are not declined.

6 Conclusions and future work

We propose an approach to reduce the model checking-based test generation using satisfiability. The experimental results show that: it has good reduction effectiveness, and can improve the performance of the test generation based on model checking, it also can reduce the expense required in the following test execution.

The features of our method are as follows:

1. Test goal used to generate test case is chosen according to the hardness of its corresponding CNF, which is better than the random method. It requires less model checking calls and reduces more test cases.

2. Test goals covered by the test case are checked by the unsatisfiability of the CNF which is the conjunction of the test case with each test goal. LTL Rewriting method uses LTL Rewriting tool to rewrite each LTL formula. The rewriting operation is performed in each state, which may have less efficiency when the test cases are long. While the satisfiability of the CNF is solved by SAT tool, which has efficient algorithms. Even for long test cases, the corresponding CNFs can be solved quickly.
3. Bounded model checking is used to generate test cases. Bounded model checking can generate the shortest counterexample, so the length of test suite generated by our method is shorter than the LTL Rewriting.
4. Our method has better reduction effectiveness, while the coverage of the test goals and the error detection capability are not declined.

Based on the work in this paper, our future work includes:

1. When we use bounded model checking to generate test cases, we will research how the bound k can affect the reduction effectiveness, the coverage of test goals and the error detection capability.
2. Using an industrial application to validate our method.
3. To implement a prototype tool.

Acknowledgement

This work is supported by the National Natural Science Foundation of China (NSFC) under grant No.60970007,No.61073050 and No.61170044; and the Shanghai Leading Academic Discipline Project of China under grant No.J50103.

References

[1] G. Fraser, F. Wotawa, and P. Ammann. Issues in using model checker for test case generation. The Journal of System and Software, 2009, **82**: 1403-1418.
[2] A. Gargantini, C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. ACM SIGSOFT Software Engineering Notes, 1999, **24**:146-162.
[3] M. P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj and J. Gao. Auto-Generating Test Sequences Using Model Checker: A Case Study. In: A. Petrenko and A.Ulrich eds. Proc. of the 3rd Int'l Workshop on Formal Approaches to Software Testing. Berlin: Springer-Verlag, 2003. 42-59.
[4] S. Rayadurgam, M. P. Heimdahl. Coverage based Test-Case Generation using Model Checkers. In: Proc. of the 8th Annual IEEE Int'l Conf. on Workshops on the Engineering of Computer Based Systems (ECBS 2001). IEEE Computer Society, 2001. 83-91.
[5] P. Ammann, W. Ding, D. Xu. Using a Model Checker to Test Safety Properties. In: Proc. of the 7th IEEE Int'l Conf. on Engineering of Complex Computer Systems (ICECCS 2001). IEEE Computer Society, 2001, 212-221.
[6] P. E. Ammann, P. E. Black, W. Majurski. Using Model Checking to Generate Tests from Specifications. In: Proc. of the 2nd Int'l Conf. on Formal Engineering Methods(ICEFM'98). IEEE Computer Society, 1998, 46-54.
[7] G. Fraser, F. Wotawa. Property relevant software testing with model-checkers. SIGSOFT Softw. Eng. Notes, 2006, **31**:1-10.
[8] G. Hamon, L. de Moura, and J. Rushby. Generating efficient test set with a model checker. In: Proc. of the 2nd Int'l Conf. on Software Engineering and Formal Methods(SEFM 2004). IEEE Computer Society, 2004: 261-270.
[9] P. E. Ammann, P. E. Black. A specification-based coverage metric to evaluate test set. In: Proc. of the 4th IEEE Int'l Symposium on High-assurance Systems Engineering (HASE'99). IEEE Computer Society, 1999: 239-248.
[10] G. Fraser, F. Wotawa. Using LTL Rewriting to Improve the Performance of Model Checker-Based Test Case Generation. In: Proc. of the 3rd Int'l Workshop on Advances in Model-Based Testing (AMOST 2007). ACM Press, 2007: 64-74.
[11] H. W. Zeng, H. K. Miao. Opitimization of Model Checking-Based Test Generation. Journal of Computer -Aided Design & Computer Graphics, 2011, **23**: 496-502.
[12] M. O. Markus, S. David, and S. Bernhard. Model Checking A Tutorial Introduction. In: A. Cortesi, G. Fil eds. Proc. of the 6th Int'l Symposium on Static Analysis (SAS'99). Berlin: Springer-Verlag, 1999: 330-354.
[13] H. Keijo, N . Ilkka. Bounded LTL Model Checking with Stable Models. In: T. Eiter, W. Faber and M. Truszczyski eds. Proc. of the 6th Int'l Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR 2001). Berlin: Springer-Verlag, 2001: 200-212.
[14] C. Edmund, B. Armin, R. Richard, and Y. S. Zhu. Bounded Model Checking Using Satisfiability Solving. Formal Methods in System Design, 2001, **19**: 7-34.
[15] B. Dutertre The Yices SMT Solver. http://yices.csl.sri.com/documentation.shtml

Gongzheng Lu received the Master degree in Computer Software and Theory from Soochow University, Suzhou, China, in 2006. Now study the Ph.D. of Computer Application Technology in Shanghai University from 2010. He is currently a lecturer in Suzhou Vocational University. His research interests include software testing and model checking.

Huaikou Miao received the Master degree in Computer Application Technology from Shanghai University of Science and Technology, Shanghai, China, in 1986. He is currently a professor in Computer Engineering and Science at Shanghai University, China. His research interests include software formal methods and software engineering.

Honghao Gao received the Master degree in Computer Science from Zhejiang University, Hangzhou, China, in 2009. He is a member of the China Computer Federation, IEEE and ACM. Currently he is a Ph.D. candidate in the School of Computer Engineering and Science of Shanghai University and his research interests include formal method and model checking.