Chapter 3
Formal Verification, Testing, and Inspection for Intelligent Services

Min Xu and Lisong Wang

3.1 Introduction

From self-driving cars to AlphaGo, artificial intelligence (AI) is progressing rapidly. Artificial intelligence makes our lives more convenient, but it also may bring us dangers. Just like Russia's president Vladimir Putin said: "Artificial intelligence is the future, not only for Russia, but for all humankind. It comes with enormous opportunities, but also threats that are difficult to predict. Whoever becomes the leader in this sphere will become the ruler of the world." So we should have a very convincing argument for its safety before applying an advanced intelligent system. How can we realize that argument is rigorously correct? Dijkstra said: "The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness" [8]. The answer is a mathematical proof. This is the reason why we need formal methods in AI. Formal methods are used to describe and analyze systems with a set of symbols and operations; depend on some mathematical methods and theories, such as algebra, logical, graph theory, or automata; and enhance the quality and safety of systems, so we call it formal. The properties of systems described formally can eliminate misunderstandings, and a system that satisfies its specification can be verified by formal techniques. Design or coding errors can be found in formal methods before deploying it to reduce the risk of damage of a system. This can help one to only care about the main properties of the system and can easily manage the complexity of the system. Formal methods include modeling, specification, verification and testing techniques. This chapter will present the four parts and finally give an example to illustrate.

M. Xu (✉) · L. Wang (✉)
Department of Computer Science and Technology, NUAA, Nanjing, China
e-mail: xumin@nuaa.edu.cn; wangls@nuaa.edu.cn

H. Gao, Y. Yin (eds.), *Intelligent Mobile Service Computing*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-50184-6_3

3.2 Modeling

Modeling describes some properties of the system using mathematical methods. It keeps a little original details of the system by the process of abstraction. Here, a modeling formalism and a modeling language are distinct [3]. Formalisms belong to the field of mathematics composed of abstract syntaxes and formal semantics. Languages, such as NuXMV [22], are designed to implement formalisms. A language, generally, includes a comxiler, simulator, etc.

We choose a good formalism and language for formal verification, or model checking should depend on what type of system, what properties of system, etc. For example, we can formalize discrete time systems by using push-down automata [17] and finite state machines, concurrent processes by Petri nets [21] and communicating sequential processes (CSP) [16], and compositional modeling by reactive modules [2], process algebras [13], etc. The system model is verified by a specification of the properties of the system to be verified. Different specification languages deal with different properties. For example, specification languages, such as regular expressions, state charts diagrams, computation tree logic, etc., are used for reactive systems [19]. Here we will focus on behavior over time of reactive systems because these kinds of behavior often appear in intelligent systems. We will use a Kripke structure [18] to define reactive systems' behavior.

3.2.1 Kripke Structure Modeling

A Kripke structure consists of a nonempty finite set W named state, a set of relations T is a subset of $W \times W$, and a function F, if each state with a set of propositions of this state are true, then the value of F is 1 else is 0. A path in this model M from a state $w \in W$ is an infinite sequence of states $p = w_0 w_1 w_2 \ldots w_k w_{k+1} \ldots$. such that $w_0 = w$ and $R(w_i, w_{i+1})$ holds for all $k \geq 0$. Here, let $AtomP$ be a set of atomic propositions. So a Kripke structure M over $AtomP$ can be represented as a three-tuple $M(W, T, F)$.

In this chapter, we only cover the first-order logic. So here we just use the logical connectives such as not $\neg$, and $\wedge$, or $\vee$, implies $\rightarrow$, and quantifiers such as a universal quantifier($\forall$) and an existential quantifier ($\exists$).

Let $V = \{v_1, \ldots, v_n\}$ be a variable set in a system. The range of a variable in V is a finite set D. A valuation for V is a mapxing (or function) from V to D. We can assign each variable in V a value in D to represent a state $w \in W$ of a concurrent system. So a valuation $w : V \rightarrow D$ is a state w. Generally, we can use a logical formula to represent a valuation w; in other words, the valuation makes the formula true. For example, given a variable set $\{v_1, v_2, v_3, v_4\}$ and a valuation$\{w(v_1) = 10, w(v_2) = 7, w(v_3) = 9, w(v_4) = 1\}$, the corresponding formula is $(v_1 = 10) \wedge (v_2 = 7) \wedge (v_3 = 9) \wedge (v_4 = 1)$, so, here, we use w to denote the formula. Now we know we can use a formula to represent a state in a system.

For a transition in a system, we can use an ordered pair $< w, w' >$ to denote, where w' is next state in path p, then we can use $T(w, w')$ to represent.

Now we can use the first-order formula to explain a Kripke structure $M = (W, T, F)$ that represents the concurrent system.

1. W is a set of logic formulas.
2. Letting w and w' be two states, then $T(w, w') \in T$ if w is $True$ when each $v \in V$ is assigned the value $w(v)$ and the same to w'.
3. The function $F : W \rightarrow 2^{AtomP}$ is defined so that $F(w)$ which is the subset of all atomic propositions is $True$ in w. If the range of v is $\{True, False\}$, then $v \in F(w)$ illustrates $w(v) = True$, and $v \notin L(s)$ means $w(v) = False$.

3.2.2 An Example

To show how to use the notions in this section, we give a simple example.

This is a program to implement a summing procedure from 1 to 5, which means we want to get the value of $1 + 2 + 3 + 4 + 5$. sum variable denotes the sum of the value, variable i denotes the value from 1 to 5, and pc denotes the program counter.

$p_1 : sum := 0;$
$p_2 : i := 1;$
$p_3 : \ while\ i <= 5\ do$
$p_4 : \ sum := sum + i;$
$p_5 : i := i + 1$
$p_6 : end$

So for the variables $\{sum, i, pc\}$, the range of sum is an integer, denoted by Z; the range of i is the naturals, denoted by N; and the range of pc is $\{p_1, p_2, p_3, p_4, p_5, p_6\}$, denoted by PC.

The Kripke structure $M = (W, T, F)$ in the program can be represented by the following:

1. $W = Z \times N \times PC$.
2. $T = \{((sum = 0) \wedge (PC = p_1), (sum = 0) \wedge (i = 1) \wedge (PC = p_2)), ((sum = 0) \wedge (i = 1) \wedge (PC = p_2)), (sum = 0) \wedge (i = 1) \wedge (PC = p_3)), ((sum = 0) \wedge (i = 1) \wedge (PC = p_3)), (sum = 1) \wedge (i = 1) \wedge (PC = p_4)) \ldots\}$.
3. $L((sum = 0) \wedge (PC = p_1), (sum = 0)) = \{sum = 0, PC = p_1\}, L(((sum = 0) \wedge (i = 1) \wedge (PC = p_2))) = \{sum = 0, i = 1, PC = p_2\}, L(((sum = 0) \wedge (i = 1) \wedge (PC = p_3))) = \{sum = 0, i = 1, PC = p_3\}, \ldots.$

The path in the this program that starts in an initial state is $((sum = 0) \wedge (PC = p1))((sum = 0) \wedge (i = 1) \wedge (PC = p2))((sum = 0) \wedge (i = 1) \wedge (PC = p3))((sum = 1) \wedge (i = 1) \wedge (PC = p4))((sum = 1) \wedge (i = 2) \wedge (PC = p5))((sum = 1) \wedge (i = 2) \wedge (PC = p3))((sum = 3) \wedge (i = 2) \wedge (PC = p4)) \cdots.$

3.3 Specification

Specification is usually described in some mathematical or logical methods. Generally, it uses temporal logic for hardware and software systems, which can assert what behaviors of the system over time. Using these methods to specify the design of a project can keep the consistency between the different modules of a project during its development and maintenance. Only if we give the specification of a program, we can say a program is correct or not. When a system has specification, verification of a system is necessary. A contract between the developer and the customer can be described by formal specifications.

Now we introduce briefly the classical temporal logic to describe formal specification. In a reactive system, temporal logics describe formally sequences of transitions between states. In the temporal logics, time is not mentioned explicitly. Instead, we often say that eventually or possibly some designated states have reached a formula or not and do not mention time explicitly. So eventually or possibly, they are specified using special temporal operators. These operators can be combined with logical connectives. Temporal logics provide the semantics of those operators. Here, we will use a powerful temporal logics called CTL* [7, 11].

3.3.1 The Computation Tree Logic (CTL)*

The possibility of transition of a system can form a tree structure called computation tree. Properties of computation trees can be described by CTL* formulas. The root of the tree is the initial state in Kripke structure of the system and extends to be an infinite tree. The possible executions starting from the initial state can be showed by the computation tree.

In CTL* formulas are composed of propositional variables, logical constant, connectives, temporal operators, and path quantifiers. The branching structure in the computation tree is described by the path quantifiers. Path quantifiers are **A** and **E**. **A** denotes for all computation paths which specifies all of the paths have some property. **E** denotes for some computation path which specifies some of the paths have some property. There are five basic temporal operators which describe properties of a path through the tree:

1. $\Box$ means "always" which specifies that each state on the path has the property.
2. $\Diamond$ means "eventually" which some state on the path in the future will have the property.
3. $\bigcirc$ means "next time" which specifies the next state on the path has the property.
4. $\cup$ means "until" which specifies the states all hold the first property until some state holds the second property.
5. $\vee$ means "release" which specifies when the first property is released by the states, the second property is held by all the states left.

CTL* has two types of formulas. One is state formulas which are true in a specific state. Another is path formulas which are true along a specific path. $AtomP$ is the set of atomic proposition. The following rules form the syntax of state formulas:

1. p is a state formula if $p \in AtomP$.
2. $\neg p$, $p \vee q$, and $p \wedge q$ are state formulas if p and q are also state formulas.
3. $\mathbf{A}\ p$ and $\mathbf{E}\ p$ are state formulas if p is a path formula.

Two additional rules are needed to specify the syntax of path formulas:

1. p is a path formula if p is a state formula.
2. $\neg p$, $p \vee q$, $p \wedge q$, $\Box p$, $\Diamond p$, $\bigcirc p$, $p \cup q$, and $p \vee q$ are path formulas if p and q are also path formulas.

The set of state formulas of CTL* is generated by the above rules.

Now we use a Kripke structure to define the semantics of CTL*. A Kripke structure M is (W, T, F), where W is the set of states; T is the relation which is a subset of $W \times W$, which satisfies refection, which means $\forall w \in W \Rightarrow (w, w) \in T$; and $F : W \rightarrow 2^{AtomP}$ is a function that illustrates a set of atomic propositions is true in that state. A path p in M is an infinite sequence of states, $p = w_0 w_1 \cdots w_i w_{i+1} \cdots$, $\forall i\ i \geq 0, (w_i, w_{i+1}) \in T$.

The suffix of ξ starting at w_i is denoted by ξ^i. If p, s are state formulas, the notation $M, s \vDash p$ means state s can imply state p in the Kripke structure M. Similarly, $M, \xi \vDash f$ means that along path ξ can imply the path formula f in the Kripke structure M if f is a path formula. We define inductively the relation $\vDash$ as follows: (here, p is an atomic proposition, p_1 and p_2 are state formulas, and f_1 and f_2 are path formulas):

1. $M, s \vDash p$ iff $p \in F(s)$.
2. $M, s \vDash \neg p$ iff $M, s \nvDash p$.
3. $M, s \vDash p_1 \vee p_2$ iff $M, s \vDash p_1$ *or* $M, s \vDash p_2$.
4. $M, s \vDash p_1 \wedge p_2$ iff $M, s \vDash p_1$ *and* $M, s \vDash p_2$.
5. $M, s \vDash \mathbf{E} f_1$ iff there exists a path ξ from $s \Rightarrow M, s \vDash f_1$.
6. $M, s \vDash \mathbf{A} f_1$ iff for all path ξ starting from $s \Rightarrow M, s \vDash f_1$.
7. $M, \xi \vDash p_1$ iff s is the first state of $\xi \Rightarrow M, s \vDash f_1$.
8. $M, \xi \vDash \neg f_1$ iff $M, \xi \nvDash f_1$.
9. $M, \xi \vDash f_1 \vee f_2$ iff $M, \xi \vDash f_1$ *or* $M, \xi \vDash f_2$.
10. $M, \xi \vDash f_1 \wedge f_2$ iff $M, \xi \vDash f_1$ *and* $M, \xi \vDash f_2$.
11. $M, \xi \vDash \bigcirc f_1$ iff $M, \xi^1 \vDash f_1$.
12. $M, \xi \vDash \Diamond f_1$ iff there exists a $k \geq 0 \Rightarrow M, \xi^k \vDash f_1$.
13. $M, \xi \vDash \Box f_1$ iff $\forall\, i \geq 0, M, \xi^i \vDash f_1$.
14. $M, \xi \vDash f_1 \cup f_2$ iff $\exists k\ k \geq 0,\ M, \xi^k \vDash f_2$ *and* $\forall j i \leq j < k, M, \xi^j \vDash f_1$.
15. $M, \xi \vDash f_1 \vee\ f_2$ iff $\forall j,\ j \geq 0, \forall i,\ i < j, M, \xi^i \nvDash f_1 \Rightarrow\ M, \xi^j \vDash f_2$.

Note: any other CTL* formula can be expressed by the operators $\vee, \neg, \bigcirc, \cup,$ *and* $\Diamond$.

3.3.2 Fairness

In many cases, we want to keep not only correctness but also fair computation paths. For example, we may want to consider some server protocols which can provide reliable server which have the property that no client ever continuously submits requests but never respondence. The semantics of the logic is named the fair semantics. A set of states can describe a fairness constraint by a formula of the logic. A fair path should include a state of each fairness constraint infinitely when we use sets of states to represent fairness constraints.

A fair Kripke structure can be described by four-tuple $M = (W, T, F, V)$, where W, T, and F are the same as we defined before and $V \subseteq 2^{Atom\,P}$ is a set which describe fairness constraints. Let $\xi = w_0 w_1, \cdots, w_i w_{i+1}, \cdots$ be a path in M. It can be defined by $inf(\xi) = \{w | w = w_i \text{ for infinitely many } i\}$.

If the path ξ is a fair path, it should satisfy the following condition:

$$\xi \text{ is a fair } \Leftrightarrow \forall P\ P \in F,\ inf(\xi) \cap P \neq \emptyset.$$

The semantics of an ordinary Kripke structure of CTL* can describe the semantics of an ordinary Kripke structure of CTL*. In states of the fair Kripke structure M, we use $M, s \vDash_V p$ to represent that the state formula p is true and $M, s \vDash_V f$ to represent that the path formula f is true along path ξ. The semantics of an ordinary Kripke structure can change just only clauses 1, 5, and 6 in the original semantics.

1. $M, s \vDash_V p$ iff $\exists$ a fair path ξ starting from w and $p \in F(s)$.
5. $M, s \vDash_V \Diamond f_1$ iff $\exists$ a fair path ξ from s s.t. $M, s \vDash_V f_1$.
6. $M, s \vDash_V \Box f_1$ iff $\forall$ fair path ξ starting from s s.t. $M, s \vDash g_1$.

We can use server protocols for reliable server to illustrate the use of fairness. Here we use only one fairness constraint for each client that illustrates the reliability of that server. The fairness constraint is $\forall client_i(\neg request(client_i) \vee respondence(client_i))$. So, a computation path ξ is fair $\Leftrightarrow$ $\forall client_i$ $(\neg request(client_i) \vee respondence(client_i)) \cap inf(\xi) \neq \emptyset$.

The more details can be found in [14].

3.4 Verification

In the development phase, we can evaluate our work whether it satisfies the specification of the requirements of that phase or not through verification. In other words, verification can help us to determine if the work we have done meets the requirements and specifications. At present, verification consists of two methods: one is deductive verification and the other is model checking.

3.4.1 Deductive Verification

Using axioms and rules to prove whether systems are correct or not is called deductive verification. Most of computer scientists believe deductive verification is very important. So we have used it in many fields of software development. Deductive verification can be automated to prove infinite-state systems. However, deductive verification needs much knowledge of mathematics; thus, only experts can utilize it. Moreover, deductive verification is time-consuming to prove some little scale problem. Finally, when we cannot prove the system is correct, we cannot also prove that the system is wrong. So here we will only emphasize model checking.

3.4.2 Model Checking

We can use a technique of model checking for automation to verify finite state systems. General approaches search exhaustively the state of system to check if specification of the system has been met or not. This procedure always will stop and give us an answer that the system is true of false if we have sufficient memory and time.

We can easily describe the model checking problem with Kripke structure and a temporal logic formula. Using temporal logic formula p to represent some specification and Kripke structure $M = (W, T, F)$ to describe the system, then we can check $M, s \vDash p$ or not. There are initial states in the system, and model checking will check whether the system satisfies these specifications according to the initial states.

3.4.3 Symbolic Model Checking

In 1987, McMillan [4, 20] verified much larger systems through the state transition graphs of symbolic representation. They utilized Bryant's ordered binary decision diagrams (OBDD) [5] to construct the new symbolic representation. OBDDs is a very efficient algorithm which provides a canonical form for Boolean formulas. Symbolic representation can describe the rules in the state space according to the specifications, so it can verify systems whose scale is larger than algorithms described by handled explicit state. By the new representation, the original CTL model checking algorithm [7] can verify some systems whose states are more than 10^{20} states.

3.4.3.1 Fixed-Point Representations

A Kripke structure $M(W, T, F)$ is finite, where W is a finite set and the power of W is denoted by $P(W)$. Let $\pi : P(W) \rightarrow P(W)$ be a transform or a function.

1. π is monotonic if $R \subseteq S$ implies $\pi(R) \subseteq \pi(S)$;
2. π is $\cup$ continuous if $R_1 \subseteq R_2 \subseteq \cdots \subseteq R_i \subseteq R_{i+1} \subseteq \cdots \Rightarrow \pi(\cup_i(R_i)) = \cup_i(R_i)$;
3. π is $\cap$ continuous if $P_1 \supseteq P_2 \supseteq \cdots \supseteq P_i \supseteq P_{i+1} \supseteq \cdots \Rightarrow \pi(\cap_i P_i) = \cap_i \pi(P_i)$;

$\pi^i(W) = \underbrace{\pi(\cdots(\pi(\pi(W))))}_{i}$.

The recursive definition is the following:

$$\begin{cases} \pi^0(W) = W & n = 0 \\ \pi^{i+1}(W) = \pi(\pi^i(W)) & n = i \end{cases}$$

There are least fixed point denoted by $least(\pi(W))$ and greatest fixed point denoted by $great(\pi(W))$ on a monotonic function, just like the following:

$$\begin{cases} least(\pi(W)) = \cap\{W | \pi(W) \supseteq W\} \\ great(\pi(W)) = \cup\{W | \pi(W) \subseteq W\} \end{cases}$$

When π is monotonic and *cup* or *cap* is continuous, the definition is as follows:

$$\begin{cases} least(\pi(W)) = \cap_i \pi^i(True) & \cap\, continuous \\ great(\pi(W)) = \cup_i \pi^i(False) & \cup\, continuous \end{cases}$$

We can get some lemmas defined on finite Kripke structures [7, 12].

Lemma 1 *π is $\cup$ continuous and $\cap$ is continuous if W is finite and π is monotonic.*

Lemma 2 *$\forall\, i\ \pi^i(False) \subseteq \pi^{i+1}(False)$ and $\pi^i(True) \supseteq \pi^{i+1}(True)$ if π is monotonic.*

Lemma 3 *$\exists n_0 \in Z$, $s.t.\ least(\pi(W)) = \pi^{n_0}(False)$. and $\exists m_0 \in Z, s.t.\ great(\pi(W)) = \pi^{m_0}(Ture)$ If W is finite and π is monotonic, where Z is an integer set.*

We can use a least or greatest fixed point to define CTL operators when we use $\{w|M, w \vDash p\}$ in $P(W)$ to present CTL formula g [10].
g_1, g_2 are CTL formulas.

- $\mathbf{A}\Diamond g_1 = least(g_1) \vee \mathbf{A} \bigcirc W$
- $\mathbf{E}\Diamond g_1 = least(g_1) \vee \mathbf{E} \bigcirc W$

- $\mathbf{A}\Box g_1 = great(g_1) \wedge \mathbf{A} \bigcirc W$
- $\mathbf{E}\Box g_1 = great(g_1) \wedge \mathbf{E} \bigcirc W$
- $\mathbf{A}[g_1 \cup g_2] = least(g_2) \vee (g_1 \wedge \mathbf{A} \bigcirc W)$
- $\mathbf{E}[g_1 \cup g_2] = least(g_2 \vee (g_1 \wedge \mathbf{E} \bigcirc W)$
- $\mathbf{A}[g_1 \vee g_2] = great(g_2) \wedge (g_1 \vee \mathbf{A} \bigcirc W)$
- $\mathbf{E}[g_1 \vee g_2] = great(g_2) \wedge (g_1 \vee \mathbf{E} \bigcirc W)$

We can easy to know least fixed points correspond to final event and greatest fix points correspond to hold properties forever. Thus, $\mathbf{A}\Diamond g_1$ has a least fixed point and $\mathbf{A}\Box g_1$ has a greatest fixed point.

3.4.3.2 Symbolic Model Checking for CTL

We use OBDD to represent the Kripke structures and the logic of quantified Boolean formulas (QBF) [1, 15] to denote operations on Boolean formulas.

The set of formulas $QBF(V)$ can be defined as follows:
$V = \{v_0, \cdots, v_{n-1}\}$ is a set to represent propositional variables:

- $v \in V$ is a formula;
- $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ are formulas if f_1 and f_2 are also formulas,
- $\exists v f$ and $\forall v f$ are formulas, if f is a formula and $v \in V$.

We use a function $\pi : V \rightarrow \{0, 1\}$ to represent a truth assignment. Here we introduce the notation $\pi_{(a \rightarrow v)}$ to represent the truth assignment when $a \in \{0, 1\} = V$. So it can be defined by the following:

$$\pi_{(a \rightarrow v)}(v') = \begin{cases} a & \text{if } v = v' \\ \pi(v') & \text{otherwise.} \end{cases}$$

$\pi \vDash f$ denotes that f is true if $\pi(f) = 1$, where f is a formula and π is a truth assignment. The definition of $\vDash$ is the following:
f, f_1, f_2 are formulas in $QBF(V)$.

- $\pi \vDash v \Leftrightarrow \pi(v) = 1$
- $\pi \vDash \neg f \Leftrightarrow \pi \nvDash f$
- $\pi \vDash f_1 \vee f_2 \Leftrightarrow \pi \vDash f_1 \vee \pi \vDash f_2$
- $\pi \vDash f_1 \wedge f_2 \Leftrightarrow \pi \vDash f_1 \wedge \pi \vDash f_2$
- $\pi \vDash \exists v f \Leftrightarrow \pi_{(a \rightarrow 0)} \vDash f \vee \pi_{(a \rightarrow 1)} \vDash f$, and
- $\pi \vDash \forall v f \Leftrightarrow \pi_{(a \rightarrow 0)} \vDash f \wedge \pi_{(a \rightarrow 1)} \vDash f$.

Relational product operations exist in which quantifiers can be represented by $\exists x[f_1(x, y) \wedge f_2(x, y)]$ generally.

3.4.3.3 Algorithm of Model Checking

The algorithm of model checking can be implemented by a procedure or function name $module\_checker$. The input of $module\_checker$ is the CTL formula to be checked, and the output of $module\_checker$ is an OBDD [9]. The definition is the following: a is a proposition f, f_1, f_2 are formulas.

- $module\_checker(a) = \{v \in V | \pi(v) = a\}$,
- $module\_checker(f_1 \wedge f_2) = module\_checker(module\_checker(f_1) \wedge module\_checker(f_2))$,
- $module\_checker(\neg f) = \neg module\_checker(module\_checker(f)$,
- $module\_checker(\mathbf{E} \bigcirc f) = module\_checker\mathbf{E} \bigcirc (module\_checker(f))$,
- $module\_checker(\mathbf{E}[f_1 \cup f_2]) = module\_checker\mathbf{E} \cup (module\_checker(f_1), module\_checker(f_2))$,
- $module\_checker(\mathbf{E} \Box f) = module\_checker\mathbf{E} \Box (module\_checker(f))$.

$module\_checker\mathbf{E} \bigcirc$ means if the state has a successor in f which is true, the formula $\mathbf{E} \bigcirc f$ is true. In other words,

$$module\_checker\mathbf{E} \bigcirc (f(v)) = \exists v'[f(v') \wedge T(v, v')].$$

Where $T(v, v')$ is a relation in OBDD.
$module\_checker\mathbf{E} \cup$ can use the least fixed point to compute

$$\mathbf{E}[f_1 \cup f_2] = least(f_2) \vee (f_1 \wedge (\mathbf{E} \bigcirc W)).$$

The other formula can be processed similarly.

3.4.4 Fairness in Model Inspecting

For fairness, we assume to use CTL formulas $Con = \{c_1, c_2, \cdots, c_n\}$ to represent fairness constraints. We can define procedure $module\_checker Fair$ to check whether formulas f of specifications satisfy the Con or not.

Fairness constraints $\mathbf{E} \Box\, c$ means there is a path which holds all c from the start to infinity. And the state W makes c true which has the following properties:

1. $\forall w \in W \Rightarrow f = True$,
2. $\exists\, p(f = True\ \forall w \in p)$

where p is a path in CTL.
By means of a fixed point, we can represent symbolic model checking as follows:

$$\mathbf{E} \Box\, f = great(f) \wedge \bigwedge_{k=1}^{n} \mathbf{E} \bigcirc\ \mathbf{E}[f \cup (W \wedge P_k)]$$

So under fairness constraint, $module\_checker Fair E\Box$ can be computed as follows:

$$module\_checker Fair E\Box(f) = great(f) \wedge \bigvee_{i=1}^{n} \mathbf{E} \bigcirc (\mathbf{E} \cup (f, W \wedge p_i))$$

$module\_checker Fair E\bigcirc$, $module\_checker Fair E\cup$ can be computed similarly.

3.5 Testing

Software testing describes a process used to facilitate the qualification, integrity, security, and quality of software. In other words, software testing is a review or comparison process between actual output and expected output according to the specification. Testing methods can be applied on an actual system directly rather than a model and can deal with infinite-state systems. But testing cannot cover all the possible execution cases of a system, just depends on some criteria.

3.5.1 Software Testing Method

The testing methods mainly include white box testing, black box testing, and gray box testing from the perspective of whether you care about the internal structure or specific implementation of the software. White box testing methods mainly include code inspection method, static quality measurement method, logical coverage method, basic path test method, domain test, symbol test, path coverage, etc. Black box testing methods mainly include equivalence class division method, boundary value analysis method, error inference method, causality diagram method, decision table drive method, orthogonal experiment design method, function diagram method, etc.

The test methods can be divided into static tests and dynamic tests from the perspective of whether to execute the program. Static tests include code inspection, static structural analysis, code quality metrics, etc. Dynamic testing consists of three parts: constructing test cases, executing programs, and analyzing the output of programs.

Testing has different stages as follows:

- Unit (module) testing. The unit test is mainly to test the module of the software and find out that the actual function of the module does not conform to the specification and coding errors. Because the module is small in scale, single in function, and simple in structure, the test methods adopted are static test method and white box test.

- Integration testing. Integration testing is the second phase of software testing. At this stage, modules that have been assembled in strict accordance with program design requirements and standards are usually tested simultaneously to clarify the correctness of the assembly of the program structure and to discover problems related to the interface. At this stage, a combination of white box and black box is generally used for testing to verify the rationality of the design at this stage and the realization of required functions.
- System testing. System tests check whether the system meets the software requirements. The main test content in this phase includes robustness test, performance test, function test, installation or anti-installation test, user interface test, stress test, reliability and safety test, etc. The system test mainly uses the black box method for testing.
- Validation testing. Validation testing is the testing work to be performed before the software product is put into actual execution. Compared with system test, validation testing differs from testers only, and validation test is performed by the user. The main goal of validation testing is to show users that the software developed meets predetermined requirements and relevant standards and to verify the effectiveness and reliability of the software's actual work and to ensure that users can successfully use the software to complete established tasks and functions.

Now we just briefly introduce what are white box testing and black box testing as end of the section.

Black box testing, as its name implies, simulates a software testing environment as an invisible "black box." Observe the data output through data input and check whether the internal function of the software is normal. When the test is unfolded, data is entered into the software and waits for data to be outputted. If the data output is consistent with the expected data, it proves that the software passes the test. If the data is different from the expected data, even if the difference is small, it also proves that there is a problem in the software program, and it needs to be resolved as soon as possible.

Compared with black box testing, white box testing has a certain degree of transparency. The principle is to debug the internal working process of the product according to the software's internal applications and source code. During the testing process, it is often analyzed in collaboration with the internal structure of the software. The biggest advantage is that it can effectively solve the problems of the internal applications of the software. It is often combined with the black box test method during the test. The test method can also effectively debug such situations. Among them, the judgment test is one of the most important test program structures in the white box test method. Such a program structure, as an overall implementation of the program logic structure, has a more important role for the program test. This type of testing method covers all types of code in the program, and covers a wide range, which is suitable for multi-type programs. In actual detection, the white box test method is often used in combination with the black box test method. Take the unknown error detected in the dynamic detection method as an example. First, use

the black box test method. If the program input data is the same as the output data, the internal data is not. If there is a problem, it should be analyzed from the code side. If there is a problem, use the white box test method to analyze the internal structure of the software until the problem is detected and amended in time.

3.6 Model Checking in Practice

3.6.1 The NuXMV Model Checker

NuSMV [6] is a classic model detection tool, which implements symbol model detection technology efficiently. NuSMV uses BDD to alleviate and achieve the state explosion problem, has a good software architecture, and is easy to customize and extend. NuXMV inherits all the functionalities of NuSMV and extends to specify the infinite-state systems.

In this part, we will introduce the NuXMV grammar and how to use it to check an intelligent system which should satisfy some properties.

3.6.2 Grammar of NuXMV

NuSMV uses its language to describe Kripke's structure and special verification specifications. The Kripke structure is often called finite-state machine (FSM) in NuXMV [22]. NuXMV has two useful expressions: $init$ expression and $next$ expression. The $init$ expression is used to describe the initial state, and the $next$ expression is used to describe the transition relationship. Programs written in NuXMV are often called smv programs. The smv program consists of modules. The types of state variables are very similar to other computer language, such as C, Java, Python, etc. Here do not provide to describe detailed, these can be found in the paper [22].

3.6.2.1 MODULE

A module consists of a module name and a module definition, and a module definition consists of a parameter and a body. The main part of the module is divided into three categories: Variables, Constraint, and Specification. The Variables section is used to describe the state set of the Kripke model; the Constraint section is used to describe the transition relationship of the Kripke model and some restrictions on the model; and the Specification section is used to describe the specification of special verification. The smv program must have at least one module called main, and the

main module cannot have formal parameters. Multiple module descriptions can be used to describe the FSM and then combined into a whole FSM.

The following is an example 1:

```
1  MODULE main
2  VAR
3   s : boolean;
4  ASSIGN
5   init(s) := FALSE;
6   next(s) := TRUE;
7  CTLSPEC
8         EX s = TRUE
```

where "VAR i: boolean" denotes i is a variable and its type is boolean, "ASSIGN" belongs to the constraint which describes how a system works, 'init(i) := FALSE' denotes i initial value is FALSE, "next(i) := TRUE" denotes its value in the next state is TRUE, "CTLSPEC" introduces a formula in CTL to describe specification of the system, and **EX** means the system exists in the next state whose value is TRUE.

3.6.2.2 Types and Variables

NuXMV provides many data types, such as Boolean, integer, enumeration, word, and arrays types. Variables in NuXMV are declared by **VAR** which describe the states of a system. The definition of the variable of form is $< state\_variable\_name >:< data\_type >;$, where $< state\_variable\_name >$ denotes the name of the variable and $< data\_type >;$ denotes the type of data; in general, we will add a semicolon at the end of the statement. The second and third lines in example 1 show how to apply (VAR) to define variables.

Some data type are introduced by the following:

- Boolean Type: symbolic values **FALSE** and **TRUE**,
- Enumeration Types: full enumerations of all the values, for example, {SUCCESS, 1, 5, PASS},
- Integer: positive or negative integer number,
- Real: the rational numbers,
- Array: for example, array 0.5 of integer: 0 is lower bound, 3 is upper bound for the index, and integer is the type of the elements in the array,
- ...

NuXMV assigns values to variables by the keyword **ASSIGN**. We can use **init** to assign the initial values of the state in the system and **next** to assign the value of the next state of the system. Lines 4, 5, and 6 in example 1 show how to apply **ASSIGN** to assign values to variables.

There are two important expressions, Case Expression and If-Then-Else Expression, in NuXMV.

The syntax of If-Then-Else Expression is $< bool\_expr >? < expr1 >:< expr2 >$, where $< bool\_expr >$ must be a Boolean expression and $< expr1 >$ and $< expr2 >$ are any expressions; if $< bool\_expr >$ is true, then we can get the value of $< expr1 >$; else get the value of $< expr2 >$.
The syntax of Case Expression is the following:

```
case
condition1 : expression1;
condition2 : expression2;
...
1 :         expression N
esac
```

When the first value of condition k is true, the Case Expression returns the value of the kth expressionK on the right-hand side of ": ," where "1" means other cases.

3.6.2.3 Specifications

NuXMV provides linear temporal logic (LTL), computation tree logic (CTL), and property specification language (PSL) to check whether the system satisfies the specification or not. NuXMV use **CTLSPEC**, **LTLSPEC**, and **PSLSPEC** to insert formulas of specifications to check.
NuXMV represent differently five basic temporal operators in CTL in Sect. 3.3.1. Refer to the paper [22] for details.

1. **G** denotes $\Box$,
2. **F** denotes $\Diamond$,
3. **X** denotes $\bigcirc$,
4. **U** denotes $\cup$.

Lines 7 and 8 in example 1 show how to apply **CTLSPEC** to check.

3.6.2.4 Module and Program

There must be a module named main in the program of NuXMV, which is just the main function in C language program. The other modules, in the program of NuXMV, are similar to general functions. Now we can run example 1. Because the example satisfy the specification **EX**s = **TRUE**, means the system should exist a next state is TRUE, obviously, it is correct. So we can obtain the result as following:

```
-- specification EX s = TRUE  is true
```

If we change the specification to the **EG** s = **TRUE** that means there exist a path holds s always true in the all states of the path, here, just a state s and the initial value of s is **FALSE**. So it is impossible in the example; the NuXMV gives the counterexample as follows:

```
Trace Type: Counterexample
  -> State: 1.1 <-
    s = FALSE
```

3.6.3 An Example

Now general intelligent services are very complex and concurrent and to know that the system has properties that we need, formal methods are a good way to help us. Here an example has been illustrated.

Now we consider that there are two Agents that can entertain in one place, but this place can only be accessed by one Agent at any time. Now we design a program that allows both Agent1 and Agent2 to have the opportunity to enjoy this place. Let each Agent to have four states: sleeping, trying, enjoying, and exiting. The sleeping state indicates that the Agent is idle, the trying state indicates that the Agent wants to enter this place to entertain, the enjoying state indicates that the Agent is entertaining, and the exiting state indicates that the Agent wants to leave this place. If only one Agent wants to go to this place while trying, then it can go. When both Agents are in the trying state and both want to go, then set a variable *semaphore* type to Boolean. If the *semaphore* value is **FALSE**, Agent1 can go in and change the value of semaphore to **TRUE**. If the semaphore value is **TRUE**, Agent2 can go in and change the value of semaphore to **FALSE**. To describe the syntax of NuXMV in more detail, consider the following program:

```
1  MODULE main
2  VAR
3    state_agent1: {sleeping, trying, enjoying,exiting};
4    state_agent2: {sleeping, trying, enjoying,exiting};
5    semaphore : boolean;
6    ag1: process agent(state_agent1, state_agent2, semaphore, FALSE);
7    ag2: process agent(state_agent2, state_agent1, semaphore, TRUE);
8
9  ASSIGN
10   init(semaphore) :=  FALSE;
11
12 CTLSPEC
13   EF(( state_agent1 = enjoying) & (state_agent2 = enjoying))
14 CTLSPEC
15   AG(( state_agent1 = trying) -> AF(state_agent1 = enjoying))
16 CTLSPEC
17   EX( state_agent1 = trying) & EX(state_agent2 = trying)
18
19 MODULE  agent(state0, state1, semaphore, semaphore0)
20 ASSIGN
21    init(state0) := sleeping;
22    next(state0) :=
23        case
24         (state0 = sleeping) : {trying, sleeping};
25         (state0 = trying) & (state1 = sleeping) : enjoying;
26         (state0 = trying) & (state1 = trying) &
```

```
27          (semaphore = semaphore0) : enjoying;
28          (state0 = enjoying) : {enjoying, exiting};
29          (state0 = exiting) : {sleeping};
30          TRUE : state0;
31        esac;
32
33      next(semaphore):=
34        case
35          (semaphore = semaphore0) & (state0 = enjoying): !semaphore;
36          TRUE : semaphore;
37        esac;
38
39  FAIRNESS
40    running
```

Here we define two modules: one is the main module *main* and the other is the *agent* module. The *main* module calls the *agent* module to simulate the two Agents. In the part of defining variables, lines 2 and 3 define the state space of Agent1 and Agent2, $\{sleeping, trying, enjoying, exiting\}$, and the fourth line defines the variable semaphore and initializes the value of the variable semaphore. FALSE, as shown in lines 9 and 10, that is, when both Agents are in the trying state at the beginning, let Agent1 enter first. The *agent* module is instantiated in the **VAR** statement, as shown in lines 6 and 7, mainly to instantiate two Agents, because to run concurrently, the keyword **process** is used. From lines 12 to line 17, we define three specifications using CTL. Line 13 checks whether the model allows both Agents to be in the enjoying state. Line 15 is to check whether Agent1 must be able to enter the enjoying state. Line 17 is to check that both Agent1 and Agent2 are in the trying state.

When the Agent module is called, the four parameters are given such as $state0$, $state1$, $semaphore$, and $semaphore0$. $state0$ indicates the current state of the Agent. $state1$ indicates the state of another Agent at the moment. When the Agents are in the $trying$ state, in which Agents $semaphore$ and $semaphore0$ are equal, that Agent can be in the $enjoying$ state. In the **ASSIGN** statement part, we give the Agent the initial state of $sleeping$, as shown in line 21. The case statements in lines 22–30 give the value of the next state variable $state0$ of the Agent. For example, when the status of the Agent is $enjoying$, the next status can be either $enjoying$ or $exiting$, as shown in line 27. The case statement in lines 32–36 gives the next value of the variable $semaphore$. The last **FAIRNESS** statement $running$ is to allow the Agent process to run indefinitely.

When NuXMV is run on the program, the following output is produced:

```
1 specification EF (state_agent1 = enjoying & state_agent2
 = enjoying) is false
2  -- as demonstrated by the following execution sequence
3  Trace Description: CTL Counterexample
4  Trace Type: Counterexample
5    -> State: 1.1 <-
6      state_agent1 = sleeping
7      state_agent2 = sleeping
8      semaphore = FALSE
```

```
9 specification AG (state_agent1 = trying -> AF state_agent1
  = enjoying) is false
10  -- as demonstrated by the following execution sequence
11  Trace Description: CTL Counterexample
12  Trace Type: Counterexample
13    -> State: 2.1 <-
14      state_agent1 = sleeping
15      state_agent2 = sleeping
16      semaphore = FALSE
17    -> Input: 2.2 <-
18      _process_selector_ = ag2
19      running = FALSE
20      ag2.running = TRUE
21      ag1.running = FALSE
22    -> State: 2.2 <-
23      state_agent2 = trying
24    -> Input: 2.3 <-
25    -> State: 2.3 <-
26      state_agent2 = enjoying
27    -> Input: 2.4 <-
28      _process_selector_ = ag1
29      ag2.running = FALSE
30      ag1.running = TRUE
31    -- Loop starts here
32    -> State: 2.4 <-
33      state_agent1 = trying
34    -> Input: 2.5 <-
35      _process_selector_ = ag2
36      ag2.running = TRUE
37      ag1.running = FALSE
38    -- Loop starts here
39    -> State: 2.5 <-
40    -> Input: 2.6 <-
41      _process_selector_ = ag1
42      ag2.running = FALSE
43      ag1.running = TRUE
44    -- Loop starts here
45    -> State: 2.6 <-
46    -> Input: 2.7 <-
47      _process_selector_ = main
48      running = TRUE
49      ag1.running = FALSE
50    -> State: 2.7 <-
51  specification EX (state_agent1 = trying & EX state_agent2 =
trying) is true
```

We can note from the results of the above result of the program running. The first line illustrates that both Agents are in the *enjoying* state which is false, which means that there is only one in *enjoying* state at any time. This is what we expect. Line 51 shows that both Agents are in the *trying* state. This is true, which indicates

that two Agents can compete to enter *enjoying* at the same time. It is also allowed by us, so it is also correct. From lines 9 to 51, there are counterexamples where Agent1 cannot enter the *enjoying* state when Agent1 is in the *trying* state. The reason is that on line 26, when Agent2 enters the *enjoying* state, it keeps this state forever, so Agent1 cannot enter. So we add the FAIRNESS statement to the main module, and the program is shown below.

```
1  MODULE main
2  VAR
3    state_agent1: {sleeping, trying, enjoying,
     exiting};
4    state_agent2: {sleeping, trying, enjoying,
     exiting};
5    semaphore : boolean;
6    ag1: process agent(state_agent1,
     state_agent2, semaphore, FALSE);
7    ag2: process agent(state_agent2,
     state_agent1, semaphore, TRUE);
8
9  ASSIGN
10   init(semaphore) :=  FALSE;
11
12 FAIRNESS
13   !(state_agent1 = enjoying)
14 FAIRNESS
15   !(state_agent2 = enjoying)
16
17 CTLSPEC
18   EF(( state_agent1 = enjoying) &
     (state_agent2 = enjoying))
19 CTLSPEC
20   AG(( state_agent1 = trying) ->
     AF(state_agent1 = enjoying))
21 CTLSPEC
22   EX( state_agent1 = trying) &
     EX(state_agent2 = trying)
23
24
25 MODULE  agent(state0, state1,
   semaphore, semaphore0)
26 ASSIGN
27    init(state0) := sleeping;
28    next(state0) :=
29        case
30         (state0 = sleeping) :
```

```
            {trying, sleeping};
31          (state0 = trying) &
            (state1 = sleeping) : enjoying;
32          (state0 = trying) &
            (state1 = trying) &
33          (semaphore = semaphore0)
            : enjoying;
34          (state0 = enjoying) :
            {enjoying, exiting};
35          (state0 = exiting) :
            {sleeping};
36          TRUE : state0;
37         esac;
38
39      next(semaphore):=
40        case
41          (semaphore = semaphore0) & (state0
            = enjoying):  !semaphore;
42          TRUE : semaphore;
43        esac;
44
45  FAIRNESS
46    running
```

We added the fairness constraint given by the **FAIRNESS** statement in lines 12 and 15, so that no Agent can always be in the *enjoying* state. Let the NuXMV program run again, and the result is as follows.

```
1  specification EF (state_agent1 = enjoying &
   state_agent2 = enjoying) is false
2  -- as demonstrated by the following
   execution sequence
3  Trace Description: CTL Counterexample
4  Trace Type: Counterexample
5    -> State: 1.1 <-
6      state_agent1 = sleeping
7      state_agent2 = sleeping
8      semaphore = FALSE
9  specification AG (state_agent1 = trying ->
   AF state_agent1 = enjoying)  is true
10  -- specification (EX state_agent1 =
    trying & EX state_agent2 = trying)  is true
```

From line 9, we can see that$\mathbf{AG}((state_agent1 = trying) -> \mathbf{AF}(state_agent1 = enjoying))$ is now satisfied.

References

1. V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley Longman Publishing Co., Inc., Boston, 1974)
2. R. Alur, T. Henzinger, Reactive modules. Form. Methods Syst. Des. **15**, 7–48 (1999)
3. D. Broman, E. Lee, S. Tripakis, M. Törngren, Viewpoints, formalisms, languages, and tools for cyber-physical systems, in *6th International Workshop on Multi-paradigm Modeling (MPM'12)* (2012)
4. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98**(2), 142–170 (1992)
5. R.E. Bryant, Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
6. A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: an open source tool for symbolic model checking, in *CAV*, ed. by E. Brinksma, K.G. Larsen. LNCS, vol. 2404 (Springer, 2002), pp. 359–364
7. E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in *Logic of Programs: Workshop*, Yorktown Heights, NY, May 1981. LNCS, vol. 131 (Springer, 1981)
8. E.W. Dijkstra, The humble programmer. Commun. ACM **15**(10), 859–866 (1972)
9. E.M. Clarke Jr., Orna Grumberg, Lucent Technologies, *Model Checking* (MIT Press, Cambridge, MA, 1999)
10. E.A. Emerson, E.M. Clarke, Characterizing correctness properties of parallel programs using fixpoints, in *Automata, Languages and Programming*. LNCS, vol. 85 (Springer, 1980), pp. 169–181
11. E.A. Emerson, J.Y. Halpern, "Sometimes" and "Not Never" revisited: on branching time versus linear time. J. ACM **33**, 151–178 (1986)
12. E.A. Emerson, C.-L. Lei, Efficient model checking in fragments of the propositional mu-calculus, in *LICS86* (1986), pp. 267–278
13. W. Fokkink, *Introduction to Process Algebra* (Springer, Heidelberg, 2000)
14. N. Francez, *Fairness* (Springer, 1986)
15. M.R. Garey, D.S. Jolmson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (W. H. Freeman and Company, San Francisco, 1979)
16. C. Hoare, *Communicating Sequential Processes* (Prentice Hall, New York, 1985)
17. J. Hopcroft, R. Motwani, J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd edn. (Addison-Wesley, Reading, 2006)
18. G.E. Hughes, M.J. Creswell, *Introduction to Modal Logic* (Methuen and Co. Ltd., London, 1968/1977)
19. Z. Manna, A. Pnueli, *Temporal Verifications of Reactive Systems-Safety* (US, Springer, New York, 1995)
20. K.L. McMillan, *Symbolic Model Checking* (Kluwer Academic Publishers, Norwell, 1993)
21. W. Reisig, *Petri Nets: An Introduction* (Springer, Heidelberg, 1985)
22. https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf