International Journal of Software Engineering and Knowledge Engineering

Auto-Generation of Test Cases for Infinite States Reactive Systems Based on Symbolic Execution and Formula Rewriting

Donghuo Chen

School of Computer Science and Technology, Soochow University, Suzhou, 215006, China

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

It is an attractive research topic to use model checking technique to automatically generate test cases in the research community of formal method and software testing, and recent years has witnessed many work. For infinite states systems with input/output domains defined on unbounded and abstract types, in many contexts, explicit finite state models are not attained easily with the reasonable cost, therefore, testing with traditional model checking is often considered. This paper presents the idea of auto-generation of test cases based on symbolic execution and temporal formula rewriting method. The method proceeds with building the symbolic representation of program execution model, such that it can avoid explicitly building the model of infinite states systems with the enumeration of value of input and output or state explosion problem; Then temporal formula (test purposes) rewriting is applied to the symbolic execution model of program to generate complex constraint requirements according to the counterexample patterns related to test purposes and the suitable SMT(Satisfiability Modulo Theory) solver is called for generating test cases. Notablely, the procedures of model building and formula rewriting(the light-weighted model checking) are independent, in other word, the counterexamples are derived without system knowledge. The prototype tool is developed, including main data structures and algorithms. Some case studies are conducted though they are simple.

Keywords: Generation of Test Case; Model Checking; Symbolic Execution; Formula Rewriting.

1. Introduction

Testing is the most fundamental and successful method for ensuring the quality of software systems. In modern software engineering practice, the cost of testing activities is more than 50% of total development cost, even sometimes 80%. The testing activities are cost-labored and inefficient, and the automation of testing activities, including generating test cases, executing testing and evaluating the result, can change the situation. But due to the complexity of systems under testing (abbrev. SUT) and testing activities, many challenges in automating testing are faced, especially automated generation of test cases. In black box testing, test cases are generated according to systems requirements, such as equivalence partitioning and boundary value analysis [1]; in the setting of white box testing, given different coverage criterions, test cases are generated directly according to program source codes and system designs, and typical coverage criterions include statement coverage, path coverage and MC/DC [2], etc. But the limitations of these methods are obvious: (1) there is no test oracle for specifying the expected output, (2) no viable techniques work well in

different situations in automating the generation of test cases and (3) no rigorous format specifies test requirement. Many research work [3, 4, 5] uses static syntax analysis or dynamical simulation execution of behavior model derived by program codes to automatically derive test cases, but these methods do not scale well due to high complexity of program codes, more importantly, they do not still solve the test oracle problem.

In modern software engineering, especially model driven development, formal or semi-formal models in different levels of abstraction, which specify static structure and dynamical behaviors of systems, play an important part in understanding and analyzing all facets of the targeted software. It is widely recognized that formal specifications and models can bring much to software testing, and all kinds of formal analysis techniques provide effective means for solving the problems faced by software testing, including theorem prover, model checking and constraint solving. Recent years have witnessed that numerous methods, especially testing with model checker, have been proposed for the derivation of test cases from various kinds of formal specifications and models. The detailed work is summarized in [6, 7]. Model checking [8] is a technique for automatically verifying the correctness of concurrent systems including hardwares/softwares. So far, many model checkers have been successfully applied in industry-level projects. With the inputs of finite state machine modeling the behaviors of systems and the property often specified by temporal logic formula, mode checker confirms whether the property is satisfied by exploring the state space of finite state machine, moreover, for a property not satisfied, the model checker produces a path named a counter example in order to show how the verified system violates the property. In the setting of testing, a counter example related to a property can be naturally interpreted as a test case. As a prior, model checking need build a model of finite state machine, so in the many contexts the state explosion unavoidably happens, say nothing of that the explicit state model is not available at all with the acceptable cost.

When input, output domains and state variables are typed with boundless types, some special methods are necessary to derive manageable models, for example, abstraction interpretation [9, 10]. However, testing models need special consideration, for instance, the models as the oracles, are not permitted to over-approximate the origin systems. In the sequel, many technique cannot be directly used to derive testing models, such predicate abstraction [11], partial order reduction and symmetry reduction.

Motivated by the above discussion, this paper presents the method for auto-generating test cases of reactive systems based on symbolic execution and LTL formula rewriting. Symbolic execution is a basic method for analyzing and reasoning complex systems, which introduces the concept of ”symbolic constant” to build symbolic execution models of systems as a replacement of models with direct enumeration of value about input and output variables, etc., such that explicitly building models is avoidable. The method has been used to automatically generate test cases as a good means [12, 13]. Formula rewriting is an important technique for implementing first order logic theorem provers and SMT (**S**atisfiability **M**odulo and **T**heory) solvers, which are widely used in program analysis and verification. The paper regards LTL formula rewriting based on states and symbolic states as a light-weight model checking technique to automatize the generation of test cases of infinite state system without the procedure of over-/under- approximate abstractions.

The organization structure of the paper is as follows: the second section introduces the elementary knowledge about mode checking and generation of test cases based on model checker; The third section emphasizes on the existed challenges in automatically generating test cases of infinite states system, and at the same time, introduces a standard structure IOSTS (**I**nput/**O**utput **S**ymbolic **T**ransition **S**ystems). The fourth section detailedly discusses LTL formula rewriting based on state as en emphasis except for the syntax and semantics of LTL. The forth section shows the frame and algorithm of auto-generating test cases of reactive systems based on symbolic execution and LTL formula rewriting. The sixth section deliberates the design and implementation of a prototype tool and conducts some small examples. The last section concludes the paper.

2. Model Checking and Test Cases Generation

Model checking [8] is a technique of formal verification presented in 1980s, which achieves great breakthrough in software/ardware verification due to its merit relative to theorem proving, such as automation and counter example generation. With transition system modeling system and temporal logic (a kind of modal logic) specifying property, model checker mathematically proves whether the transition system is a model of some temporal logic formula. Next, some basic definitions are provided for further presentation starting with defining Kripke structure, semantical model of temporal logic, for example LTL. Let AP be the set of atomic propositions

Definition 2.1 Kripke structure K is a tuple (S, S_0, T, L), where $S, S_0, T,$ and L are defined as follows:

- S is a finite set of states;
- $S_0 \subseteq S$ is the state of initial states;
- $T \subseteq S \times S$ is a total binary relation, that is, $\forall s \in S \cdot (\exists t \in S \cdot (S, t) \in T)$;
- $L : AP \to 2^S$ is a label function, for $p \in AP$, $L(p)$ is the set of states, where the proposition p is satisfied.

Generally speaking, the initial state is sole, but here the limitation is loosed. The variables of states in testing models of reactive systems are divided three parts: input variables, output ones and internal ones, among which the former two shows the interaction between SUT and its environment. Figure 1 (a) shows an example of the regular test model. Note that it is straightforward to transform test model to Kripke structure by defining simple rules. Based on Kripke structure, test case in testing with model checker is formally formulated as follows.

Definition 2.2 Given a Kripke structure $K = (S, S_0, T, L)$, a state sequence $tc =< s_0, s_1, \ldots, s_n >$ is named test case, where, for $0 \leq i \leq n, s_i \in S$ and $s_0 \in S_0$. n is the length of test case tc, denoted $|tc|$. a set of test case tc composes a test suite TS.

Figure 1 (b) is a simple example of test case, a finite path in Kripke structure. The in-

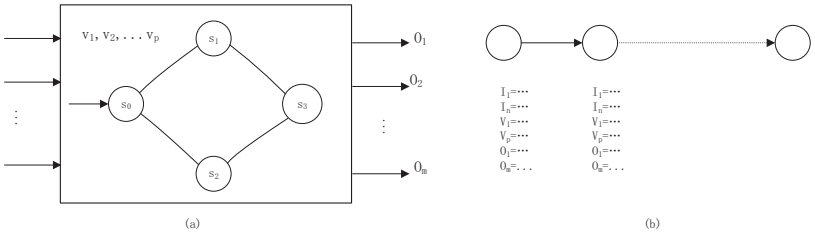

Fig. 1. Test Model and Test Case

put variales, such as $I_1, \ldots, l_n$, and the output ones, such as $O_1, \ldots, O_m$ are regarded as the controllable part and and the observable part, respectively, which play the test oracle when testing execution. In testing based on model checker, tc is a counter example related to some temporal property, automatically generated by calling model checker. The entire technique frame of generation of test case based on model checking is shown in Figure 4. Given a Kripke structure K and a property p specified by LTL formula in the context of the paper, let $CE(p)$ denote some counter example derived from K. Trap property related to test coverage criterion $\mathcal{C}$ is defined as follows.

Definition 2.3 Given a Kripke structure K and a set of properties $\mathcal{P}$ constructing according to some test coverage criterion $\mathcal{C}$, let $CE(\mathcal{P}) = \{CE(p) : p \in \mathcal{P}\}$. Then $p \in \mathcal{P}$ is called trap property if the member in $CE(\mathcal{P})$ is interpreted as a test case.

In black-box testing and white-box testing, test coverage criterion is set according to informal requirement specifications, design specifications and the measure of syntax in program implementations. The typical examples include statements coverage, conditions coverage, decisions coverage, boundary values coverage and composition coverage, etc. Model/specification-based testing enriches the intension of test coverage criterion [15].Testers can construct test coverage criterion based on formal specifications in syntax level and formal models in semantics level, such as specification language-specific coverage criterion and transition systems coverage criterion, and one can find more details in [7]. Moreover, the technique of model checking provides easiness of generating test suite related to different test coverage critera.

Next, a simple example illustrates how to construct trap property using temporal logics, such as LTL [14]. Firstly, let test coverage criterion $\mathcal{C}$ be the edges of state transitions (simple transition coverage) explicitly shown in Figure 1 (a), then trap properties is as follows:

$$s_0 \wedge \alpha \rightarrow \mathcal{X}(\neg s_1)$$
$$s_1 \wedge \beta \rightarrow \mathcal{X}(\neg s_3)$$
$$s_0 \wedge \gamma \rightarrow \mathcal{X}(\neg s_2)$$
$$s_2 \wedge \delta \rightarrow \mathcal{X}(\neg s_3)$$

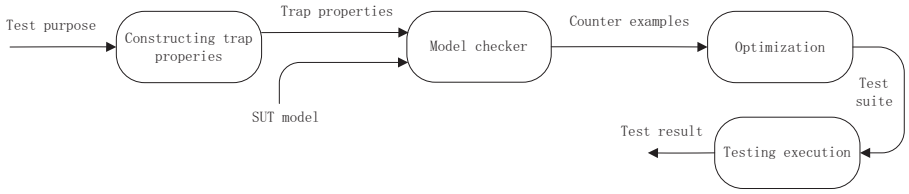

Fig. 2. Procedure of Auto-Generation of Test Cases

Where, s_0, s_1, s_2 and s_3 represent the pre/post state conditions of transitions, and α, β, γ and δ are guard conditions. In the case of the more complex coverage criterion, such as control and data coverage criteria, the construction of trap properties is not so straightforward and the procedure of generating counter examples needs more cost.

The technique frame in Figure 4 defines the main activities of testing with model checker. Consider the fact that the manageable formal model of finite state machine is uneasily attainable in most cases, therefore the key activity, model checking, is disenabled. The paper presents a novel method, which specifies model of SUT in the form of IOSTS structure, and uses symbolic execution and formula rewriting as the basis of auto-generation of test cases. The method provides a more feasible and more flexible replacement of testing with model checker when preserving the advantages of model checking. The next section introduces the details about IOSTS structure.

3. Generation of Test Cases for Infinite States System

In testing with model checker, the availability of testing model as the test oracle, such as finite state machine, overwhelmingly challenges testers. The work of building testing model is demanding arduous, especially SUT has to be specified by infinite states model with boundless types. The previous work in testing model checker targeted the control-oriented systems. Certainly, testing model as the test oracle, cannot be directly derived from the implementation of SUT, including the source codes and the machine codes, etc..

IOSTS structures symbolically specify behavior and the data without explicitly enumerating the values of variables with infinite domain, and have appropriate expressiveness ability to specifying test models as test oracle with acceptable cost. Much research work about IOSTS-based testing can be found in $[16, 18]$ etc., and these work show the usefulness as a good media for integrating formal methods and testing. Next, the detail of IOSTS is introduced. First, some conventions are clarified. v denotes a variable and D_v the domain of the values v takes. Naturally, a set of variables is expressed as $V = \{v_1, \cdots, v_n\}$, and as the product of $D_{v_1} \times \cdots \times D_{v_n}$. Thus an element of D_V is a vector of values for the variables in V. In the IOSTS model, the guards (predicates) are logic formulae made of the set of variable V and logical connectives, whose values are in the set $\{\mathit{True}, \mathit{False}\}$. An assignment for a variable v and a set $V' \subseteq V$of variables depending on the set V are formulated as $D_V \rightarrow D_v$ and $D_V \rightarrow D_{V'}$.

Definition 3.1 An Input/Output Symbolic Transition System M is defined by a tuple $(V, \Theta, \mathcal{A}, T)$, where:

- $V = V_i \cup V_x$ is the set of variables, partitioned into a set V_i of internal variables and a set V_x of external variables.
- Θ is the initial conditions. It is a predicate $\Theta \subseteq D_{V_i}$ defined on internal variables. In general case, it is assumed that Θ has a unique solution in D_{V_i};
- $\mathcal{A} = \mathcal{A}_? \cup \mathcal{A}_i \cup \mathcal{A}_\tau$ is the finite set of alphabet representing input/output actions and internal actions. Each input or output action a carries parameters, whose types are specified by $sig(a) =< t_1, \cdots, t_k >$ and regarded as the signature of action a in $\mathcal{A}_?$ and $\mathcal{A}_!$, which are the set of input actions and the set of output actions, respectively. For distinguishing input actions and output actions, the name of the former suffixes "?", accordingly, that of the later prefixes "!", such as $a?$ and $a!$.
- T is a finite set of symbolic transitions. A symbolic transition $t =< a, \Delta, G, A >$ in T, also written $[a(\Delta) : G(v, \Delta)?v' := A(v, \Delta)]$, is defined as follows:
  (a) $a \in \mathcal{A}_? \cup \mathcal{A}_!$ is an input or output action and $\Delta =< p_1, \cdots, p_k >$ is the communication parameters of a. Without loss of generality, it is assumed that each action a always carries the same parameters vector Δ which is well typed by the signature $sig(a) =< t_1, \cdots, t_k >$ of a, such that both D_Δ and $D_{sig(a)}$ are $D_{t_1} \times \cdots \times D_{t_k}$; Otherwise, $a \in \mathcal{A}_\tau$, as internal action, is not often equipped with communication parameters.
  (b) $G \subseteq D_{V_i} \times D_\Delta$ is a guard defined on the internal variables and the communication parameters. For the sake of efficiently reasoning, the guard conditions are expressed by formulae in a logic framework, such as propositional logic, whose satisfiability is decidable.
  (c) An assignment $A : D_{V_i} \times D_\Delta \to D_{V_i}$ mainly defines the evolution of the state variable of reactive systems, i.e. internal variables. For convenience, let A_v and $A_{V'}$ are the projection of A on the assignment of the variable $v \in V$ and the set of variables $V' \subseteq V$.

From the above definition, we conclude that IOSTS structure is an extension of I/O automata with communication parameters, actions, guards and assignment expressions which detailedly specify complex data dependency besides the behaviors of SUT. A toy example is shown in Figure 3. IOSTS structures do not define the concept of state due to its abstract format, as a replacement, it equips with a special program counter variable encoding the control locations between transitions, such as l_0, l_1, l_2 and l_3 in Figure 3, among which, the first location l_0 is the initial control location. The interactional behaviors between SUT and environment are modeled by sequences of control locations and actions without internal actions, such as $l_0 \xrightarrow{in?(p)} l_1 \xrightarrow{in?(p)} l_2 \xrightarrow{ok!(p)} l_0 \cdots$ and $l_0 \xrightarrow{in?(p)} l_1 \xrightarrow{in?(p)} l_2 \xrightarrow{nok!(p)} l_0 \cdots$. These sequences identify actions in step when executing testing. Strictly speaking, a sequence modeling the interactional behavior should exclude the existed internal actions.

IOSTS is easily used to visually specify the interactive behaviors between the environment and SUT, therefore, which, some tools, such as STG [24], use to specify both testing

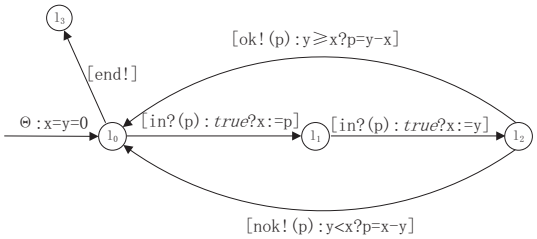

Fig. 3. An Example of IOSTS Structure

model and testing purpose. But the method has some shortcomings. On the one hand, the new algorithm need redesign instead of reuse the existing algorithms and tools of model checking; on the other hand, IOSTS has the limited expressivity to specify the other practically needed test purposes despite its many advantages of expressing interactive actions sequence. In practice, testers often have to test the targeted system from different angles besides its interactive behaviors, such as control and data flow, inclusively, a more expressive formalism is needed. LTL, a specification formalism widely used in formal verification, can concisely and naturally specify all kinds of test purposes including interactive behaviors. Take an example as follows: $\mathcal{F}(def(x) \wedge \mathcal{X}(\neg def(x))\ U\ u(v,x))$, where x and v are variables, $def(x)$ and $u(v,x)$ are the predicates "x is defined" and "v is assigned with the expression containing the variable x". It explicitly defines the test purpose of DU chain related to control and data flow. Another example is $\mathcal{GF}s$ with complex temporal feature.

Considering the aspects of generation of test cases for infinite states system, a novel idea is presented, which integrates the advantages of traditional model checking, SAT and IOSTS, the abstract testing model. Figure 4 shows the overall framework. It takes as the inputs test models specified by IOSTS and test purposes specified by LTL formulae.

The formula rewriting and symbolic simulator, as the replacement of model checker, play the pivotal roles. The formula rewriting technique is used to compute the necessary condition, which the test case related to some trap property should satisfy, instead of verify whether a property is satisfied by a path just as in runtime verification. The necessary condition is encoded by a free-quantifier first order formula, denoted $Constraint()$. Given a test case $\pi =< s_0, s_1, \cdots, s_n >$, $s_0 \wedge s_1 \wedge \cdots \wedge s_n \models Constraint()$ holds. The next section will expound the technical detail of LTL formulae rewriting.

Symbolic execution is an important technique of handling the symbolic and abstract specification without directly enumerating huge space of concrete states of the targeted system, widely used in program static analysis, testing and verification. Because LTL is semantically interpreted as state-based structure, the symbolic-state based execution model of IOSTS is constructed using symbolic execution technique. Firstly, some basic concepts are provided.

Definition 3.1 Given an IOSTS structure $M = (V, \Theta, \mathcal{A}, T)$, for a variable $v \in V$, let $\mathcal{T}$ be the type of v, D_v the domain of value. The symbolic type of $\mathcal{T}$ denoted $\mathcal{ST}$ is a type,

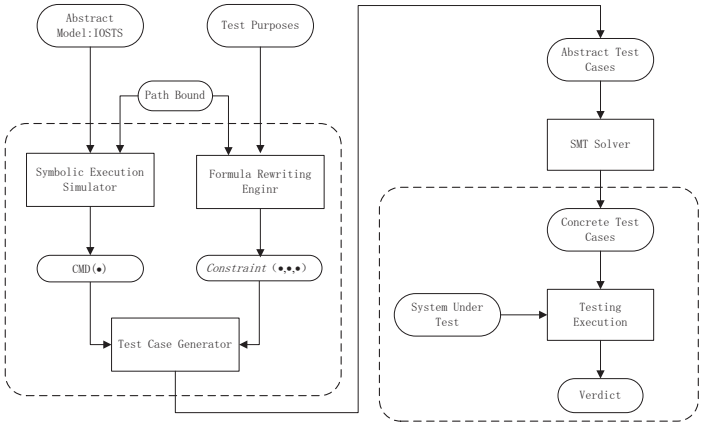

Fig. 4. The Main Framework of Auto-Generation of Test Cases for Infinite States System

whose domain is a infinite countable set $\mathcal{SD}_v = \{t_1, t_2, \cdots\}$, and the elements $t_1, t_2, \cdots$ are named symbolic constants. Each symbolic constant t_i can represent any concrete value typed $\mathcal{T}$.

In Figure 3, all variables are integer int, whose symbolic type is $\mathcal{S}int$. Moreover, a special symbol constant $\perp$ is introduced for every symbolic type representing nondeterministic value. Given two groups assignments: $(v_1 = t_1, v_2 = t_1)$ and $(v_1 = \perp, v_2 = \perp)$ for variables v_1 and v_2, it is concluded that the predicate $v_1 = v_2$ is satisfied in the first group, but not in the second.

With the enumeration of the values of variables, the semantics model of IOSTS structure can be defined in term of labeled transition system, LTS for short [18]. But accurately modeling infinite states systems by this means is infeasible in model-based testing. Next, the concept of symbolic state is introduced, as the basis of deriving the symbolic state based symbolic execution model from IOSTS structure. For an IOSTS structure $M = (V, \Theta, \mathcal{A}, T)$, let $SExpr(V)$ be the meaning expressions set defined on $\bigcup_{v \in V} D_v$, $\bigcup_{v \in V} \mathcal{SD}_v, f_1, \cdots, f_n$, where for $1 \leq i \leq n$, f_i is the operator with unambiguous meaning, such as $plus$, $substract$ and reference operator[ ] of element in an array, etc., $Pred(SExpr(V))$ the set of predicates defined on $SExpr(V)$.

Definition 3.2 Given an IOSTS structure $M = (V, \Theta, \mathcal{A}, T)$, binary tuple (Ω, Π) is named symbolic state, where $\Omega : V \to SExpr(V)$ is an assignment, which assigns the expression in $SExpr(V)$ to the variables in V, and $\Pi \in Pred(SExpr(V))$ is a predicate named state condition or state invariant, also, which is regarded as a new state variable. $\Omega[v]$ and $\Omega[V']$ are the projection of the assignment Ω in the variable $v \in V$ and the variables set $V' \subseteq V$.

According to the following two rules, the symbolic execution model related to an

IOSTS structure $M = (V, \Theta, \mathcal{A}, T)$ can be constructed.

Rule 1: The initial symbolic state (Ω_0, Π_0): Ω_0 assigns the symbolic constants to the variables in V and $\Pi_0 = \Theta[\Omega_0]$, where $\Theta[\Omega_0]$ is the expression replacing the variables in Θ with the related symbolic constants assigned by Ω_0.

Rule 2: For $[a(\Delta) : G(V_i, \Delta)?v' := A(v, \Delta)] \in T$ and $\Delta = \{p_1, \cdots, p_k\}$, the symbolic states pair $(\xi_1 = (\Omega_1, \Pi_1), \xi_2 = (\Omega_2, \Pi_2))$ is derived by assigning the symbolic constants to the variables in Δ, where, $\Omega_2 = A(\Omega_1[V_i]), \Omega_1[\Delta])$, $\Pi_2 = G(\Omega_1[V_i]), \Omega_1[\Delta])$. Generally, if the A only rearranges the only variable v, then $v = \Omega_2[v]$ and $\forall w \in V \cdot (w \notin \{v\} \Rightarrow \Omega_2[w]) = \Omega_1[w]$ in ξ_2. (ξ_1, ξ_2) is named symbolic states transition.

Starting from the initial symbolic state (Ω_0, Π_0), a finite or infinite symbolic states tree can be constructed by using Rule 1 and Rule 2, and the tree with the root node (Ω_0, Π_0) is the symbolic execution model related to IOSTS structure M, denoted $\mathcal{M}$. Given a finite symbolic execution path $\pi =< \xi_0, \xi_1, \cdots, \xi_n >$, $PC = \Pi_0 \wedge \Pi_1 \cdots \wedge \Pi_n$ is called the path condition of π. Let $\pi =< \xi_0, \xi_1, \cdots >$ be the symbolic execution path with $\xi_0 = (\Omega_0, \Pi_0)$, and ι the substitution using numeric values to instantiate the symbolic constants. Then π^ι represents the concrete execution path in labeled transition structure $M^\dagger$ which is the semantics model of M and PC is satisfiable with the assignment ι. Further, let $path(\mathcal{M})$ and $Path(M^\dagger)$ be the feasible symbolic execution paths in symbolic execution model $\mathcal{M}$ and concrete execution paths set in labeled transition system $M^\dagger$, respectively, according to the relationship between $\mathcal{M}$ and $M^\dagger$, the following conclusion can be drawn:

$$\pi \in Path(M^\dagger) \Rightarrow \exists \pi' \in Path(\mathcal{M}) \exists \iota \cdot \pi'^\iota = \pi$$
$$\pi \in Path(\mathcal{M}) \Rightarrow \forall \iota \exists \pi' \in Path(M^\dagger) \cdot \pi^\iota = \pi'$$

In the symbolic execution model, the internal and external variables are uniformly regarded as state variables, but the function of the external variables V_x, as communication parameters, are localized, that is, the values of communication parameters Δ in some transition cannot be again referred in other locations, where are set the nondeterminate value $\perp$. About IOSTS structure, some more complex features are not considered in the paper, such as nondeterminacy and concurrency. Of course, the presented method can handle the IOSTS structure with nondeterminacy and concurrency.

4. LTL Formula Rewriting

The last section introduces the challenges faced by testing systems with boundless types based on model, and presents the solution shown in Figure 4. The section will put the emphasis on the technical details starting with LTL formula rewriting.

4.1. *LTL and Interpretation of Finite Path*

4.1.1. *Syntax and Semantics of LTL*

In most cases, the behavior of the targeted system can be modeled by tree-like structure or the set of finite and infinite paths; in the level of syntax, the properties satisfied by the

behavior model is specified using temporal logic, a concise and intelligible format. Linear temporal logic (LTL) [19] is a widely used temporal logic, and many model checkers based on LTL have been applied to verify the industry-level hardwares and softwares, typically for example, SPIN. Let AP be a set of atomic propositions, the abstract definition of LTL formula is the following:

$$\alpha \triangleq true|false|a|\neg\alpha|\alpha_1 \wedge \alpha_2|\alpha_1 \vee \alpha_2|\alpha_1 \rightarrow \alpha_2|\alpha_1 \mathcal{U} \alpha_2|\mathcal{F}\alpha|\mathcal{X}\alpha|\mathcal{G}\alpha$$

where, $a \in AP$, $\mathcal{X}$, $\mathcal{G}$, $\mathcal{F}$ and $\mathcal{U}$ are temporal operators, which mean "next", "globally", "future" and "until", respectively, and other operators standard logic connectives. Next, the complete definition of the semantics of LTL formulae is provided.

The semantics of LTL formulae is defined in term of Kripke structure. Given a Kripke structure K and a infinite path π, the following (1)-(10) terms detailedly interpret the LTL formulae in the level of semantics.

(1). $(K, \pi) \models true$
(2). $(K, \pi) \not\models false$
(3). $(K, \pi) \models a$ `iff` $a \in L(\pi(0))$
(4). $(K, \pi) \models \neg\alpha$ `iff` $(K, \pi) \not\models \alpha$
(5). $(K, \pi) \models \alpha_1 \wedge \alpha_2$ `iff` $(K, \pi) \models \alpha_1$ `and` $(K, \pi) \models \alpha_2$
(6). $(K, \pi) \models \alpha_1 \vee \alpha_2$ `iff` $(K, \pi) \models \alpha_1$ `or` $(K, \pi) \models \alpha_2$
(7). $(K, \pi) \models \mathcal{X}\alpha$ `iff` $(K, \pi^1) \models \alpha$
(8). $(K, \pi) \models \mathcal{F}\alpha$ `iff` $\exists i \geq 0 \cdot (K, \pi^i) \models \alpha$
(9). $(K, \pi) \models \mathcal{G}\alpha$ `iff` $\forall i \geq 0 \cdot (K, \pi^i) \models \alpha$
(10). $(K, \pi) \models \alpha_1 \mathcal{U} \alpha_2$ `iff` $\exists i \geq 0 \cdot (K, \pi^i) \models \alpha_2 \wedge \forall 0 \leq j < i \cdot (K, \pi^j) \models \alpha_1$

where, $\pi(0)$ denotes the initial state of π, π^i the postfix of π starting from the $(i+1)^{th}$ state. If for every path π, $(K, \pi) \models \alpha$ holds, then $(K, \pi) \models \alpha$ can be simplified not explicitly illustrating π, such as $K \models true$ and $K \not\models false$. Moreover, in the unambiguous context, K is also omitted, such as $\pi \models \alpha$. The infinite path semantics of LTL formulae has built the basis of relating trap properties specified by LTL formulae and test cases modeled by finite path, and the next subsection discusses the detail.

4.1.2. *The Meaning of Finite Path*

Property violation can be mostly witnessed by a finite path (test case), thus some insights about the meaning of finite path related to LTL formula need clarify in the context of the infinite path semantics of LTL. Firstly, consider the finite path in Figure 5 (a). Obviously, $\mathcal{G}(p \rightarrow \mathcal{X}q)$ is not satisfied by the path as the state s_n labeled by p has not a successive state labeled by q; in the other hand, it cannot refute the $\mathcal{G}(p \rightarrow \mathcal{X}q)$, as a counter example. Any infinite path can avoid the predicament. For solving the problem created by finite path, some strategies have to be used for handling the finite path and LTL formula. A straightforward way is to repeat the final state of the finite path, such that an infinite path is derived when the main feature of the origin path is preserved. But when the infinite path is used to evaluate LTL formula, some side effect exists. Again consider the example in Figure 5 (a),

if $s_n \in L(\neg q)$ which means $s_n \notin L(q)$, we immediately derive the incorrect conclusion that the path is the counter example of $\mathcal{G}(p \rightarrow \mathcal{X}q)$. Next, the test case model is introduced. Based on the test case model, a more elegant measure is adopted to formally clarify how a finite path refutes a property in the setting of testing. Let $Length_t(s) \leq l$ be a proposition, which evaluates to true only if the current state of a sequence is any state prior to the final state. The similar idea can be found in [20], where l is the size of the finite path t

Definition 4.1.2.1 Given a Kripke structure $K = (S, s_0, T, L)$ and a test case $t =< s_0, s_1, \cdots, s_n >$, the test case model related t denoted K^t is the tuple (S_t, S_0, T_t, L_t), where $\{s_0, s_1, \cdots, s_n\} \subseteq S_t \subseteq S, T_t \subseteq T$, for $p \in AP$ $L_t(p) = L(p)$,$L_t(length_t(s_i) \leq l) = \{s_0, s_1, \cdots, s_l\}$, and $L_t(\neg(length_t(s_i) \leq l)) = \{s_i | i > l\}$.

The test case model derived from a finite path t may be regarded as a special Kripke structure, in which, all paths include the prefix t, moreover we can distinguish the prefix t and the corresponding postfix by the proposition $Length_t(s) \leq l$, that is, in the prefix, each state satisfies $Length_t(s) \leq l$, but in the postfix, each state does not. By more observation, we easily find that $K^t \not\models \mathcal{G}(p \rightarrow \mathcal{X}q)$ does not certainly lead to the conclusion that the prefix t must be the counter example of $\mathcal{G}(p \rightarrow \mathcal{X}q)$. The original intention for defining test case model that if $K^t \not\models p$ for some property, then t is the counter example of p, otherwise $K^t \models p$ is still in unfulfilment. To bridge the gap, we use the proposition $length_t(s) \leq l$ to construct a new formula p^t from p. Take an example as $p = \mathcal{G}(p \rightarrow \mathcal{X}q)$ and $p^t = \mathcal{G}(Length_t(s) \leq l \rightarrow (p \rightarrow \mathcal{X}(Length_t(s) \leq l \rightarrow q)))$. p^t is derived by recursively replacing the subformula α of some property p with $Length_t(s) \leq l \rightarrow \alpha$. Obviously, $K^t \not\models p^t$ implies that $K^t \not\models p$, moreover, t or its prefix refutes p. Using the above technique, a finite path can be used to evaluate the true value of LTL formulae when preserving the infinite path semantics of LTL formulae. Of course, $K^t \models p^t$ only shows fact that t cannot refute p, but in the setting of model based testing, it is rational.

In model checking, another class of properties needs infinite path to refute, such as the liveness property $\mathcal{F}p$. When the trap properties are specified by the class of formulae, we have to find the appropriate finite paths showing the features of infinite paths in order to falsify the trap properties as counter examples (test cases). Model-checkers make use of so called lasso-shaped sequences, a special kind of infinite path where a finite subsequence at the end of a trace is repeated infinitely, which produce the test cases related to the liveness trap properties. Let $\pi =< s_0, \cdots, s_i, \cdots, \cdots >$, $\rho =< s_i, \cdots, s_j >$ and $\rho' =< s_0, \cdots, s_{i-1} >$. If $(s_j, s_i) \in T$, $\rho'\rho^\omega$ is a lasso-shape path with ρ infinitely repeated. In essence, the structure and property of the lasso-shape path $\rho'\rho^\omega$ can be manifested by the finite path $\rho'\rho\rho$ shown in Figure 5, even $\rho'\rho$. It is summarized that the liveness trap properties can be falsified by finite paths derived from lasso-shape paths. the lasso-shape path $\rho'\rho^\omega$ is denoted $K^{\omega\rho}$ as the test model related to the liveness trap property $\mathcal{F}p$.

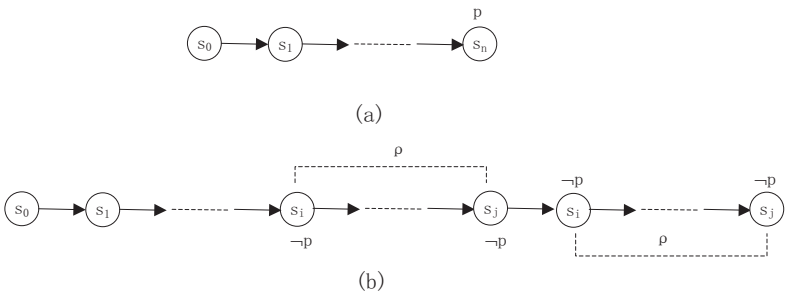

Fig. 5. Examples of Finite Paths

4.2. *Formula Rewriting*

The state-based LTL formula rewriting approach is used to automatically verify the assertions specify by LTL about the execution trace of the targeted system in running verification [21]. Due to the complexity of space and time in exploring full state space, LTL formula rewrting, as a light-weight model checking technique, can be introduced into the domain of testing with model checker. It helps us to take low cost to generate test case and reduce the redundancy of test suite [22]. few research work in this field is done. The paper extends the usefulness of LTL formula rewriting in testing with model checker as the basis of automatically generating test cases of infinite states system. The subsection will expound how to integrate the formula rewriting and symbolic execution model of program to compute the theory expressed in the multi-type first order logic for formalizing the requirement of the corresponding test cases.

Given a Kripke structure $K = (S, s_0, T, L)$ and a finite path $\pi =< s_0, s_1, \cdots, s_n >$, the formula rewriting approach determines whether π satisfies some trap property targeting to generate a test case by recursively checking the responsibility of the corresponding states in π. Let $\theta[s]$ denote that state s is applied to the formula. In essence, we can regard formula rewriting as a light-weighted model checking technique in the sense that it checks the set of states in the related path rather than the whole state space. Application of a state to a formula determines, whether the propositions valid in that state have an effect on the formula. The following rewriting rules show for the state how to be applied the formula according to the syntax structure of the formula.

$$
\begin{array}{rcll}
a[s] & = & \left\{ \begin{array}{l} false \;\; s \notin L(a) \\ true \;\; s \in L(a) \end{array} \right. & (RW1) \\
\neg\alpha[s] & = & \neg(\alpha[s]) & (RW2) \\
(\alpha_1 \wedge \alpha_2)[s] & = & \alpha_1[s] \wedge \alpha_2[s] & (RW3) \\
(\alpha_1 \vee \alpha_2)[s] & = & \alpha_1[s] \vee \alpha_2[s] & (RW4) \\
(\alpha_1 \rightarrow \alpha_2)[s] & = & \alpha_1[s] \rightarrow \alpha_2[s] & (RW5) \\
(\mathcal{X}\alpha)[s] & = & \alpha & (RW6) \\
(\mathcal{G}\alpha)[s] & = & \alpha[s] \wedge \mathcal{G}\alpha & (RW7) \\
(\mathcal{F}\alpha)[s] & = & \alpha[s] \vee \mathcal{F}\alpha & (RW8) \\
(\alpha_1 \mathcal{U} \alpha_2)[s] & = & \alpha_2 \vee (\alpha_1[s] \wedge (\alpha_1 \mathcal{U} \alpha_2)) & (RW9)
\end{array}
$$

According to the state-based formula rewriting rules, the rule path-based formula rewriting of the finite path, denoted $\alpha[\pi]$ or $\alpha[< s_0, s_1, \cdots, s_n >]$, can be formulated as follows:

$$
\begin{array}{l}
\alpha[< s_0, s_1, \cdots, s_n >] = (\alpha[s_0])[< s_1, \cdots, s_n >] \\
\qquad\qquad = ((\cdots((\alpha[s_0])[s_1])\cdots)[s_{i-1}])[< s_i, \cdots, s_n >] \quad (RW)
\end{array}
$$

The above expression defines the detailed procedure of evaluating the formula in the path. The rewriting result of $\phi[s_k]$ is $true, false, \circ\mathtt{r}\ \phi'$, where $0 \leq k < n$ and $\phi = (\cdots((\alpha[s_0])[s_1])\cdots)[s_{k-1}]$. According to different results, different conclusions are derived:

- The result $false$ implies that a counter example is found, for example $< s_0, s_1, \cdots, s_k >$, and the procedure of rewriting is terminated,
- if the result is one of $true$ and α' for $k = n$, a witness of validating the property is found in the test model, otherwise
- the procedure of rewriting continues.

Given $\alpha = \mathcal{G}(p \rightarrow \mathcal{X}q)$ and $\pi =< s_0 = (\neg p, q), s_1 = (p, q), s_2 = (p, \neg q) >$, the next example shows the method of path based formula rewriting.

$$
\begin{array}{l}
\alpha[\pi] = (\mathcal{G}(p \rightarrow \mathcal{X}q)[(\neg p, q)])[\pi^1] = ((p \rightarrow \mathcal{X}q)[(\neg p, q)] \wedge \mathcal{G}(p \rightarrow \mathcal{X}q))[\pi^1] \\
= ((p \rightarrow \mathcal{X}q)[(p, \neg q)] \wedge \mathcal{G}(p \rightarrow \mathcal{X}q))[\pi^2] \\
= ((true \rightarrow q)[(p, q)] \wedge \mathcal{G}(p \rightarrow q)[(p, q)])[\pi^3] \\
= ((p \rightarrow \mathcal{X}q)[(p, q)] \wedge \mathcal{G}(p \rightarrow \mathcal{X}q))[(p, \neg q)] \\
= (true \rightarrow q)[(p, \neg q)] \wedge (\mathcal{G}(p \rightarrow \mathcal{X}q))[(p, \neg q)] = false
\end{array}
$$

Conclusively, π is the test case for the trap property α.

4.3. *Temporal Path Constraint*

Model checking technique cannot be directly applied to the symbolic execution model derived from IOSTS structure for generating test cases related to trap properties. Here a

novel method without calling model checker is presented, and it contributes us what testing with model checker can give, which uses the adapted version of LTL formula rewriting rules to compute the complex constraint specifying the test purpose (named temporal path constrain). The temporal path constrain is encoded into the quantifier-free first order formulae with multi-type, in other word, the quantifier-free first order formula, which can be efficiently reasoned by the SMT tool, defines the necessary condition of the path as the targeted test case. Notablely, the temporal path constrain is computed without the knowledge of testing model. Given a LTL formula α, a path bound n and a infinite path $\pi =< \xi_0, \xi_1, \xi_2, \cdots >$, where every state is labeled by $true$:

$$\begin{aligned}\alpha[< \xi_0, \xi_1, \xi_2, \cdots >] = \cdots &= \overline{((\cdots((\alpha[\xi_0])[\xi_1])\cdots)[\xi_{i-1}])}[\xi_i \cdots] = \cdots \\ &= \underline{((\cdots((\alpha[\xi_0])[\xi_1])\cdots)[\xi_n])}[\xi_{n+1}, \cdots] \quad (RW')\end{aligned}$$

(RW) and (RW') with the consistent syntax have different functions: RW' sequently labels the state in π with the necessary condition for confirming the temporal property α, but (RW) verifies whether α is satisfied. In (RW'), the part with the overline represents the result after labeling the state ξ_{i-1}. When the result, the formula with the underline after labeling ξ_n is $true$, the rewriting procedure terminates. $< \xi_0, \xi_1, \cdots, \xi_n >$ is called the temporal path constraint related to α, also denoted $\xi_0 \wedge \xi_1 \wedge \cdots \wedge \xi_n$. If $((\cdots((\alpha[\xi_0])[\xi_1])\cdots)[\xi_{i-1}])$ for $i \geq 1$ is $false$, it is concluded that α is inconsistent and the rewriting procedure terminates.

The rewriting rule (RW') can be applied to compute temporal path constraint of test case related to some trap property in the case of testing with model. Next, let us deliberate the concepts of focus state of test case and counterexample pattern. Given a trap property α and the corresponding test case $\pi =< s_0, \cdots, s_n >$, the state s_n is titled the focus state for the test case π in the sense that any proper prefix π' of π cannot refute α ($K^{\pi} \not\models \alpha^{\pi}$ and $K^{\pi'} \models \alpha^{\pi'}$), or π' refutes α and the length of π' is not in the range of path bound. Provided that the length of counterexample related to some trap property is predefined and the state in the last position is the focus state for test case, the rewriting rules (RW') can determine the responsibility taken by every state in the corresponding position according to the structure of trap property, called the counterexample pattern. Figure 6 definitely shows the pattern for formulae in the form of $\mathcal{F}\alpha, \alpha_1 \mathcal{U} \alpha_2, \mathcal{G}\alpha$ and $\mathcal{X}\alpha$, and we can easily derive the pattern for the other formulae. The patterns clearly formalize the constrain conditions for the corresponding trap properties, such as $\exists i \geq 0 \cdot (\forall j \leq i \cdot (\pi^j \models \alpha) \wedge \pi^{i+1} \not\models \alpha)$ for $\mathcal{G}\alpha$. Note that the pattern of counterexample for $\mathcal{F}\alpha$ should have the feature of lasso-shape path. If α is not atomic formula or its negative, the rule (RW') is recursively applied.

Let $\mathcal{M}$ be the symbolic execution model of the IOSTS structure $M = (V, \Theta, \mathcal{A}, T)$ and P the set of trap properties. Given some trap property $\theta \in P$, $\pi =< \xi_0, \xi_1, \cdots >$ and the corresponding counterexample pattern $\mathcal{P}$, the expression $Constraint(\pi, \mathcal{P}, \theta)$ denotes the temporal path constrain, which is necessarily satisfied by any concrete test case related to the trap property θ, and $Sat(\xi_i, \theta')$ denotes $\theta'[< \xi_i, \xi_i + 1, \cdots >]$. According to RW', $Sat(\xi_i, \theta')$ is computed recursively calling rewriting rules $(RW1) - (RW9)$. Of course, the rule $(RW1)$ need replace by the following rules $(RW1')$ and $(RW1'')$ due to different

essence of (RW) and (RW'). The temporal path constraint $Constraint(\pi, \mathcal{P}, \theta)$ such as the trap properties $\mathcal{X}\alpha$ and $\mathcal{G}\alpha$ are listed as follows:

$$\begin{aligned} Constraint(\pi, \mathcal{P}, \mathcal{X}\alpha) &= Sat(\xi_0, true) \wedge Sat(\xi_1, \alpha) \\ Constraint(\pi, \mathcal{P}, \mathcal{G}\alpha) &= Sat(\xi_0, \alpha) \wedge \cdots \wedge Sat(\xi_{n-1}, \alpha) \wedge Sat(\xi_n, \neg\alpha) \\ &\vdots \end{aligned}$$

As noted previously, (RW') is used to compute the temporal path constrain, the necessary condition necessarily satisfied by test case for some trap property, therefore, $(RW1)$ should be replaced by the following rules $(RW1')$ and $(RW1'')$

$$\begin{aligned} a[s] &= a \ \ (RW1') \\ \neg a[s] &= \neg a \ (RW1'') \end{aligned}$$

Where a is the atomic proposition. The rules $(RW2) - (RW9)$ reflect the logic structure of LTL formula with the underlying temporal semantics, and $(RW1')$ and $(RW1'')$ label states with atomic propositions or their negate.

Given a trap property, the concrete test case is jointly determined by the temporal path constrain and the symbolic path produced from IOSTS. Let a tuple $Theory(\alpha, \mathcal{P}, \pi) = (Constraint(\pi', \mathcal{P}, \alpha), CMD(\pi))$ decode the complete assertions satisfied by the concrete test case related to the trap property α, where $CMD(\pi)$ specifies the abstract behavior of the symbolic path π. When $\pi =< \xi_0, \cdots, \xi_n >$, straightforwardly, $CMD(\pi)$ is the conjunction of the assignments and state variants in all symbolic states, including Ω_i and Π_i, $0 \leq i \leq n$. Modern SMT tools, such as Z3, can effectively solve $Theory(\alpha, \mathcal{P}, \pi)$ to derive the concrete test case. $Constraint(\pi', \mathcal{P}, \alpha)$ and $CMD(\pi)$, the quantifier-free first order formulae with multi types, are defined on the variables set V and the symbolic constants of the corresponding types. It is highlighted that π' in $Constraint(\pi', \mathcal{P}, \alpha)$ is only a time line induced by the a totally ordered set $(\mathcal{S}, <)$, isomorphic to $(N, <)$. In order to explicitly specify the assertions and relations among them in different states, the variables occurring in $Constraint(\pi', \mathcal{P}, \alpha)$ and $CMD(\pi)$ need rename, in other word, relate the variables with the states (pisitions). Only by modifying the rule $(RW1')$ and $(RW1")$, can the formula rewriting and variable renaming be processed simultaneously. The new rules $R - RW1'$ and $R - RW1''$ are shown as follows:

$$\begin{aligned} a[s] &= a[v^s / v \in depend(a)] \ \ (R - RW1') \\ \neg a[s] &= \neg a[v^s / v \in depend(a)] \ (R - RW1'') \end{aligned}$$

Where $depend(a)$ is the set of variable occurring in a and v^s is the renamed version of v. Similarly, $CMD(\pi)$ is renamed according to the following expression:

$$CMD(\pi)[V^\pi / V] = \bigwedge \{\Omega_i[V^{\xi_i}/V] | 0 \leq i \leq n\} \wedge \bigwedge \{\Pi_i[V^{\xi_i}/V] | 0 \leq i \leq n\}.$$

$\Omega_i[V^{\xi_i}/V]$ and $\Pi_i[V^{\xi_i}/V]$ are the result of renaming the variables in Ω_i and Π_i. The implementation of variables renaming can easily resort to the SSA (Static Single Assignment [25]) method used in compiling theory. Inclusively, with the complete conditions of

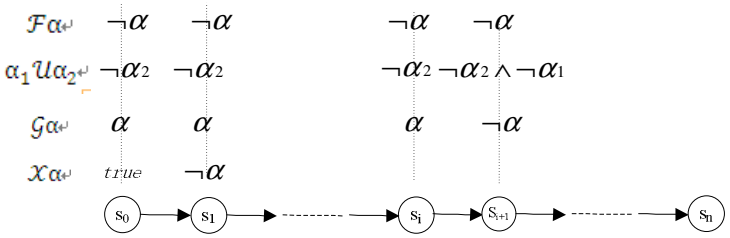

Fig. 6. The Patterns of Counter Example for $\mathcal{F}\alpha, \alpha_1\mathcal{U}\alpha_2, \mathcal{G}\alpha$ and $\mathcal{X}\alpha$

test case related to some trap property expressed by the quantifier-free first order formula as the input, the SMT tool is called to automatically generate the concrete test case.

5. Auto-Generation of Concrete Test Cases

5.1. *Elements of SMT*

SMT (Satisfibility Module Theory) is the extension of SAT in predicate logic [23]. SMT solvers, which play important roles in analysis and verification of software and hardware, are the core engine in the technique frame presented here. Pure formal logic discusses the theory about general reasoning, but does not involve the knowledge about some special background. In practice, most SMT tools can reason complex knowledge expressed by the logic formulae including all sorts of background theory, such as integer theory, real theory, EUF(Enqulity of Uninterpreted Function) theory and some special theory about array, list and vector, etc., used for analysis of software and hardware. The built-in theory provides the strong ability to specify and resolve the problem like generating test cases.

Many SMT tools were developed due to the purpose of research and application in industry, such as Z3, CVC and MathSAT. Most tools provide the standard and usable man-machine interfaces and programming ones. The researchers designed the standard language and commands for communicating with SMT tools named SMT-LIB, therefore SMT tools are conveniently used in different cases. Please refer the literature [23], etc., about the more detail of SMT.

5.2. *Algorithm*

Auto-generation of test cases based on symbolic execution and formula rewriting has the advantages of avoiding to explicitly build the states model of system under test and selecting the test cases targeting on testing abstract behaviors and different kinds of syntax structure elements of system specifications in all levels of abstract. Table 1 shows the skeleton of the algorithm of auto-generation of test cases for infinite states system. This algorithm takes the policy of depth first search, and its time complexity is determined by the size of symbolic execution model, the length of pattern of counterexample and the text complexity of trap properties.

The values of $lowbound$ and $highbound$ are set according to testing scene in practice.

Table 1. The Main Algorithm of Auto-Generation of Test Cases for Infinite States System

Input K: IOSTS struture, P: a set of trap properties;
Output TS: test suite

```
for (α in P ) {
  if(test(α,TS)) continue;// delete redundancy phenomena in TS
  M=SymSimulator(K);// M: the symbolic execution model
  flag=false;
  for (π in Path(M) ) {
    for(P=GetCEPattern(α) ) {
      constraint=LTLRW(P, π) ;// call the formula rewriting engine
      if(lowbound ≤ length ≤ highbound){
     //length is the length of pattern of counterexample
        path_condition=CMD(π);
        tc=SMT(constraint,path_condition); // call the SMT tool
        TS=TS∪{tc};
        flag=true;
        break; }
    }
    if(flag=true) break;
  }
}
```

In fact, they play the similar role as the bound in bound model checking. Further, some measures can be taken to improve the performance of the algorithm, such as different search policy, more concise pattern of counter example and special background knowledge.

Next, a simple example of the beverage machine shows the procedure of generating concrete test cases. Figure 7 defines the behavior of the beverage machine and the interaction between the system and its environment using IOSTS structure. Some internal actions are discarded for the sake of simplification. The values of some variables like $discount$ are set by the internal actions. The type of all the variables occurring in Figure 7 is the integer except $discount$.

Take the trap property $\alpha = \mathcal{G}(vQuan \geq 5 \rightarrow \mathcal{X}(sum \neq p \times discount))$ as an example. Let $mQuan = c$ and $mBeverage = s$, where c and s are the symbolic constants. A pattern of counterexample of $\alpha = \mathcal{G}(vQuan \geq 5 \rightarrow \mathcal{X}(sum \neq p \times discount))$ is graphed by Figure 8. The following table specifies $Constraint(\pi, \mathcal{P},$
$\mathcal{G}(vQuan \geq 5 \rightarrow \mathcal{X}(sum \neq p \times discount)))$ and $CMD(\pi)$. $domain(discount, p)$

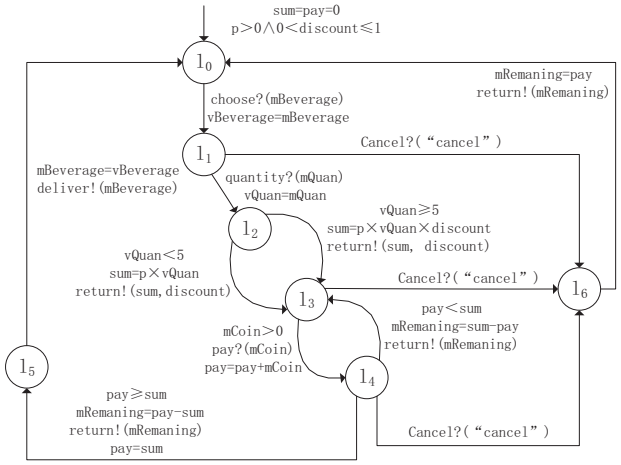

Fig. 7. IOSTS: Beverages Machine

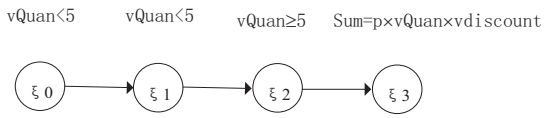

Fig. 8. The pattern of counter example of $\alpha = \mathcal{G}(vQuan \geq 5 \rightarrow \mathcal{X}(sum \neq p \times discount))$

defines the background knowledge about the variables $discount$ and p, and other constraint conditions. Using SMT tool Z3 to parse the standard SMT-LIB document and resolving $Theory(\mathcal{G}(vQuan \geq 5 \rightarrow \mathcal{X}(sum \neq p \times discount)), \mathcal{P}, \pi)$, test cases can be derived, such as $(mCoin = 5, mCoin = 1, mCoin = 2)$, where only the parameters of input and output actions are shown.

6. Prototype Tool and Experiment

6.1. *Architecture and Interface*

Based on the preceding methodology, the prototype tool is developed. Its main components include LTL formula rewriting engine, IOSTS symbolic execution simulator, interface languages analyzer, and C/C++ API of Z3 DLL, etc. The tool takes LTL and STG format as main interface languages, respectively used to specify test purposes and systems under test. STG format is a simplified version of IF specification [26], which is the input language of STG [27] with the ability of defining reactive systems. The basic grammar rules of STG format is shown in table 3, including $< system >$, $< process >$ and $< transition >$. Other elements without explicit definition can be understood from the literal meaning. The IOSTS structure can naturally interpret the semantics of the language defined by the rules in table 3. The detail is left out. Figure 9 shows the main window, where the documents

Table 2. The SMT-LIB Specification of $Theory(\mathcal{G}(vQuan \geq 5 \rightarrow \mathcal{X}(sum \neq p \times discount)), \pi, \mathcal{P})$

```
;Constraint(π, P, G(vQuan ≥ 5 → X(sum ≠ p × discount)))
(set-logic AUFNIRA)
(declare-funs((vQuan0 Int) (sum0 Int) (p0 Int) (discount0 Real)))
...
(declare-funs((vQuan3 Int) (sum3 Int) (p3 Int) (discount3 Real)))
(assert (and (and (< vQuan0 5) (< vQuan1 5)) (>= vQuan2 5) ))
(assert(= sum3 (* p3 (* vQuan3 discount3))))
; Domain(discount, p)
(assert(and (and (= discount0 0.8) (= discount1 0.8))
(and (= discount2 0.8) (= discount3 0.8) ))
(assert ( or (and (= p1 2)(= vBeverage1 1)) ( or (and (= p1 4)
(= vBeverage1 2)) (and (= p1 5)(= vBeverage1 3)))) )
(assert (and (= p2 p1) (= p3 p1)))
```

```
; CMD(π)
(declare-funs (c Int) (s Int)) (assert(> p0 0))
(assert(= discount0 0.8) )
(assert (and (= sum0 0) (= pay0 0)))
(declare-funs((vBeverage0 Int))) (declare-funs((vBeverage1 Int)
(vBeverafe2 Int) (vBeverage3 Int)))
(assert(= vBrverage1 s))(assert(= vQuan2 c))
(assert(= sum3 (* p3 (* c discount3))))(assert(>= c 5))
(assert(and (= vQuan1 vQuan0) (= vQuan3 c)))
(assert(and (=vBeverage2 s)(= vBeverage3 s)))
(assert(and (= sum1 0) (= sum2 0) ))
check-sat (get-info model)
```

respectively describe the STG format specification of windscreen wiper controller (adapted from [20]), the related test purposes and teste cases. Moreover, the tool provides the interfaces handily defining the domain knowledge of system under test, including complex initial conditions and state invariant.

6.2. *Implementation of Formula Rewriting Engine*

The formula rewriting engine is the most core component, and the implementation is simply demonstrated in the sequel.

The formula rewriting engine transforms a LTL formula α to a formula of proposition logic f according to the rewriting rules. Every formula like f, named indexed propositional formula which consists of atomic propositions with the subscript of state position,

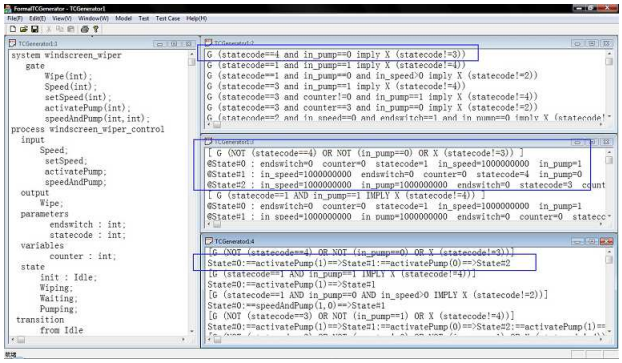

Fig. 9. The Main Window

determines some length-fixed paths with the indispensable requirement in every state for validating LTL formula α. The complete procedure is divided into two steps: construction of rewriting tree and reduction of indexed propositional formula. The algorithm of the first step can be thought of as an implementation of the tableau construction for LTL as a whole, but the output is a rewriting tree, rather than a Büchi automaton. The details are referred to [8]. Take $\alpha = a\ and\ (b\ or\ (c\ and\ XX\ d))\ and\ X\ e$ as an example, where, $a, b, \cdots$ are atomic propositions, and and, or and X logic connectives and temporal operator. Figure 10 shows the result of the first step, which is inductively defined on the following data structure:

$< system >::= system < system - id >;$ $[const < constant >]$ $[type < type >]$ $gate < gate >$ $< process >_+$
$< process >::= process < process - id >;$ $input < gate - id >^+$ $output < gate - id >^+$ $[parameters < params >^+]$ $[variable < vars >^+]$ $state < state >^+$ $transition < transition >^+$
$< transition >::= from < state - id >$ $[< guard >]$ $[< action >]$ $[< statement >]$ $to < state - id >$

Table 3. Formal Grammar of STG Format

```
struct Node {
    CArray<CProposition *,CProposition* > _stateINV;
    CArray<struct Node *,struct Node *> _nextNodeList;
    CArray<CProposition *, CProposition *> _disjuctive; }
```

The data structure is coded based on MFC and C++. Next, we expound the second step: reduction of indexed propositional formula, consequently, make the underlying meaning of the tree shown in Figure 10 clear. Let $Cond_\alpha(p)$ be the constraint condition in the form of index propositional formula reduced from the tree in Figure 10, $INV(s)$ the state invariance of state s. The computation of $Cond_\alpha(p)$ tells the essence of indexed propositional formula.

$$Cond_\alpha(p) = INV(s_0) \wedge INV(s_1)$$
$$INV(s_0) = \bigwedge_{\beta \in \_stateINV|_{s_0}} \beta \wedge \bigvee_{i=1,2} Cond_{\alpha_i}(p_i), \text{ where } \alpha_1 = b, \alpha_2 = c\ and\ XX\ d$$
$$Cond_{\alpha_1}(p_1) = \_disjuctive(1)|_{s_0} = b_{s_0}$$
$$Cond_{\alpha_2}(p_2) = \_disjuctive(2)|_{s_0} \wedge INV(s'_1) \wedge INV(s'_2) = c_{s_0} \wedge true_{s'_1} \wedge d_{s'_2}$$
$$INV(s_1) = \bigwedge_{\beta \in \_stateINV|_{s_1}} \beta = e_{s_1}$$

Summarily, $Cond_\alpha(p) = a_{s_0} \wedge (b_{s_0} \vee c_{s_0} \wedge true_{s'_1} \wedge d_{s'_2}) \wedge e_{s_1}$
$= a_{s_0} \wedge b_{s_0} \wedge e_{s_1} \vee a_{s_0} \wedge c_{s_0} \wedge true_{s'_1} \wedge d_{s'_2} \wedge e_{s_1}$

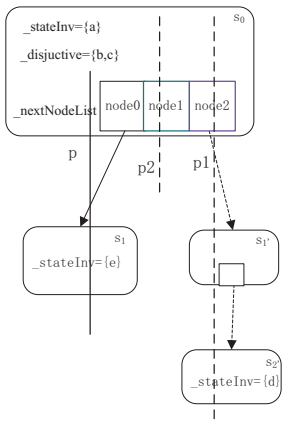

Fig. 10. The Rewriting Result of $a\ and\ (b\ or\ (c\ and\ XX\ d))\ and\ X\ e$

The first clause of DNF of $Cond_\alpha(p)$ implies that a path (s_0, s_1) is a witness of validating α if a and b are satisfied by the initial state s_0 and e is satisfied by s_1. In the same way, the second clause can be interpreted. Furthermore, a_{s_0}, b_{s_0} and $d_{s'_2}$, etc. can be simplified by omitting the state labels in the subscripts like s and s', then $Cond_\alpha(p) = a_0 \wedge b_0 \wedge e_1 \vee a_0 \wedge c_0 \wedge true_1 \wedge d_2 \wedge e_1$. Table 4 gives the algorithm of reduction

of indexed propositional formula. For the sake of space, the details of implementation of other components are not introduced.

```
Procedure sparseTree( tree, index)
   Input: Node * tree, int index
   Output: IndexedFormula path_formula // indexed propositional formula
1 Begin
2   path_formula=true_index;
3      foreach(inv∈ _stateINV)
4           path_formula=path_formula∧Index( inv,index);
5   nextNode=first(_nextNodeList);// the first of _nextNodeList
6    path_formula=path_formula∧ sparseTree(nextNode,index);
7   disjunct=false_index;
8       foreach(1 <=pos<length){// length denots the size of tail(_nextNodeList)
9           disjunct=disjunct ∨(Index(_disjuncts[pos],index+1)
10                  ∧sparseTree(_nextNodeList[pos],index+1));
11   path_formula=path_formula ∧disjunct; }
12 End
```

Table 4. The Algorithm of reduction of indexed propositional formula

6.3. *Symbolic Execution Simulator*

Symbolic execution simulator is another core component with the responsibility of deriving the symbolic execution model, and choosing the symbolic path with different strategies. The first version of prototype tool is implemented naively without aggressively reducing the amount of invalid path enumerations. The invalid path enumerations lead to needlessly time-consuming procedures for calling SMT solver. The experimental result of the subsection will show this point.

Figure 11 clearly outlines the technique route for improving the performance of symbolic execution simulator. The strategies are divided into off-line strategies and on-line ones among which, the off-line strategies preprocess the original IOSTS specification of SUT, such as elimination of infeasible path and invariant label of loop structure, and the on-line ones are used to control and optimize the procedure of enumerating paths and solving $Theory(\cdot,\cdot,\cdot)$. The loop structures are indicated by decomposing the SCC (Strong Connectivity Component) in symbolic execution model. The off-line and on-line strategies are not integrated into the prototype tool, therefore, the detail of Figure 11 is not detailedly expounded.

6.4. *Experiment*

The goal of our experiment is to show the function and performance of the prototype tool, rather than comparing the performance with the other related tools based on different

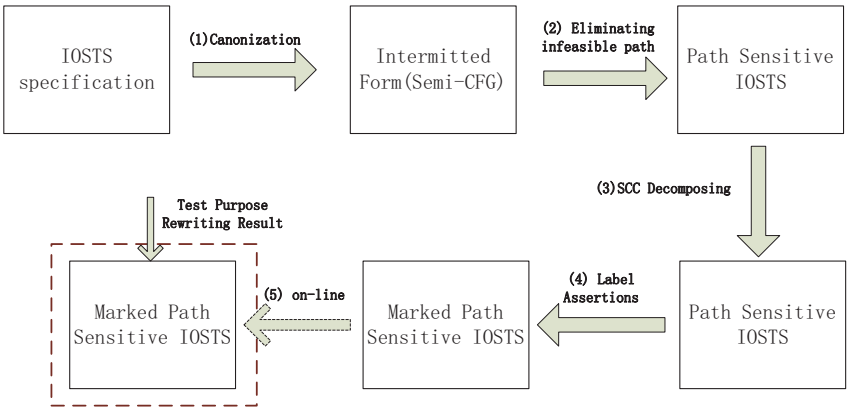

Fig. 11. The Optimization of Symbolic Execution Simulator

methodologies. The chosen systems under test include windscreen wiper controller, beverage machine and elevator controller, which are specified by STG format. Table 5 lists all considered test purposes. The experiments are performed on Microsoft Windows machine with an Inter Core 2.0 G processor and 2GB RAM. The experimental result is shown in Table 6. The second column depicts the average length of test cases of SUT (Ave. Len.), and the successive ones respectively the maximal length (Max. Len.), the average amount of the traversed transitions for generating one test case (Sum. Len.), peak memory (Peak Mem.) and execution time (Time).

The index of average amount of the traversed transitions reflects the efficiency and validity of the prototype tool. Besides Ave. Len. and Max. Len., the static structure of specification of SUT and the range of path bound have impact on the Sum. Len.. Some extreme cases can lead to the mass of invalid traversal of transitions just like what shown in the last row of Table 6. In order to improve the performance of prototype tool, some more elaborate strategies need to be taken into account: (1) Classifying the set of transitions according to domain knowledge and rewriting result of test purposes to control the procedure of the enumeration of symbolic execution paths, (2) dynamical path bound and (3) optimizing the decision procedure of Z3, etc.. Figure 11 shows more details. Moreover, the implementation of prototype tool involves frequent dynamic memory access, which impact on the overall performance, such as peak memory and execution time, therefore, the operator system-independent memory manager is a requisite. Ulteriorly,

It should be emphasized that the experiment work in the section is not adequate. We need conduct some industry-level case studies for evaluating our methodology and tool.

7. Concluded Remark and Future Work

Testing with model checkers is an important breakthrough in the automatization of testing. Due to the complexity of system and the difference between testing model and verifying model, the function of general model checking technique is very restricted in generating

Systems	Test Purpose
Windscreen wiper	[1]G(statecode==4 and in_pump==0 imply X (statecode!=3)) [2]G(statecode==1 and in_pump==1 imply X (statecode!=4)) [3]G(statecode==1 and in_pump==0 and in_speed¿0 imply X (statecode!=2)) [4]G(statecode==3 and in_pump==1 imply X (statecode!=4)) [5]G(statecode==3 and counter!=0 and in_pump==1 imply X (statecode!=4)) [6]G(statecode==3 and counter==3 and in_pump==0 imply X (statecode!=2)) [7]G(statecode==2 and in_speed==0 and endswitch==1 and in_pump==0 imply X (statecode!=1)) [8]G(statecode==2 and in_pump==1 imply X (statecode!=4))
Beverage machine	[1]G(mRemaining>0 imply mRemaining!=pay-sum) [2]G(Quan<5 and Quan!=0 imply X (sum!=price * Quan)) [3]G(Quan>=5 imply X (sum!=price * Quan * discount)) [4]G(mRemaining>0 and mCancel==1 imply X(mRemaining!=pay)) [5]G(mCoin>0 imply X(pay!=sum)) [6]G(mCoin>0 imply X (pay<=sum)) [7]G(mBeverage=1 imply X (vBeverage!=1)) [8]G(statecode==1 imply not (Quan==0 and pay==0 and sum==0))
Elevator	[1] G(statecode==2 imply 2*curr!=max) [2]G(curr==goal and statecode==3 imply X (statecode!=2)) [3]G(curr<goal and curr>5 imply X(statecode!=3)) [4]G(curr>goal and curr<5 imply X(statecode!=3)) [5]G(goal>5 and curr==goal imply X(statecode!=2)) [6]G(statecode==3 and goal==max imply curr!=goal)

Table 5. The Set of Test Purposes

Systems	Ave. Len.	Max. Len.	Sum. Len.	Peak Mem.(MB)	Time (s)
Windscreen wiper	4	7	64	1.4	0.8
Beverage machine	5	8	56	1.0	0.3
Elevator	7	13	10000	10	106

Table 6. The Exprimental Report

test cases. This paper presents the idea of auto-generation of test cases based on symbolic execution and temporal formula rewriting method and discusses the basic technique. The method has some advantages and extends testing with model checker and other testing based on model. The basic algorithms based on the methodology are implemented and some small examples are studied. Of course, the work in the paper is not enough. Our future work will focus on improving the performance of the algorithm and conducting the case studies in industry level.

References

[1] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test Generation. *IEEE Software*, 1996, 13(5):83-88.

[2] J. J. Chilenski, S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering. Journal* ,1994, 9(5):193-200.

[3] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 2004, 14(2): 102-156.

[4] D. Marinov. Automatic Testing of Software with Structurally Complex Inputs. *PhD thesis*, MIT, 2005.
[5] P. Godefroid, N. Klarlund and K. Sen. DART: directed automated random testing. *PLDI05*, 2005, 213-223.
[6] R. M. Hierons, K. Bogdanov and J. P. Bowen. Using formal specifications to support testing. *Computing Survey*, ACM, 2009, 41(2) 9: 1-76.
[7] G. Fraser, F. Wotawa and P. E. Ammann. Testing with model checking: a survey. *Software Testing, Verification and Reliability*, 2009, 19: 215–261.
[8] E. M. Clarke, O. Grumberg, D. A. Peled. *Model Checking*. MIT Press, 1999.
[9] P. Cousot , R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. *POPL* 77, 1977, 238-252.
[10] D. Dams, R. Gerth and O. Grumberg. Abstract interpretation of reactive systems. ACM Trans. Program. Lang. Syst., 1997, 19(2) : 253-291.
[11] W. Prenninger, A. Pretschner. Abstractions for model-based testing. *ENTCS* 116, 2005, 59-71.
[12] J. C. King. Symbolic execution and program testing. *Communications of the ACM, 1976*, 19(7) : 385-394.
[13] N. Tillmann, W. Schulte. Unit tests reloaded: parameterized unit testing with symbolic execution. *IEEE Software*, 2006, 23(4):38-47.
[14] M. Huth, M. Ryan. *Logic in Computer Science: Modeling and Reasoning about System*. Cambridge University Press, 1999.
[15] P. Ammann, J. Offutt and Wuzhi Xu. Coverage Criteria for state based specification. *Formal Method and Testing*, LNCS 4949, 2008, 118-156.
[16] C. Jard, T. Jeron. TGV: Theory, principles and algorithms C a tool for the automatic synthesis of conformance test cases for nondeterministic reactive systems. *Int. Journal on Software Tools for Technology Transfer*, 2005, 7(4):297-315.
[17] N. Lynch, M. Tuttle. *Introduction to IO Automata*. CWI Quarterly 3 (2), 1999.
[18] B. Jeannet, T. Jeron and V. Rusu. Model-based test selection for infinite-state. *FMCO*, LNCS 4709, 2007, 47-69.
[19] A. Pnueli. The temporal logic of programs. *Proc. of the 18th IEEE Symposium on Foundations of Computer Science*, 1977, 46-77.
[20] G. Fraser, F. Wotawa. Using model-checkers to generate and analyze property relevant test-cases. *Software Qual J*, 2008, 16:161-183.
[21] G. Rosu, K. Havelund. Rewriting-based techniques for running verification. *Journal of Automated Software Engineering*, 2005, 12(2): 151-197.
[22] G. Fraser, F. Wotawa. Using LTL rewriting to improve the performance of model checker based test case generation. *Proceedings of the Third Workshop on Advances in Model Based Testing*, 2007, 64-74.
[23] R. Nieuwenhuis, A. Oliveras and C. Tinelli. Solving SAT and SAT modulo theories: from an abstract DPLL procedure to DPLL(T). *Journal of ACM*, 2006, 53(6) : 937-977.
[24] D. Clarke, T. Jeron and V. Rusu, etc. STG: A Symbolic Test Generation Tool. LNCS2280, 2002, 152-173.
[25] Appel, Andrew W.. SSA is Functional Programming. *ACM SIGPLAN Notices*, 1998, 33 (4): 17C20.
[26] http://www-verimag.imag.fr/ async/IF/index.shtml.en.
[27] D. Clarke, T. Jeron and V. Rusu, etc. STG: A Symbolic Test Generation Tool. LNCS2280, 2002, 152-173.