Model Checking is Refinement—
Computational Temporal Logic is Equivalent to Failure Trace Testing

by

Zhiyu ZHANG

A thesis submitted to the
Department of Computer Science
in conformity with the requirements for
the degree of Master of Science

Bishop's Unversity
Sherbrooke, Quebec, Canada
December 2008

Abstract

The two major systems of formal verification are model checking and algebraic model-based testing. Model checking is based on some form of temporal logic such as linear temporal logic (LTL) or computational temporal logic (CTL). The most powerful and realistic logic being used is CTL, which is capable of expressing most interesting properties of processes such as liveness and safety. Model-based testing is based on some operational semantics of processes (such as traces, failures, or both) and their associated preoders. The most fine-grained preorder beside bisimulation (of theoretical importance only) is the one based on failure traces. We show that these two most powerful variants are equivalent. That is to say, we show that for any failure trace test, there exists a CTL formula equivalent to it (meaning that a system passes the test if and only if the system satisfies the formula and the other way around). When we specify the system, we can use temporal logic formulas such as CTL formulas to express the properties of it. We can also use algebraic method such as labelled transition system or finite state automaton to describe the system's desired behaviour. If parts of a larger system are specified by these two means at the same time, combining the result of doing model checking and the result of applying model-based testing won't be ideal, satisfactory and even correct. In this case, we can convert one specification to the form of the other. We then do a formal verification for the whole system. The result will be more convincing and correct.

Key Words

Model checking, model-based testing, stable failure, failure trace, failure trace preorder, temporal logic, computational tree logic, labelled transition system, Kripke structure.

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I want to thank the members of the thesis committee Marc Frappier, Lin Jensen and Dimitri Vouliouris for useful advice in preparing the final manuscript.

I am deeply indebted to my supervisor Professor Dr. Stefan D. Bruda whose help, stimulating suggestions and encouragement helped me in all the time of the research and the writing of this thesis. Many thanks again for his support and advice.

Finally, I would like to give me special thanks to my parents whose patient love enabled me to complete this thesis.

Table of Contents

List of Figures and Illustrations

Chapter 1
Introduction

Computing systems are becoming more and more important in our daily life. Our cell phones, cars and planes are all embedded with computing systems. Security protocols for communication enable electronic commerce, internet and telephone networks. Highway and air traffic control systems are also using hardware and software systems. Not only critical systems (whose failure causes death or loss of property or both) but also family equipment is nowadays digital. The assurance of correctness for hardware and software systems is therefore very important and often no failure is acceptable. As a matter of fact, we are becoming more and more reliant on computing devices than ever before. The importance of developing methods of checking and ensuring correctness is critical.

Verification is the process of proving or disproving the correctness of a system with respect to a certain specification or property. There are three kinds of methods of verification [4, 6]: simulation (or testing) [21, 22], formal testing [11, 13, 14, 20, 23] and model checking [5, 9, 10, 16, 17, 19, 27, 28, 29, 33]. Simulation is a traditional, non-formal method of verification and it is still the main method. The idea of simulation is to inject signals into the system, compute and then observe the resulting signals. We judge whether the results are the same as expected. The disadvantage of simulation is that it cannot check all the possible situations, which means it can disprove but cannot prove. Ergo, simulation is suitable for finding a lot of errors in the beginning stages of the verification of processes. Testing is a method in which test cases are derived systematically from the specification. We run all the test cases against the system under test and observe the final results of the run. Model checking is a method for formally verifying finite-state systems. In model checking, the specifications of the system are described by temporal logic formulas. We then construct a model of the system. We label all the states in the model where the formulas hold to see whether the initial states (the states in the model with no actions leading to them) of the model satisfy the formulas or not.

Model checking is a complete verification technique. The disadvantage of model checking is that it needs to model the whole system first and then prove the correctness of the design. Simulation performs an imperfect verification on the original system because it cannot check all the possible situations, while model checking performs a formal verification on a potentially imperfect model of the system. Model checking is complete but not compositional. So a concurrent system will have a huge amount of states. Model-based testing is compositional but not necessarily complete as some tests could take infinite time to run and also tests are derived from an incomplete model of the system. The logical nature of specification for model checking allows us to only specify the properties of interest while the algebraic nature of specification for model-based testing is restrictive. In model-based testing, the specification is represented as a labeled transition system or a finite state automaton. So one has to more or less present the whole specification. Both model checking and model-based testing are formal verification methods, that is, they allow the formal and automated verification of software.

The thesis we advance is that: CTL model checking and failure trace testing are equivalent.

In defence of our thesis, we show in this work that for each computational temporal logic formula, there is an equivalent test suite and the other way around.

Why all of these are important? We can have a mixed specification in terms of both model checking and model-based testing. Such a specification could be given by somebody else, but most often an algebraic specification is just more convenient for some components while logic specifications are more suitable for others. Indeed, considering a system where a part is specified by temporal logic formulas, it could be model checked to show the correctness of that part. Let the other part of the same system be specified algebraically, which could be verified using model-based testing. We cannot be convinced by just combining the results together that the whole system will be correct. However, we now can convert CTL formulas to failure trace tests or the other way around, and then apply on the whole system either of the two methods of formal verification. The result will be convincing and correct.

The paper continues as follows: we introduce model checking, model-based testing, preorder testing and failure trace testing in Chapter 2 along with the foundation work that has been done. Our work is presented in Chapter 3, where we build the equivalence between labeled transition systems and Kripke structures, the conversion from failure trace tests to CTL formulas and the conversion from CTL formulas to tests. We conclude in Chapter 4.

Chapter 2
Preliminaries

2.1 Model Checking

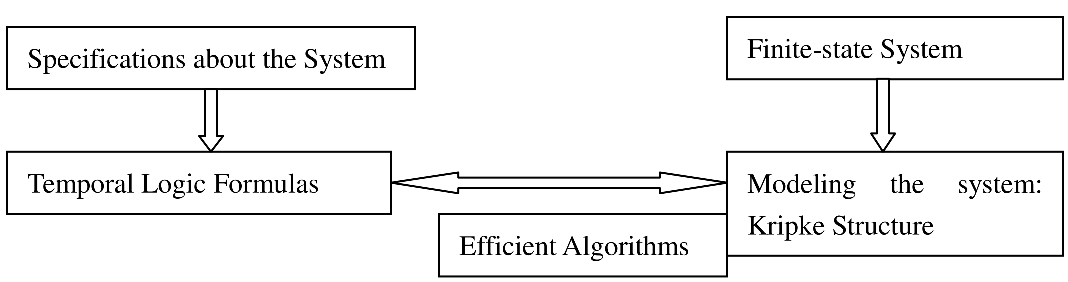


Figure 2.1: The construction of Model Checking

A system specification [12] can be described by a temporal logic formula f [1, 3, 4, 7, 8, 10, 34]. We then need to construct a formal model for the system. The model should have the same properties as the system. We use a Kripke structure [1, 2] $K = (S, S_0, R, L)$ to represent a finite-state system. Our goal will then be to find the set of all states in S that satisfy f: $\{s \in S \mid K, s \models f\}$. Some states of the system are designated as initial states. The system then satisfies the specification provided that all the initial states are in the set defined above. Figure 2.1 shows the idea of model checking.

The verification is automatic. In practice, we can check the results of the verification. If there is an error trace (that is, an explanation of the error), then we can

track down where the error occurred. We can then fix the problem caused by incorrect implementation.

Here we describe a logic which is called *temporal logic* for specifying the properties of the labeled transition systems. Temporal logic has found extensive applications in the area of automatic specification and verification of programs, especially for those concurrent programs in which the computation is performed by two or more processors that run in parallel. In order to ensure correct behavior of such a program, it is necessary to specify the way in which the actions of the various processors are interrelated. Temporal logic is the formalism for describing sequences of transitions between the states of a system. The logic has the advantage of not always requiring to specify the full behavior of a system, but to concentrate on specifying particular properties of a system, such as safety, fairness, etc. that are of interest [28]. Safety properties can be stated like “bad things do not happen"; i.e. the automobile should not start the engine without the gear being placed in park. Fairness properties can be stated like "something should happen infinitely often"; i.e. for a communication protocol, the formula $\neg send_i \vee receive_i$ means either a message is not sent or a message is received. The logic that uses time to describe sequences of states is called temporal logic. In temporal logic time is however not mentioned explicitly; however, we can notice that certain actions are executed or will be executed eventually by the definitions and notations of the logic. The logic uses atomic propositions and Boolean operators, such as conjunction, disjunction and negation to build up complicated and sophisticated expressions to describe the properties of the systems. These operators can also be combined by nesting them arbitrarily. A proposition such as "It is snowing." can be true at some time and can turn out to be false at another time. The value of the proposition varies with the time, but it cannot be both true and false at the same time. In temporal logic, the values of the propositions vary with the passing of time. On the opposite, we have non-temporal logics, where the values of propositions are constant with time.

CTL* [3, 7, 35], CTL [26, 34] (computation tree logic) and LTL [18, 24 , 25] (linear-time temporal logic) are all temporal logics. CTL* is the most general. Both LTL and CTL are strict subsets of CTL*.

CTL* formulas are used to describe the properties of computation trees. Such a tree is shaped by designating a state in some Kripke structure (a Kripke structure is used to model a system. We mentioned Kripke structure earlier in the paper and we will give the formal definition later) as the initial state and afterwards unwinding the structure into an infinite tree with the designated state as root.

CTL* formulas consist of path quantifiers and temporal operators. There are two path quantifiers: “A” and “E”. Path quantifiers are used to describe the branching structure of the computation tree as the nodes or states in the computation tree develop several successive states which then generate multiple paths starting from the same state. “A” then refers to all (for all the computation paths) and “E” refers to exist (for some of the computation paths). There are five basic temporal operators:

--X “next”: requires that a property holds in the second state of the path.

--F “eventually”: requires that a property will hold at some state of the path. That is, $Ff = \top Uf$. U refers to “until”, see the definition below.

--G “always or globally” requires that a property holds at every state on the path. $Gf = \neg F \neg f$.

--U “until”: It holds if there is a state on the path where the second property holds and it must hold forever afterwards. At every preceding state on the path, the first property holds. The second property needs to be true as soon as the first property becomes false in order for the whole formula to hold. That is to say, the second property will be verified in the future.

--R “release”: It requires that the second property hold along the path up to where the first property holds. But the first property is not required to hold eventually and it can only hold once. In other words, it means if the first statement is true, then the second statement won't be taken into consideration or there is no guarantee the second property will be verified in the future.

Figure 2.2 illustrates graphically the two dual logic "Until" and "Release" operators.

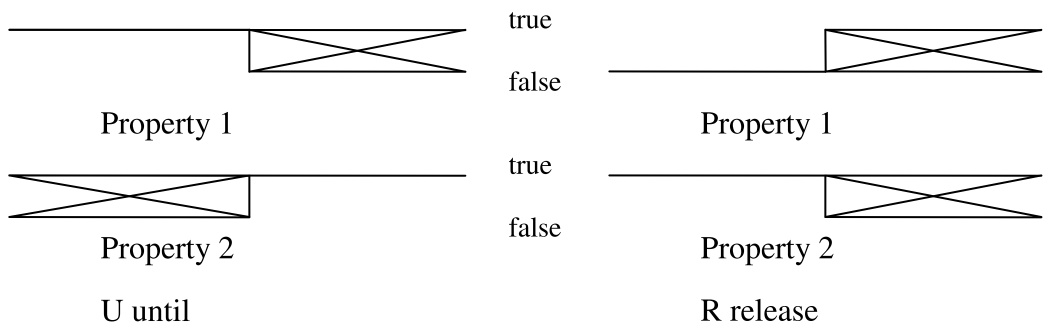


Figure 2.2: Until and Release operators

We have two kinds of formulas in CTL*: state formulas and path formulas. State formulas hold in a specific state and use temporal operators, while path formulas hold along the specific path and use path quantifiers.

2.1.1 The Syntax of CTL Formulas

There are two sub-logics of CTL*: CTL and LTL. LTL uses the linear time approach which considers time to be a linear sequence which does not branch. The temporal logic CTL was introduced by Clarke and Emerson [8]. CTL uses the branching time approach, which adopts a tree structured time, allowing some states to have more than a single successor. In CTL, the temporal operators X, F, G, U, R must be immediately preceded by a path quantifier A or E. CTL expresses the properties along a tree-like time flow. We use AP to denote the set of atomic properties ranged over by a, a_1, a_2,… a_i.

The set of CTL state formulas is the smallest set of formulas such that:

1. $\top$, $\bot$, atomic propositions a, negation of atomic propositions $\neg a$ are CTL state formulas.
2. If f, g are CTL state formulas, then $(f \wedge g)$ and $(f \vee g)$ are state formulas.
3. If f is a path formula, then Ef and Af are CTL state formulas.

The set of CTL path formulas is the smallest set of formulas such that:

If f, g are CTL path formulas, then X f, F f, G f, f U g, f R g are path formulas. These path formulas must be preceded by a path quantifier A or E to form CTL formulas and thus become state formulas. The syntax of CTL can be defined as follows:

$$\Phi ::= \top \mid \bot \mid a \mid \neg f \mid f_1 \wedge f_2 \mid f_1 \vee f_2 \mid \mathrm{AX}f \mid \mathrm{AF}f \mid \mathrm{AG}f \mid \mathrm{A}\, f_1\ \mathrm{U}\, f_2 \mid \mathrm{A}\, f_1\ \mathrm{R}\, f_2 \mid \mathrm{EX}f \mid \mathrm{EF}f \mid \mathrm{EG}f \mid \mathrm{E}\, f_1\ \mathrm{U}\, f_2 \mid \mathrm{E}\, f_1\ \mathrm{R}\, f_2 \mid$$

Each of the ten path formulas beginning with path quantifier A and E can be expressed in terms of three operators EX, EG, EU. This is useful for model checking.

$\mathrm{AX}f = \neg \mathrm{EX}(\neg f)$ $\quad$ $\mathrm{EF}f = \mathrm{E}[\top\ \mathrm{U}\, f]$ (because $\mathrm{F}f = \top\ \mathrm{U}\, f$)

$\mathrm{AG}f = \neg \mathrm{EF}(\neg f) = \neg \mathrm{E}(\top\ \mathrm{U}\ (\neg f))$ $\quad$ $\mathrm{AF}f = \mathrm{A}(\top\ \mathrm{U}\, f) = \neg \mathrm{EG}(\neg f)$

$\mathrm{A}\, f_1\ \mathrm{U}\, f_2 = \neg \mathrm{E}[\neg f_2\ \mathrm{U}\ (\neg f_1 \wedge \neg f_2)] \wedge \neg \mathrm{EG}(\neg f_2)$

$\mathrm{A}\, f_1\ \mathrm{R}\, f_2 = \neg \mathrm{E}[\neg f_1\ \mathrm{U}\ \neg f_2]$

$\mathrm{E}\, f_1\ \mathrm{R}\, f_2 = \neg \mathrm{A}[\neg f_1\ \mathrm{U}\ \neg f_2]$

Figure 2.3 shows graphically the systems in which the various CTL formulas hold.

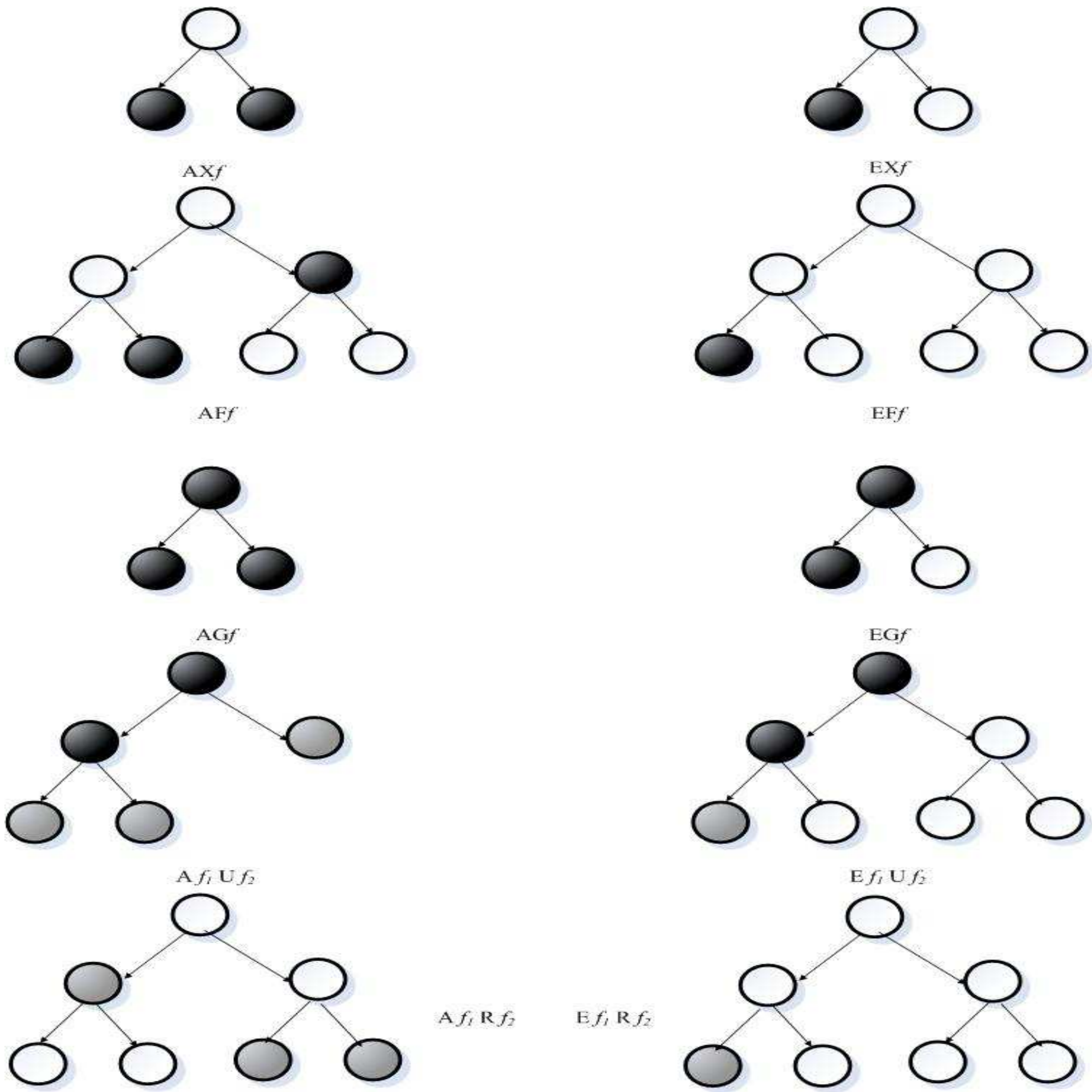


Figure 2.3: The graphs for CTL formulas

2.1.2 The Semantics of CTL Formulas

Definition 2.1 Kripke Structure. A Kripke structure K for AP is a tuple (S, S_0, R, L) where:

1. S is the set of states.

2. S_0 is the set of initial states.

3. $R \subseteq S \times S$ is the transition relation; an element (s, t)$\in$ R, where s, t$\in$ S is usually written

as s R t or $s \rightarrow t$. We require that R be total, so for $\forall s \in S$, $\exists t \in S$, such that $(s, t) \in R$.

4. L: $S \rightarrow 2^{AP}$ is the valuation that checks which atomic propositions are true. It is a function that labels each state with a set of atomic propositions that are true in that state.□

Definition 2.2 Path in a Kripke Structure.

A *path* π in a Kripke structure is a nonempty finite or infinite sequence $s_0 \rightarrow s_1 \rightarrow s_2 \ldots \in S$, such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. The path starts from state s_0. For any Kripke structure K and states S in the Kripke structure, we can build a computation tree with root labeled s_0, such that $s \rightarrow t$ is an edge in the tree iff $(s, t) \in R$, where $s, t \in S$.□

The truth value of CTL formulas is then defined in terms of the Kripke structure. The notation K, s $\models f$ means that in a Kripke structure K, formula f is true in state s. The notation K, $\pi \models f$ means that in a Kripke structure K, formula f, is true along the path π.

Definition 2.3 We define the meaning of $\models$ recursively. f and g here are state formulas unless stated otherwise. State formulas are defined as follows:

1. K, s $\models \top$ is true for any state s in any Kripke structure K.

   K, s $\models \perp$ is false for any state s in any Kripke structure K.

2. K, s $\models$ a $\Leftrightarrow$ $a \in L(s)$, $a \in AP$ is an atomic proposition.

3. K, s $\not\models f$ $\Leftrightarrow$ $\neg$(K, s $\models f$).

4. K, s $\models f \wedge g$ $\Leftrightarrow$ K, s $\models f$ and K, s $\models g$.

5. K, s $\models f \vee g$ $\Leftrightarrow$ K, s $\models f$ or K, s $\models g$.

6. K, s $\models$ Ef $\Leftrightarrow$ there is a path $\pi = s \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_i$ with an initial state s, K,

$\pi \models f$ holds, where f is a path formula.

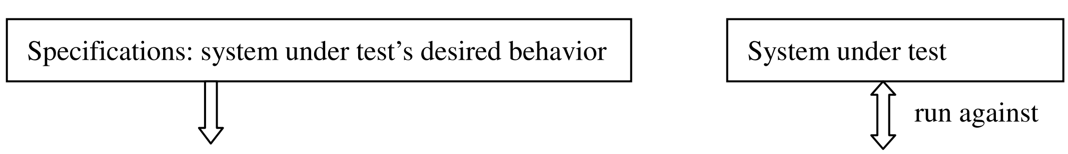
7. K, s $\models$ Af $\Leftrightarrow$ for all the paths $\pi = s \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_i$ with an initial state s, K, $\pi \models f$ holds, where f is a path formula.

We use π^i to denote the ith state of π. Path formulas are defined as follows:

8. K, $\pi \models$ Xf $\Leftrightarrow$ K, $\pi^1 \models f$. Here π^1 refers to the state s_1 (there exists a state s, a path $\pi = s\pi^1$ and $(s, s_1) \in R$) and the formula f holds at that state.
9. K, $\pi \models f$ U g $\Leftrightarrow$ there exists $j \geq 0$ such that K, $\pi^j \models g$ and $\pi^k \models g$ for all $k \geq j$, meaning g holds at the state s_j and the later states, and for all $i<j$, K, $\pi^i \models f$, meaning f holds from the initial state of π up to the state s_i (including s_i).
10. K, $\pi \models f$ R g $\Leftrightarrow$ for all $j \geq 0$, and every $i<j$ such that K, $\pi^i \not\models f$, meaning f does not hold from the initial state of π up to the state s_i (including s_i), then K, $\pi^j \models g$ and $\pi^k \models g$ for all $0 \leq k \leq j$, meaning g holds at the state s_j and the previous states.□

CTL semantics is defined over Kripke structures, where each state is labeled with atomic propositions. By contrast, the common model used for system specifications in model-based testing is the finite state machine (FSM) model, or equivalently the state transition system or the labeled transition system (LTS), where the labels or formulas are associated not with the states but with the transitions. There are generally less numbers of states in labeled transition systems compared to those in Kripke structures.

2.2 Model-based Testing



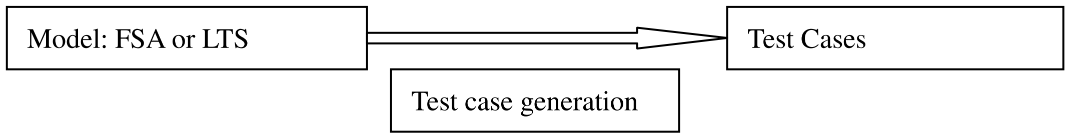


Figure 2.4: The construction of Model-based Testing

Figure 2.4 shows the idea of model-based testing. Model-based testing [11, 13, 14, 20, 23] is a kind of testing in which test cases (a set of conditions and variables) are derived from a model that describes some functional aspects of the system under test. Tests are derived systematically and formally from the model, such that they are sound and complete. The system under test is the system or process we need to check for correctness. The model is usually a partial or abstract representation of the system under test's desired behaviours. The model is translated into a finite state automaton [2, 5, 31]or a labeled transition system [20, 30, 31].

Definition 2.4 Finite State Automaton (FSA).

A FSA is a quintuple F = (K, Σ, δ, s, F), where K is a finite set of states, Σ is an alphabet, s $\in$ K is the initial state, F $\subseteq$ K is the set of final states, $\delta \subseteq K \times \Sigma \times K$ is the transition function.□

Definition 2.5 Labeled Transition Systems (LTS). An LTS is a tuple M = (S, A, $\rightarrow$, S_0), where S is a countable, non empty set of states. $S_0 \in$ S is the initial state. A is a countable set of labels denoting observable events. Observable events or actions are those that can be observed by the external environment. $\rightarrow \subseteq S \times (A \cup \{\tau\}) \times S$ is the transition relation, where τ is the internal action that cannot be observed by the external environment. We often use $p \xrightarrow{a} q$ instead of $(p, a, q) \in \; \rightarrow$.□

An LTS has a countable set of states, while an FSA has only a finite number of states. That's to say, there are many more available states in an LTS than those in an FSA, since a countable set could be either an infinitely countable set or a finite set.

In general, we can think of a set of processes or systems under test (represented as an LTS) and a set of relevant tests. Tests run parallel with the process or system under test and synchronize with it over visible or observable actions. Now we define a set T of tests and a set P of processes. A run of a test t and a process p represents a possible sequence of states and actions of t and p running synchronously. There exists a set of runs Obs(t, p), where $t\in T$, $p\in P$. If we have $r\in Obs(t, p)$ then the result of t testing p may be the run r. We take the outcomes of the particular runs of a test as being success or failure. We represent the outcomes as elements of the two-point lattice:

$$\odot = \begin{array}{c} \top \\ | \\ \bot \end{array}$$

The outcome of t testing p is $\top$ if there exists $r\in Obs(t, p)$ such that r is successful.

The outcome of t testing p is $\bot$ if there exists $r\in Obs(t, p)$ such that r is unsuccessful, or r contains a state q such that $q\uparrow$ and q is not preceded by a successful state. We say that such a state q diverges or is divergent in the sense that its properties are undefined: $q\uparrow$ iff $\exists q_1, q_2, \ldots q_k, \ldots$, such that $q \xrightarrow{\tau} q_1 \xrightarrow{\tau} q_2 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_k \xrightarrow{\tau} \ldots$. That is to say, there is an infinite path of τ actions starting from the state q. When a state is convergent or converges, such a state is well defined.

Since some tests are nondeterministic, there may be many different runs of a given test on a process under test; therefore, a set of outcomes is required to give the results of possible runs.

There are three standard powerdomain constructions.

$$
P_{may} = \begin{array}{l} \{\top\} \approx \{\top, \bot\} \\ \mid \\ \{\bot\} \end{array}
$$

The may powerdomain P_{may} incorporates may information; it identifies that a process may pass a test in order to be regarded as successful.

$$
P_{must} = \begin{array}{l} \{\top\} \\ \mid \\ \{\bot\} \approx \{\top, \bot\} \end{array}
$$

The must powerdomain P_{must} incorporates must information; it identifies tests that must be successful.

$$
P_p = \begin{array}{l} \{\top\} \\ \mid \\ \{\top, \bot\} \\ \mid \\ \{\bot\} \end{array}
$$

The Plotkin powerdomain P_p combines both may and must information.

2.3 Preorder Testing

To analyze processes, it is necessary to consider those sequences of events that can be observed as the interface of the processes. A trace [5, 15] is simply a record of events in the order that they occur. The set of traces of a process is the set of all sequences that might possibly be recorded. The set of all possible traces of a process P is denoted by traces(P).

Traces are simply a particular class of finite or infinite sequences of events that are drawn form $A^{\surd} = A \cup \{\surd\}$, which are all the executions available from the process. A represents the set of actions of the process. The √ (tick) represents termination. No event of the process can happen after the termination happens. In other words, the termination event occurring in a trace must appear at the end. A path π starting from state $p' \in P$ is a sequence ($\langle p_{i-1}, a_i, p_i \rangle$), where $0<i\leq k$, and $k\in N$, such that $k = 0$, or $p_0 = p'$ and $p_{i-1} \xrightarrow{a_i} p_i$ for all i, where $0<i\leq k$. We use $|\pi|$ to refer to k, the length of π. If $|\pi|<\infty$. Then we say that π is finite. The visible trace trace(π) of π is denoted as the sequence $(a_i)_{i\in I_\pi} \in A^*$, where $I_\pi =_{def} \{0<i\leq|\pi| \text{ and } a_i \neq \tau\}$.

In the above definition of trace, we ignore the intermediate states (which involve internal actions) and internal transitions (which are not observable by the external environment), and consider only visible transitions. The intermediate states are those states that have internal outgoing actions, in other words, they are not stable states. Here, a trace becomes a record of the visible events of an execution. We denote the sequence of actions of a process by W, thus $W \subseteq A^*$, which means W is a sequence of actions whose elements are the combination of $a_i \in A^*$.

The notation $p \overset{w}{\Rightarrow} p'$ states that there exists a sequence of transitions whose initial state is p and whose final state is p'. The visible transitions form the sequence w. Since the termination can only occur at the end of an execution, the tick √ may happen in w only at the end. In other words, p' has no transition. $\exists p': p \overset{w}{\Rightarrow} p'$ can be shortened as $p \overset{w}{\Rightarrow}$.Formally, $q \overset{\varepsilon}{\Rightarrow} q'$ iff $q = q'$ or $q \overset{\tau}{\rightarrow} q'$ or $\exists q_1: q \overset{\tau}{\rightarrow} q_1 \overset{\varepsilon}{\Rightarrow} q'$. $q \overset{a}{\Rightarrow} q'$ and $a \in A$ iff $\exists q_1, q_2: q \overset{\varepsilon}{\Rightarrow} q_1 \overset{a}{\rightarrow} q_2 \overset{\varepsilon}{\Rightarrow} q'$; $q \overset{w_1 w_2}{\Rightarrow} q'$ iff $\exists q_1: q \overset{w_1}{\Rightarrow} q_1 \overset{w_2}{\Rightarrow} q'$. The traces of a process P then can be defined as traces(P) = $\{w \mid p \overset{w}{\Rightarrow}\}$.

Definition 2.6 Finite Trace of Processes (Fin(P)).

The finite traces of a process P can be defined as Fin(P) = $\{tr \in traces(P) \mid \delta(tr) \subseteq A^{\surd} \wedge |tr| \in N \wedge \surd \notin \delta(init(tr))\}$, where init(P) = $\{a \in A | P \overset{a}{\Rightarrow}\}$ and $\delta(tr)$ refers to the set of all the elements in the trace tr. It is a subset of the set of actions of the process. |tr| refers to the length of the trace tr. The third condition tells us that the tick cannot happen at the beginning of a trace, which means in effect that the tick can only appear as the last element of a trace.□

A process P which can make no internal progress which means it has no intermediate states or outgoing internal actions is said to be *stable* [36], defined as $P\downarrow = \neg(P \overset{\tau}{\rightarrow})$. We say such a state is *stable* in the sense that it cannot perform any internal action. More generally, a *stable* process P can always respond in some way to the offer of a set of event $X \subseteq A^{\surd}$, if there is at least one $a \in X$ that P can execute. If there is no such $a \in X$, then P will *refuse* the entire set X.

A *refusal* [36] can be thought of as an experiment on the process P. The environment will offer the set X, and wait as long as necessary to observe if any events in the set X are performed. If no events are performed, then X is considered to be a refusal of P, denoted by P ref X. It is defined as: $P \text{ ref } X \equiv \exists p' : p \overset{\varepsilon}{\Rightarrow} p' \;\wedge\; p'\downarrow \;\wedge\; \forall a \in X : \;\neg(p' \overset{a}{\rightarrow})$.

The refusal of an offer set X might happen during an execution of the process P. The refusal thus will be recorded together with the finite sequence w (the sequence of events of how the process proceeds) in order to indicate at which step, or state the refusal happens.

Definition 2.7 Stable Failure of Processes (SFp).

The observation (w, X) is called a *stable failure* [36] of P.

$SFp = \{(w, X) \mid \exists p^w : p \overset{w}{\Rightarrow} p^w \;\wedge\; p^w\downarrow \wedge\; p^w \text{ ref } X\}$.□

The process P performs the events in w, then reaches a stable state where it refuses all the events in the set X. In other words, if the process P reaches a state after the performance of w where only events in the set X are provided, then no further execution can be done.

Definition 2.8 Stable Failure Preorder ($\sqsubseteq_{SF}$).

Let P and Q be two processes. Then:

$P \sqsubseteq_{SF} Q$ if and only if $Fin(P) \subseteq Fin(Q)$ and $SFp \subseteq SFq$.□

P implements Q if and only if the set of finite traces of process P is included in the set of finite traces of process Q. In addition, the set of stable failure of process P is also included in the set of stable failure of process Q. Given the preorder [30, 31], one can easily define the stable failure testing equivalence $\simeq_{SF}$: $P \simeq_{SF} Q$ iff $P \sqsubseteq_{SF} Q$ and $Q \sqsubseteq_{SF} P$. Other preorders can be defined. For instance, in trace preorder $\sqsubseteq_T$, we record all the traces of a process. In testing preorder $\sqsubseteq$, we only distinguish between success and failure of tests. In general, $\sqsubseteq_{SF}$ is one of the finest preorders.

Figure 2.5 illustrates two processes that are equivalent under testing preorder, trace preorder and are not equivalent under stable failure preorder. (as process P refuses the set {a} or the set {b}, while process P' refuses the set {a, b} or the set {b}).

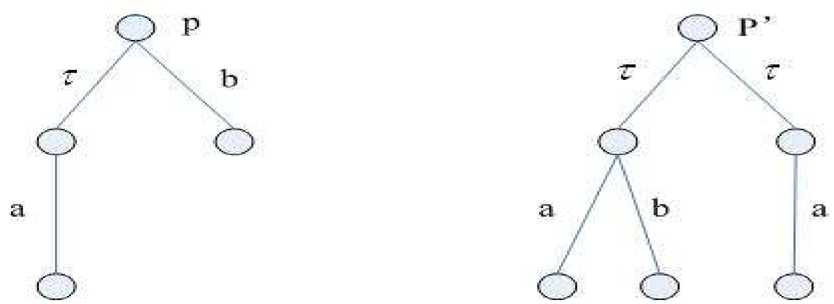


Figure 2.5: Processes equivalent under testing preorder, trace preorder and not equivalent under stable failure preorder ($p \simeq p'$; $p \simeq_T p'$; $p \not\simeq_{SF} p'$)

2.4 Failure Trace Testing

Definition 2.9 Failure Trace [15].

A failure trace *f* is a string of the form:

$f = \{A_0\}a_1\{A_1\}a_2 \ldots\ldots a_n\{A_n\}$, $n \geq 0$, with

-- $a_i \in A^*$, $1 \leq i \leq n$

-- $A_i \subseteq A$ or $A_i = \emptyset$, $0 \leq i \leq n$, it is the set of refusals.□

Let P be a process and let $w = a_1a_2 \ldots\ldots a_n$ be a trace of P. Then,

$\exists P_0P_1 \ldots P_n$: $P \stackrel{\varepsilon}{\Rightarrow} P_0 \stackrel{a_1}{\Rightarrow} P_1 \stackrel{a_2}{\Rightarrow} \ldots\ldots P_{n-1} \stackrel{a_n}{\Rightarrow} P_n$.

Let *f* be a failure trace:

--if $P_i \xrightarrow{\tau}$, then $A_i = \emptyset$, $(0 \le i \le n)$, which means that P_i is not a stable state, thus that state refuses an empty set of events.

--if $\neg(P_i \xrightarrow{\tau})$, then $A_i = \emptyset$ or $A_i \subseteq (A\backslash init(P_i))(0 \le i \le n)$, where $init(P_i) = \{g \in A \mid p \xrightarrow{g}\}$, which means that at some states of process P, if those states are stable with no internal actions accepted, then the failure trace refuses nothing, or it refuses any events that are not possibly available to those states.□

In all, we get a failure trace of P by taking a trace of P and then put in the trace after the stable states refusal sets of those states.

Definition 2.10 The Testing Language TLOTOS [32].

A is the countable set of observable actions as we defined in the definition of labeled transition systems. T is the set of processes or tests. T_1 and T_2 are processes or tests.

The syntax of TLOTOS is defined as follows:

stop $\mid$ a; T_1 $(a \in A)$ $\mid$ i; T_1 $\mid$ θ; T_1 $\mid$ pass $\mid$ T_1 [] T_2 $\mid \Sigma T \mid$

The semantics of TLOTOS is defined as follows:

1. inaction: no rules.

2. action prefix: $a; T_1 \xrightarrow{a} T_1$.

$i; T_1 \xrightarrow{\tau} T_1$.

3. deadlock detection: $\theta; T_1 \xrightarrow{\theta} T_1$.

4. successful termination: $pass \xrightarrow{\gamma} stop$.

5. choice: $g \in A \cup \{\gamma, \theta, \tau\}$:

$$\frac{T_1 \xrightarrow{g} T_1'}{T_1 \ [] \ T_2 \xrightarrow{g} T_1'}$$

$$\frac{T_2 \xrightarrow{g} T_2'}{T_1 \ [] \ T_2 \xrightarrow{g} T_2'}$$

6. generalized choice: $g \in A \cup \{\gamma, \theta, \tau\}$:

$$\frac{T_1 \xrightarrow{g} T_1'}{\sum \{T_1\} \ \cup \ T \xrightarrow{g} T_1'}$$

In addition, we define the deadlock detection label θ which has the lowest priority as follows:

$P \parallel_\theta T$

1. $$\frac{P \xrightarrow{\tau} P'}{P \parallel_\theta T \xrightarrow{\tau} P' \parallel_\theta T}$$

2. $$\frac{T \xrightarrow{\tau} T'}{P \parallel_\theta T \xrightarrow{\tau} P \parallel_\theta T'}$$

3. $$\frac{T \xrightarrow{\gamma} Stop}{P \parallel_\theta T \xrightarrow{\gamma} Stop}$$

4. $$\frac{P \xrightarrow{a} P', \quad T \xrightarrow{a} T' \text{ (where } a \in A)}{P \parallel_\theta T \xrightarrow{a} P' \parallel_\theta T'}$$

5. $\neg(\exists P \in A \cup \{\tau, \gamma\}: P \parallel_\theta T \xrightarrow{p})$

$$\frac{T \xrightarrow{\theta} T'}{P \parallel_\theta T \xrightarrow{\theta} P \parallel_\theta T'}$$ □

Definition 2.11 Sequential Tests [15].

We shorten sequential tests to SeTests in the following. We define the set of sequential tests as follows:

1. Succ∈ SeTests.
2. Fail∈ SeTests.
3. If T∈ SeTests, then for a∈ A, a; T∈ SeTests.
4. If T∈ SeTests, then for A⊆A, $\sum\{a;\text{stop} \mid a\in A\}$ []θ; T∈ SeTests.□

We can make a bijection between failure traces and sequential tests.

Let T be a sequential test, then we can recursively define the failure trace $f = ftr$(T).

T	f
T: Succ	f: Ø
Fail	X, X⊆A
a; T'	a ftr(T')
$\sum\{a;\text{stop} \mid a\in A\}$ [] θ; T'	A ftr(T')

Let f be a failure trace, then we can recursively define the sequential test T=st(f).

f	T
f: Ø	T: Succ
X, X⊆A	Fail
af	a; st(f)
Af	$\sum\{a;\text{stop} \mid a\in A\}$ [] θ; st(f).

Thus, we say that for all failure traces f, $ftr(\mathrm{st}(f)) = f$, and for all tests T, $\mathrm{st}(ftr(T)) = T$. So we can convert sequential tests to failure traces and the other way around.

We can thus convert a process-oriented testing equivalence to a testing-based testing equivalence.

Theorem 2.1

Let P be a process, T be a sequential test and f a failure trace, then (P may T) $\Leftrightarrow$ ($f \in ftr(P)$), where $f = ftr(T)$.

More simply expressed, if P runs parallel with the test T, then there is a successful run iff f is a failure trace of T and it represents T. f should be a possible existent failure trace in P. In the testing theory we have the theorem that ($P \sqsubseteq Q$) $\Leftrightarrow$ (for every $T \in$ Test, P may T implies Q may T). Here, we find an analogy to the definition $P \sqsubseteq_{SF} Q$ if and only if $\mathrm{Fin}(P) \subseteq \mathrm{Fin}(Q)$ and $SFp \subseteq SFq$, which is as follows: Let P and Q be two processes, we define $P \sqsubseteq_{FT} Q$ if and only if $ftr(P) \subseteq ftr(Q)$. A process-based testing equivalence on both stable failures and failure traces expresses exactly the same idea.

Proof of Theorem 2.1. (P may T) $\Leftrightarrow$ ($f \in ftr(P)$).

1. $f = Ø$ and T = PASS, for all processes P , P may PASS and $Ø \in ftr(P)$.
2. $f = af'$ and T = a;T', and $f' = ftr(T')$: P may a;T' $\Leftrightarrow$ $\exists P_0, P_1$: $P \stackrel{\varepsilon}{\Rightarrow} P_0 \stackrel{a}{\Rightarrow} P_1 \wedge P_1$ may T' $\Leftrightarrow$ $\exists P_0, P_1$: $P \stackrel{\varepsilon}{\Rightarrow} P_0 \stackrel{a}{\Rightarrow} P_1 \wedge f' \in ftr(P_1)$ $\Leftrightarrow$ $af' \in ftr(P)$.
3. $f = Af'$ and T = $\sum \{a; stop \mid a \in A\}$ [] θ; T', and $f' = ftr(T')$:

   P may $\sum \{a; stop \mid a \in A\}$ [] θ; T' $\Leftrightarrow$ $\exists P_0$: $P \stackrel{\varepsilon}{\Rightarrow} P_0 \wedge A \subseteq (A \backslash \mathrm{init}(P_0)) \wedge \neg P_0 \stackrel{\tau}{\rightarrow}$

$\wedge$ P_0 may T' $\Leftrightarrow$ $\exists P_0$: $P \overset{\varepsilon}{\Rightarrow} P_0$ $\wedge$ $A \subseteq ($ $A \backslash init(P_0))$ $\wedge$ $\neg P_0 \overset{\tau}{\rightarrow}$ $\wedge$ $f' \in ftr(P_0)$ $\Leftrightarrow$

$Af' \in ftr(P)$ and $init(P) = \{g \in A \mid p \overset{g}{\rightarrow}\}$. □

2.5 Foundation Work

We close here our survey of the existing work in the area of system verification. We note that model checking is well studied. In model checking, we specify the system by temporal logic formulas. Temporal logic comes in three varieties: CTL*, CTL and LTL. In this paper, we concentrate on CTL. A finite-state system can be viewed as a Kripke Structure (see Definition 3.1) and it can be specified using temporal logic. To verify the correctness of the program, one has only to check the Kripke structure of the system to see if the initial states of the system satisfy the given temporal logic formulas. Pioneering work in model checking is done by Edmund M. Clarke Jr., Orna Grumberg and Doron A. Peled [10] and by Rance Cleaveland and Gerald Lüttgen [5]. Formal testing is another common method of system verification. In model-based testing, finite state automata or labeled transition systems are used to model the system's desired properties. We generate test cases from those models and run the test cases against systems under test to see the results. We introduced a testing preorder which takes both trace preorder and stable failures of a system into consideration. It enhances the strength of distinguishing systems. Failure trace testing is introduced by Rom Langerak [15]. In his paper, he also introduced the set of SeTests that is sufficient to assess the failure trace relation. Let T be a test. There exists a set of SeTests, such that for all processes P: P may T $\Leftrightarrow$ $\exists T' \in$ SeTests: P may T'. Both model checking and formal testing have advantages and disadvantages.

Model checking does the whole verification on a potentially imperfect model, Kripke structure, of the system. Formal testing does the imperfect verification on the original system. In addition, when designing a system, some properties may be specified using temporal logic that are more suitable for model checking. Other desired properties of the system may be specified using finite-state automata or labeled transition systems that are more suitable for testing. A combined method which can convert the two would have obvious benefits. Consider some properties that are expressed as CTL formulas and hold for some part A of a larger system. They could be model checked. We know then that part A is correct. We have a second part B of the same system, where our specification is algebraic. We can do model-based testing on it and once more be sure that part B is correct. Now we put them together. Is the result correct? It is not always so. We need to check it formally. However, we do not have a global formal specification: part of it is logic, and other part is algebraic. Ergo, before checking the whole system, we need to convert one specification to the form of the other. Thus, we can just verify the system once.

We shall therefore introduce a new, practical and meaningful method of verification in Chapter 3.

Chapter 3

The Equivalence between CTL and Failure Trace Testing

We present in this chapter the major contribution of this thesis. Indeed, we analyze the equivalence between labeled transition systems and Kripke structures and then we establish the equivalence between failure trace tests and CTL formulas. We establish such equivalence by constructing two functions that convert failure trace tests to CTL formulas and the other way around.

3.1 The Equivalence between LTS and Kripke Structure

Definition 3.1 We say a process P satisfies a (written $P \models a$) iff $P \xrightarrow{a}$. Also, any process satisfies $\top$ and no process satisfies $\bot$. Then, a process P (i.e. LTS) is equivalent to a Kripke Structure K and a state s in K, written $P \simeq (K, s)$ iff

1. $P \models \top$ iff $K, s \models \top$. $P \not\models \bot$ iff $K, s \not\models \bot$.

2. $P \models a$ iff $K, s \models a$, where $a \in L(s)$ and $a \in AP$.

3. $P \not\models f$ iff $K, s \not\models f$.

4. $P \models f \wedge g$ iff $K, s \models f \wedge g$.

5. $P \models f \vee g$ iff $K, s \models f \vee g$.

6. $P \models Ef$ iff $K,s \models Ef$.

7. P $\models$ A*f* iff K, s $\models$ A*f*.

8. P $\models$ X*f* iff K, π $\models$ X*f*.

9. P $\models$ *f* U *g* iff K, π $\models$ *f* U *g*.

10. P $\models$ *f* R *g* iff K, π $\models$ *f* R *g*.

We say a process P satisfies formula φ if P $\models$ φ holds.□

Theorem 3.1

There exists an algorithmic function ξ which converts a labeled transition system M (or a process P) into a Kripke structure K and a state s such that M $\simeq$ (K, s) or P $\simeq$ (K, s).

For any labeled transition system M = (S, A, $\rightarrow$, S_0), its equivalent Kripke structure K' is defined as K' = (S', S_0', R', L') where:

1. S' = {<s, x>|s$\in$ S, x$\subseteq$init(s): init(s) = {a$\in$ A|s$\stackrel{a}{\Rightarrow}$}. x = Ø iff init(s) = Ø }.

2. S_0' = {<s_0, x>$\in$ S'| s_0 $\in$ S_0}.

3. R' = (<s, N>, <t, O>)|<s, N>, <t, O>$\in$ S' and

   a) for any n$\in$ N: s$\stackrel{n}{\Rightarrow}$t.

   b) for some q$\in$ S and for any o$\in$ O: t$\stackrel{o}{\Rightarrow}$q.

4. L': S' $\rightarrow 2^{AP}$, with L'(s, x) = x, where AP$\subseteq$A. □

Figure 3.1 illustrates graphically how we convert a labeled transition system to its equivalent Kripke structure.

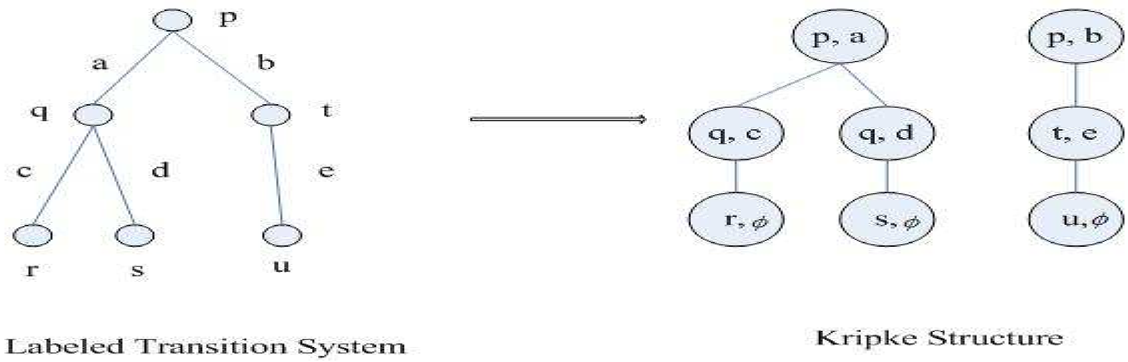


Figure 3.1: The conversion from LTS to its equivalent Kripke structure

As illustrated in the figure, we combine each state in the labeled transition system with its actions provided as properties to form new states in the equivalent Kripke structure. The transition relation of the Kripke structure is formed by the new states and the corresponding transition relation in the original labeled transition system. The labeling function in the equivalent Kripke structure links the actions to their relevant states.

Thus we can define the semantics of CTL formulas with respect to a process rather than Kripke structure.

Proof of Theorem 3.1. The proof relies on the properties of the syntax and semantics of CTL* formulas. The syntax and semantics of CTL is then restricted as usual.

Basis:

--$s \models \top$ is true for any state s in any process P and for any state in any Kripke structure K $\Leftrightarrow$ $\xi s \models \top$ $\Leftrightarrow$ K', $<s, x> \models \top$. It is immediate.

--$s \not\models \bot$ is true for any state s in any process P and for any state in any Kripke structure K $\Leftrightarrow$ $\xi s \not\models \bot$ $\Leftrightarrow$ K', $<s, x> \not\models \bot$. It is immediate.

--$s \models a$ $\Leftrightarrow$ $\xi s \models a$ $\Leftrightarrow$ K', $<s, x> \models a$, where $x \in$ init(s) and $a \in L'(s)$ for some a; $a \in AP$ is an action. It is immediate from the definition of the function ξ.

Inductive steps:

--s$\not\models f$

Because (s $\models f$) $\Leftrightarrow$ (ξs $\models f$) $\Leftrightarrow$ (K', <s, x> $\models f$) by inductive hypothesis, thus (s $\models \neg f$) $\Leftrightarrow$ (ξs $\models \neg f$) $\Leftrightarrow$ (K', <s, x> $\models \neg f$). So we get (s$\not\models f$) $\Leftrightarrow$ (ξs$\not\models f$) $\Leftrightarrow$ (K', <s, x>$\not\models f$).

--s $\models f \wedge g$ $\Leftrightarrow$ s $\models f$ and s $\models g$ $\Leftrightarrow$ K', <s, x> $\models f$ and K', <s, x> $\models g$ $\Leftrightarrow$ K', <s, x> $\models f \wedge g$.

--s $\models f \vee g$ $\Leftrightarrow$ s $\models f$ or s $\models g$ $\Leftrightarrow$ K', <s, x> $\models f$ or K', <s, x> $\models g$ $\Leftrightarrow$ K', <s, x> $\models f \vee g$.

--s $\models$ Ef $\Leftrightarrow$ there is a path π, where π = s $\rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_i$ such that s, $\pi \models f$. s, s_1, $s_2 \ldots$ $s_i \in$ S, we regard them as the processes, or they are simply the states in the process. Thus, s $\models f$ $\Leftrightarrow$ ξs $\models f$ by inductive hypothesis. The conversion from a process to its equivalent Kripke structure does not change the states themselves but add to the states their corresponding actions. $\Leftrightarrow$ there is a path π' in K', where π' = s' $\rightarrow s_1' \rightarrow s_2' \rightarrow \ldots \rightarrow s_i'$ such that K', $\pi' \models f$. s', s_1', $s_2' \ldots$ $s_i' \in$ S. $\Leftrightarrow$ K', <s, x> $\models$ Ef.

--s $\models$ Af $\Leftrightarrow$ for every path π, where π = s $\rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_i$ such that s, $\pi \models f$. s, s_1, $s_2 \ldots$ $s_i \in$ S, we regard them as the processes, or they are simply the states in the process. Thus, s $\models f$ $\Leftrightarrow$ ξs $\models f$ by inductive hypothesis. The conversion from a process to its equivalent Kripke structure does not change the states themselves but add to the states their corresponding actions. $\Leftrightarrow$ for every path π' in K', where π' = s' $\rightarrow s_1' \rightarrow s_2' \rightarrow \ldots \rightarrow s_i'$ such that K', $\pi' \models f$. s', s_1', $s_2' \ldots$ $s_i' \in$ S. $\Leftrightarrow$ K', <s, x> $\models$ Af.

f and *g* are path formulas.

We use π^i to denote the ith state of π. A path π in a Process P is a nonempty finite or infinite sequence $s_0 \rightarrow s_1 \rightarrow s_2 \ldots \in S$, such that $(s_i, s_{i+1}) \in \rightarrow$ in P for all $i \geq 0$. The path starts from state s_0.

--P, $\pi \models Xf \Leftrightarrow$ P, $\pi^1 \models f$. Here π^1 refers to the state s_1 and the formula holds at that state. $\Leftrightarrow s_1 \models f \Leftrightarrow \xi s_1 \models f$ by inductive hypothesis $\Leftrightarrow$ K', $<s_1, x> \models f \Leftrightarrow$ K', $\pi^1 \models f \Leftrightarrow$ K', $<s, x> \models Xf$.

--P, $\pi \models f \text{ U } g \Leftrightarrow$ there exists $j \geq 0$, P, $\pi^j \models g$ and P, $\pi^k \models g$ for all $k \geq j$, meaning g holds at the state s_j and the later states and for all $i<j$, P, $\pi^i \models f$, meaning f holds from the initial state of π up to the state s_i (including s_i). The conversion from a process to its equivalent Kripke structure does not change the states themselves but add to the states their corresponding actions. Here π^j, π^k, π^i in process P correspond to π^j, π^k, π^i in Kripke structure K'. $\Leftrightarrow$ there exists $j \geq 0$, K', $\pi^j \models g$ and K', $\pi^k \models g$ and for all $k \geq j$, K', $\pi^i \models f$. $\Leftrightarrow$ K', $<s, x> \models f \text{ U } g$.

--P, $\pi \models f \text{ R } g \Leftrightarrow$ for all $j \geq 0$, and every $i<j$, P, $\pi^i \not\models f$, meaning f does not hold from the initial state of π up to the state s_i (including s_i); P, $\pi^j \models g$ and P, $\pi^k \models g$ for all $0 \leq k \leq j$, meaning g holds at the state s_j and the previous states. The conversion from a process to its equivalent Kripke structure does not change the states themselves but add to the states their corresponding actions. Here π^j, π^k, π^i in process P correspond to π^j, π^k, π^i in Kripke structure K'. $\Leftrightarrow$ for all $j \geq 0$, and every $i<j$, K', $\pi^i \not\models f$, then K', $\pi^j \models g$ and K', $\pi^k \models g$ for all $0 \leq k \leq j \Leftrightarrow$ K', $<s, x> \models f \text{ R } g$. □

3.2 From Failure Trace Tests to CTL Formulas

P is any process; T is the set of test cases and φ is a set of temporal logic formulas. ψ is a function which links the sequential tests to the correspondent CTL formulas. $\psi(T) = \varphi$.

ξ is a function which converts a process P to an equivalent Kripke structure K as described in Theorem 3.1. We presume that $f_1, f_2 \ldots f_i \in \varphi$ with $0<i\leq n$ are temporal logic formulas and $t_1, t_2 \ldots t_i \in T$ with $0<i\leq n$ are sequential tests.

Theorem 3.2

There exists a function ψ such that P may T iff $\xi P \models \varphi$ where $\varphi = \psi(T)$. We then have P may t_1 iff $\xi P \models f_1$ where $f_1 = \psi(t_1)$.

Proof of Theorem 3.2. The proof is done by induction on tests as follows:

Basis:

---Succ/Pass

We put $\psi(Succ) = \top$. Any process passes Succ and any Kripke structure satisfies $\top$, thus it is immediate that P may Succ $\Leftrightarrow$ $\xi P \models \top = \psi(Succ)$.

---Fail/Stop

We put $\psi(Fail) = \bot$. No process passes Fail and no Kripke structure satisfies $\bot$, thus it is immediate that P ~~may~~ Fail $\Leftrightarrow$ $\xi P \not\models \bot = \psi(Fail)$.

---a

We put $\psi(a) = a$. It is immediate from the definition of the function ξ that P may a $\Leftrightarrow$ $\xi P \models a = \psi(a)$.

Inductive steps:

---a; T

We put $\psi(a; T) = a \wedge EX(\psi(T))$.

P may (a; T) $\Leftrightarrow$ P may a and P' may T for some $P \xrightarrow{a} P'$. P may a if and only if $\xi P \models a$. P' may T if and only if $\xi P' \models \psi(T)$ by induction hypothesis, where $P \xrightarrow{a} P'$. $\Leftrightarrow$ $\xi P \models a \wedge EX(\psi(T))$ $\Leftrightarrow$ $\xi P \models \psi(a; T)$. From theorem 3.1 we know that when we convert a labeled transition system to an equivalent Kripke structure, the news states are the original states together with their outgoing actions. As in the figure 3.1, the initial state p becomes two initial states (p, a) and (p, b). With the state satisfying the property a, the next state to that state of the Kripke structure is only q in this particular example. So $\xi P \models a \wedge EX(\psi(T))$ means that the next state to P satisfies the formula $\psi(T)$; in addition, P satisfies the property a.

---$\sum\{a;$ stopla$\in A\}$ []θ; T

We put $\psi(\sum\{a; \text{stopla} \in A\} []\theta; T) = (a_1 \vee a_2 \vee \ldots\ldots \vee a_n) \vee ((\neg(a_1 \wedge a_2 \wedge \ldots\ldots \wedge a_n)) \wedge (\psi(T))$, where $A = \{a_1, a_2, \ldots\ldots, a_n\}$.

Base case:

For any $a \in A$: P may a; stop []θ; T $\Leftrightarrow$ P may a or (P ~~may~~ a and P may T) $\Leftrightarrow$ $\xi P \models a \vee ((\xi P \not\models a) \wedge (\xi P \models \psi(T)))$. Proven.

Inductive hypothesis:

P may $\sum\{a;$ stopla$\in A\}$ []θ; T $\Leftrightarrow$ P may $\sum\{a;$ stopla$\in A\}$ or (P ~~may~~ $\sum\{a;$ stopla$\in A\}$ and P may T) $\Leftrightarrow$ P may $(a_1 \vee a_2 \vee \ldots\ldots \vee a_n)$ or (P ~~may~~ $(a_1 \vee a_2 \vee \ldots\ldots \vee a_n)$ and P may T) $\Leftrightarrow$ $\xi P \models (a_1 \vee a_2 \vee \ldots\ldots \vee a_n) \vee ((\xi P \not\models (a_1 \vee a_2 \vee \ldots\ldots \vee a_n)) \wedge \xi P \models \psi(T))$ $\Leftrightarrow$ $\xi P \models (a_1 \vee a_2 \vee \ldots\ldots \vee a_n) \vee ((\xi P \models (\neg a_1 \vee \neg a_2 \vee \ldots\ldots \vee \neg a_n)) \wedge \xi P \models \psi(T))$ $\Leftrightarrow$

$\xi P \models (a_1 \vee a_2 \vee ...... \vee a_n) \vee ((\xi P \models \neg(a_1 \wedge a_2 \wedge ...... \wedge a_n)) \wedge \xi P \models \psi(T)) \Leftrightarrow$ $\xi P \models \psi(\Sigma\{a; stop|a \in A\} []\theta; T)$.

Inductive step:

We then add a b with $b \notin A$, P may $(\{\Sigma\{a; stop|a \in A\} [](b; stop)\} []\theta; T) \Leftrightarrow$ P may $\Sigma\{a; stop|a \in A\} [](b; stop)$ or (P ~~may~~ $\Sigma\{a; stop|a \in A\} [](b; stop)$ and P may T) $\Leftrightarrow$ P may $((a_1 \vee a_2 \vee ...... \vee a_n) \vee b)$ or (P ~~may~~ $((a_1 \vee a_2 \vee ...... \vee a_n) \vee b)$ and P may T) $\Leftrightarrow \xi P \models ((a_1 \vee a_2 \vee ...... \vee a_n) \vee b) \vee ((\xi P \not\models ((a_1 \vee a_2 \vee ...... \vee a_n) \vee b)) \wedge \xi P \models \psi(T))$ $\Leftrightarrow \xi P \models ((a_1 \vee a_2 \vee ...... \vee a_n) \vee b) \vee ((\xi P \models (\neg a_1 \vee \neg a_2 \vee ...... \vee \neg a_n) \vee (\neg b)) \wedge \xi P \models \psi(T)) \Leftrightarrow \xi P \models (a_1 \vee a_2 \vee ...... \vee a_n \vee b) \vee ((\xi P \models \neg(a_1 \wedge a_2 \wedge ...... \wedge a_n \wedge b)) \wedge \xi P \models \psi(T)) \Leftrightarrow \xi P \models \psi(\{\Sigma\{a; stop|a \in A\} [](b; stop)\} []\theta; T)$.

□

3.3 From CTL Formulas to Failure Trace Tests

We assume that P is any process, T is a test suite and φ is a set of temporal logic formulas. ω is a function which links the CTL formulas to the corresponding test cases. $\omega(\varphi) = T$.

We presume that $f_1, f_2 \ldots f_i$ range over φ with $0<i\leq n$, which are temporal logic formulas and $t_1, t_2 \ldots t_i$ with $0<i\leq n$ are test cases.

Theorem 3.3

There exists a function ω such that $\xi P \models \varphi$ iff P may T, where $T = \omega(\varphi)$. We then have $\xi P \models f_1$ iff P may $\omega(f_1)$.

Proof of Therom 3.3. The proof is done by induction on computational temporal logic formulas as follows:

Basis:

---$\top$

We put $\omega(\top)$ = {Succ}. Any Kripke structure satisfies $\top$ and any process passes Succ, thus it is immediate that $\xi P \models \top \Leftrightarrow$ P may Succ.

---$\perp$

We put $\omega(\perp)$ = {Fail}. No Kripke structure satisfies $\perp$ and no process passes Fail, thus it is immediate that $\xi P \not\models \perp \Leftrightarrow$ P ~~may~~ Fail.

---a

We put ω(a) = {a; stop}. It is immediate from the definition of the function ξ that $\xi P \models a \Leftrightarrow$ P may a; stop = ω(a).

If the Kripke structure ξP satisfies the formula a, the process P itself should pass the sequential test a and the other way around.

Inductive steps:

---$\neg f$

$\xi P \models \neg f \Leftrightarrow$ P may t', and t' = $\omega(\neg f)$.

If $\omega(f)$ = t, then $\omega(\neg f)$ = t' where t' is generated like this:

Change all the states that have the verdict Succ to Fail: in terms of labeled transition system, when the system ends up at a designated Succ state, as in the definition for semantics for TLOTOS, it will perform a γ transition and lead to a stop state. However, in this construction we assign the verdict Fail to those originally designated Succ states.

When the tests end in the states with verdict Fail, they end with deadlock. Therefore, the process does not pass the tests.

Add to the rest of the states an action θ followed by an action γ, which will lead to stop: in terms of labeled transition system, at the states when the tests are not provided with the actions that could be performed, they will perform θ; γ automatically and end by stop. That means the process finally passes the tests.

ξP $\models f \Leftrightarrow$ P may t = $\omega(f)$ by induction hypothesis, thus it is obvious by the construction above that ξP $\models \neg f \Leftrightarrow$ P may t', where t' is generated by the rules above.

--- $f_1 \wedge f_2$

We put $\omega(f_1 \wedge f_2) = \{\omega(f_1) \cup \omega(f_2)\}$.

ξP $\models f_1 \wedge f_2 \Leftrightarrow \xi$P $\models f_1$ and ξP $\models f_2 \Leftrightarrow$ P may $\omega(f_1)$ and P may $\omega(f_2)$ by induction hypothesis $\Leftrightarrow$ P may $\omega(f_1) \cup \omega(f_2)$. The $\cup$ here shows that the process P needs to pass both of the test suites $\omega(f_1)$ and $\omega(f_2) \Leftrightarrow$ P may $\omega(f_1 \wedge f_2)$.

If the Kripke structure ξP satisfies the formula $f_1 \wedge f_2$, the process P should pass all of the tests in $\omega(f_1)$ and $\omega(f_2)$ and the other way around.

--- $f_1 \vee f_2$

We put $\omega(f_1 \vee f_2) = \{$t[]t': t$\in \omega(f_1)$ and t'$\in \omega(f_2)\}$.

ξP $\models f_1 \vee f_2 \Leftrightarrow \xi$P $\models f_1$ or ξP $\models f_2 \Leftrightarrow$ P may $\{$t[]t': t$\in \omega(f_1)$ and t'$\in \omega(f_2)\}$ The [] here shows that the process P needs to pass either the tests in $\omega(f_1)$ or $\omega(f_2) \Leftrightarrow$ P may $\omega(f_1 \vee f_2)$.

If the Kripke structure ξP satisfies the formula $f_1 \vee f_2$, the process P should pass either the tests in $\omega(f_1)$ or $\omega(f_2)$ and the other way around.

---EX*f*

We put $\omega(\mathrm{EX}f) = \{\sum (a; t|a \in A): t \in \omega(f)\}$.

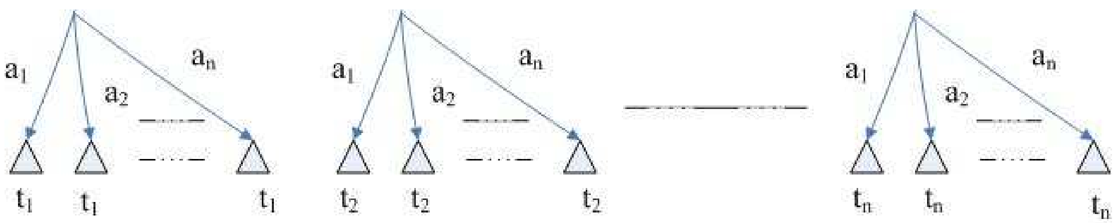


Figure 3.2: Test suite for CTL formula EX*f*

As shown in figure 3.2, the test suite is generated by combining a choice of action from the actions and the tests generated from $\omega(f)$. P may $\omega(\mathrm{EX}f)$ iff P passes each of the test cases above. P satisfies the formula EX*f* iff P can perform at least one action from the set A, and at the next states it passes the tests in $\omega(f)$.

$\xi P \models \mathrm{EX}f \Leftrightarrow$ P may $\{\sum (a; t|a \in A): t \in \omega(f)\} \Leftrightarrow$ P may $\omega(\mathrm{EX}f)$.

---AX*f*

We put $\omega(\mathrm{AX}f) = \{a; t[]\theta;\mathrm{pass}|a \in A: t \in \omega(f)\}$.

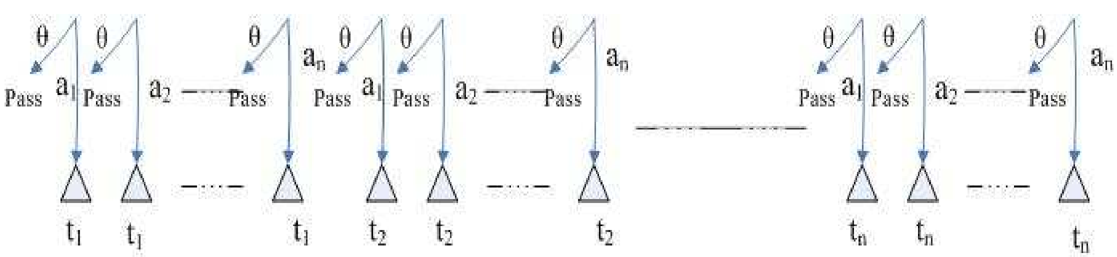


Figure 3.3: Test suite for CTL formula AX*f*

As shown in figure 3.3, the test suite is generated by combining a choice of action from the actions and the tests generated from $\omega(f)$. When the action is not provided at states, a deadlock detection transition will be taken place and lead to a pass state. The test suite is generated by providing all the actions from the set of actions. However, the system under test does not necessarily have to perform all the actions in the set of actions before going to the point where the tests are from $\omega(f)$. When the system under test runs in parallel with the test case, if one action is not provided, it encounters a deadlock but leads to a pass state. It then continues to check the results of the other runs with other test cases. P may ω(AX*f*) iff P passes each of the test cases above. P satisfies the formula AX*f* iff P can perform some of the actions in A, and if it does, then at the next state it passes the tests in $\omega(f)$.

$\xi P \models AXf \Leftrightarrow$ P may $\{a;\, t[]\theta;pass|a\in A: t\in \omega(f)\} \Leftrightarrow$ P may $\omega(AXf)$.

---EF*f*

We put $\omega(EFf) = \{t' = t[]\theta;(\sum (a;t'|a\in A)): t\in \omega(f)\}$.

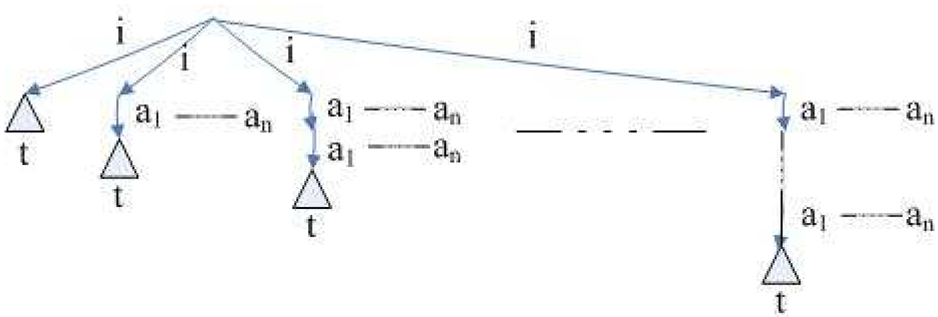


Figure 3.4: Test suite for CTL formula EF*f*

As we can see from figure 3.4, the test suite is generated by combining a choice of action from the actions and the tests in $\omega(f)$. Then, we combine a choice of action followed by another choice of action with the tests in $\omega(f)$ and so on till the last layer of the tree of Kripke structure. P may ω(EF*f*) iff P passes either of the test cases above. Thus we have internal actions leading to every test case. The resulting test suite is nondeterministic. P satisfies the formula EF*f* iff P passes the tests in $\omega(f)$, or P can perform at least one action from the set A and at the next states it passes the tests in $\omega(f)$, or P can perform at least one action from the set A, at the next states it can perform at least another action, and then it passes the tests in $\omega(f)$ and so on. It is exact the idea shown in figure 3.4. P satisfies the formula EF*f* iff P passes either of the test cases illustrated in figure 3.4.

$\xi P \models EFf \Leftrightarrow$ P may {t' = t[]θ;(∑ (a;t'|a∈ A): t∈ $\omega(f)$} $\Leftrightarrow$ P may $\omega(EFf)$.

---AF*f*

We put $\omega(AFf) = \{ \omega(f)\ []\ \omega(AXf')$, where $f' = f \vee AXf'\}$.

Here *f'* is a recursive definition. When we unfold the formula, it looks like *f* or AX($f \vee AXf'$) or AX($f \vee$ AX($f \vee AXf'$)) and so on. It shows that the process should satisfy any of the recursive unfolded formulas in order to let the process satisfy the original formula AF*f*. As in a Kripke structure, the states in the first layer need to satisfy the formula *f* or the states in the second layer need to satisfy the formula *f* and so on. In terms of labeled transition system, the process need to pass the tests in $\omega(f)$ or it performs some actions to the second layer states and then the process need to pass the tests in $\omega(f)$ and so on. The process P satisfies the formula AF*f* iff it passes one of the test cases generated from the recursively unfolded formulas.

$\xi P \models AFf \Leftrightarrow \xi P \models f'$, where $f' = f \vee AXf' \Leftrightarrow$ P may $\omega(f') \Leftrightarrow$ P may $\omega(f)$ [] $\omega(AXf') \Leftrightarrow$ P may $\omega(AFf)$.

---EG*f*

We put $\omega(EGf) = \{ \omega(f) \cup \omega(EXf')$, where $f' = f \wedge EXf'\}$.

The idea of the form EG*f* is more complicated. Since it is a globally defined property, if we have a system P, it is intuitive that we should delete the states from ξP that do not satisfy *f*. Thus, we get a new ξP' = (S', R', L'), where S' $\in$ S\ {the states do not satisfy the property}. R' = R□$_{S' \times S'}$ and L' = L□$_{S'} \rightarrow 2^{AP}$. The new transition relation in the Kripke structure is that only for the new states. The labeling function only applies to the new states too.

Here *f'* is a recursive definition. When we unfold the formula, it looks like *f* and EX(*f* $\wedge$ EX*f'*) and EX(*f* $\wedge$ EX(*f* $\wedge$ EX*f'*)) and so on. It shows that the process should satisfy all of the recursive unfolded formulas in order to let the process satisfy the original formula EG*f*. As in a Kripke structure, the states in the first layer need to satisfy the formula *f* and some successive states in the second layer need to satisfy the formula *f* and so on. In terms of labeled transition system, the process need to pass the tests in $\omega(f)$ and it performs some actions to the second layer states and then the process need to pass the tests in $\omega(f)$ and so on. The process satisfies the formula iff it passes all the test cases generated from the recursively unfolded formulas.

ξP $\models$ EG*f* $\Leftrightarrow$ ξP $\models$ *f'*, where *f'* = *f* $\wedge$ EX*f'* $\Leftrightarrow$ P may $\omega(f')$ $\Leftrightarrow$ P may $\omega(f) \cup \omega(\text{EX}f')$ $\Leftrightarrow$ P may $\omega(\text{EG}f)$.

---AG*f*

We put $\omega(\text{AG}f) = \{ \omega(f) \cup \omega(\text{AX}f')$, where $f' = f \wedge \text{AX}f' \}$.

Here *f'* is a recursive definition. When we unfold the formula, it looks like *f* and AX(*f* $\wedge$ AX*f'*) and AX(*f* $\wedge$ AX(*f* $\wedge$ AX*f'*)) and so on.

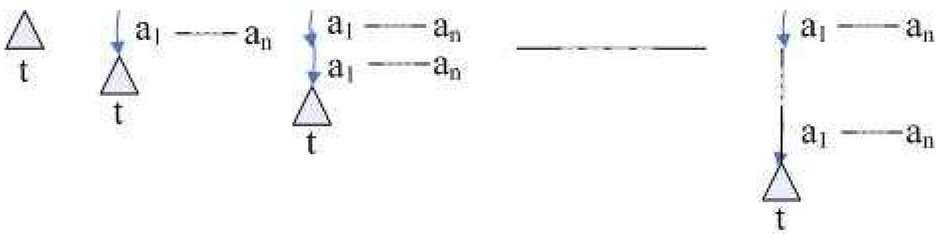


Figure 3.5: Test suite for CTL formula AGf

Figure 3.5 illustrates that the test suite is generated by combining a choice of action from the actions and the tests in $\omega(f)$. Then, combine a choice of action followed by another choice of action with the tests in $\omega(f)$ and so on till the last layer of the tree of the Kripke structure. P may ω(AGf) iff P passes each of the test cases above. P satisfies the formula AGf iff P passes the tests in $\omega(f)$, or P can perform the actions from the set A and at the next states it passes the tests in $\omega(f)$, or P can perform the actions from the set A, at the next states it can perform the actions from the set A, and then it passes the tests from $\omega(f)$ and so on. P satisfies the formula EFf iff P passes all of the test cases illustrated in figure 3.5.

$\xi P \models AGf \Leftrightarrow \xi P \models f'$, where $f' = f \wedge AXf' \Leftrightarrow$ P may $\omega(f') \Leftrightarrow$ P may $\omega(f) \cup \omega(AXf') \Leftrightarrow$ P may $\omega(AGf)$.

---E f_1 U f_2

We put $\omega(E\, f_1\, U\, f_2) = \{(\omega(f_1) \cup \omega(EXf'))\ []i;\ (\omega(f_2) \cup \omega(EXf''))$, where $f' = f_1 \wedge EXf'$ and $f'' = f_2 \wedge EXf''\}$.

The idea of the form E f_1 U f_2 is very similar to EGf. Since it is a globally defined property, if we have a system P, it is intuitive that we should delete the states from ξP that do not satisfy f_1 or f_2. Thus, we get a new $\xi P' = (S', R', L')$, where $S' \in S\backslash\{$the states do not satisfy the two properties}. R' = R□$_{S' \times S'}$ and L' = L□$_{S'} \rightarrow 2^{AP}$.

Here f' and f'' are recursive definitions. When we unfold the formula, it looks like f_1 and EX($f_1 \wedge$ EXf') or EX($f_1 \wedge$ EX($f_1 \wedge$ EXf')) and so on or with an nondeterministic internal action i then change to be f_2 and EX($f_2 \wedge$ EXf'') or EX($f_2 \wedge$ EX($f_2 \wedge$ EXf'')). It shows that the process should satisfy any of the recursive unfolded formulas in order to let the process satisfy the original formula E f_1 U f_2. As in a Kripke structure, the formula f_1 means that the states in the first layer need to satisfy the formula f_1 and then the formula EX($f_1 \wedge$ EXf') means some successive states in the second layer need to satisfy the formula f_1 and so on. At some point, some states need to satisfy the formula f_2 and from then on some successive states need to satisfy f_2 along the path later. In terms of labeled transition system, the process need to pass the tests in $\omega(f_1)$ and it performs some actions to the second layer states and then the process need to pass the tests from $\omega(f_1)$ and so on. At some point and at some states, the process need to pass the tests in $\omega(f_2)$ and it performs some actions to the successive states and then the process should pass the tests in $\omega(f_2)$ all the time later. The tests are thus nondeterministic. The process satisfies the property iff it passes either of one of the test cases generated from the recursively unfolded formulas.

ξP $\models$ E f_1 U f_2 $\Leftrightarrow$ ξP' $\models f' \vee f''$, where $f' = f_1 \wedge$ EXf' and $f'' = f_2 \wedge$ EXf'' $\Leftrightarrow$ P may $\omega(f')$ [] i; $\omega(f'')$ $\Leftrightarrow$ P may {$\omega(f_1) \cup \omega$(EXf')} [] i; {$\omega(f_2) \cup \omega$(EXf'')} $\Leftrightarrow$ P may ω(E f_1 U f_2). □

Thus we complete the proof and the conversion between CTL formulas and sequential tests.

Chapter 4
Conclusion and Open Problems

Classical methods of verifications are well studied. In this paper, we have shown the equivalence between CTL formulas and failure trace tests. Our results can be summarized as follows. According to theorem 2.1 (P may T) $\Leftrightarrow$ ($f \in ftr$(P)), where T refers to all the test cases and $f = ftr$(T); it shows that a process P passes a test T if and only if the set failure trace of the test is contained in the set of failure trace of the process. We defined the equivalence between a process or a labeled transition system and a Kripke structure in definition 3.1. In theorem 3.1, we built a function ξ that converts labeled transition system M or a process P to an equivalent Kripke structure K. As a result, we can define the semantics of CTL formulas with respect to a process rather than a Kripke structure. In theorem 3.2 we defined ψ as a function which links the sequential tests to the correspondent CTL formulas and stated that P may T iff $\xi P \models \varphi$ where $\varphi = \psi(T)$. This shows that a process P passes a test T if and only if the process satisfies the formulas that are derived from the test. We showed in theorem 3.3 that there exists a function ω which links the CTL formulas to the corresponding test

cases and stated that $\xi P \models \varphi$ iff P may T, where T = $\omega(\varphi)$. In other words, a process P satisfies the formula if and only if the process passes the tests that are derived from the formula.

We have thus developed a combined method of system verification. Using model checking and model-based testing as starting points, we approach this method by interpreting failure trace tests into CTL temporal logic formulas and CTL temporal logic formulas into failure trace tests. We strongly believe, and hope we have convinced the reader, that a combined approach as we have described in Chapter 3 is extremely helpful. It has a number of advantages over traditional approaches. Model-based testing is incomplete but has the advantage of incremental application. By contrast, model checking is complete but must be applied all at once. It has the state explosion problem [10]. CTL formulas representing loose specifications as we only specify the properties of interest, whereas algebraic models for model-based testing represent the specification quite strictly. Whether one specification is better than the other depends on the particular system under test. In our work, we convert CTL temporal logic formulas to tests and the other ways around. As a result, no matter how the system is specified (one part logically and the other algebraically), we can just apply either model checking or testing once. This is extremely important for large systems with components at different level of maturity. The canonic example is a communication protocol: the end points are algorithms that are likely to be amenable to algebraic specification, while the communication medium is something we don't know much about. It could be a direct link, a local network or something else. However, its properties are expressible in temporal logic formulas. The conversion therefore is very useful. Such a conversion can also allow the use of the fastest, most convenient and suitable method of verification.

The results of this paper are important first steps towards a more ambitious goal. We believe that this thesis opens several direction of future research. The main challenge in the method we introduced is dealing with the infinite state test cases. This problem occurs when we interpret CTL temporal logic formulas to test cases. In such a case, the test cases can have an infinite set of states. In theory, it is easy to deal with infinite state test suite, while in real practice, it is worthy of future work to eliminate infinite state or to obtain usable algorithms to run the test suite with the system under test. The tests developed here can be combined with partial application so that another interesting research direction is to find a partial application that yields total correctness at the limit and has some correctness insurance milestones along the way. Another obvious open question is whether there is an equivalence between a category of tests and the full temporal logic CTL*.

References:

[1] G. E. Hughes and M. J. Greswell. An Introduction to Modal Logic. Methuen Co.,

1968.

[2] V. Gupta. Concurrent Kripke Structures. *Proceedings of the North American Process Algebra Workshop, Cornell CS-TR-93-1369*. 1993.

[3] P. Bellini, R. Mattolini and P. Nesi. Temporal Logics for Real-time System Specification. *ACM Computing Surveys (CSUR), Volume 32 Issue 1. ACM*. Pages 12-42. March 2000.

[4] R. Gerth, D. Peled, M.Y. Vardi and P. Wolper. Simple On-the fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV. Chapman Ltd.* Pages 3-18. 1996.

[5] Rance Cleaveland and Gerald Lüttgen. Model Checking is Refinement—Relating Büchi Testing and Linear-time Temporal Logic. *ICASE Report No. 2000-14*. March 2000.

[6] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings, Symposium on Logic in Computer Science. IEEE Computer Society*. Pages 332-344. 1986.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specification. *ACM Transactions on Programming Languages and Systems, Volume 8*. Pages 244-263. 1986.

[8] E.M. Clark and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. *In Logic of Programs, Workshop, Springer-Verlag*. Pages 52-71. 1982.

[9] Stephan Merz. Model Checking—A Tutorial Overview. Institut für Informatik, Universität, München.

[10] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. Model Checking. *The MIT Press.* 1999.

[11] R. DE NICOLA and M.C.B. HENNESSY. Testing Equivalences for Processes. *Theoretical Computer Science, Volume 34.* Pages 83-133. 1984.

[12] Leslie Lamport, John Matthews, Mark Tuttle and Yuan Yu. Specifying and verifying systems with TLA+. *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop. ACM.* Pages 45-48. July 2002.

[13] Ed Brinksma, Arend Rensink and Walter Vogler. Fair Testing. In *Insup Lee and Scott A. Smolka(Eds.), "CONCUR '95: Concurrency Theory", LNCS 962, Springer*. Pages 313-327. 1995.

[14] Iain PHILLIPS. Refusal Testing. *Lecture Notes on Computer Science, Automata, Languages and Programming. Springer.* Pages 241-284. July 1986.

[15] Rom Langerak. A testing theory for LOTOS using deadlock detection. *North-Holland*. Pages 87-98. 1989.

[16] PH. SCHNOEBELEN. The Complexity of Temporal Logic Model Checking. *World Scientific Publishing Co. Ltd.* Pages 393-436. 2002.

[17] Kenneth L. McMillan. Symbolic Model Checking—an approach to the state explosion problem. CMU-CS-92-131. May 1992.

[18] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logic.

Journal of the ACM, 32: Pages 733-749, 1985.

[19] Rajeev Alur and Mihalis Yannakakis. Model Checking of Hierarchical State Machines. *ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 23 Issue 3. ACM.* Pages 175-188. May 2001.

[20] Jan Tretmans. Conformance Testing with labeled Transition System: Implementation Relations and Test Generation. *Computer Network and ISDN Systems.* Pages 49-79. 1996.

[21] Thomas J. Schriber, Jerry Banks, Andrew F. Seila, Ingolf Ståhl, Averill M. Law, Richard G. Born. Simulation textbooks. *WSC '03: Proceedings of the 35th conference on Winter simulation: driving innovation. Winter Simulation Conference. ACM.* Pages 1952-1963. December 2003.

[22] Krzysztof Pawlikowski. Steady-state simulation of queueing processes: survey of problems and solutions. *ACM computing Surveys (CSUR)*, *Volume 22 Issue 2. ACM.* Pages 123-170. June 1990.

[23] Jan Tretmans. A Formal Approach to Conformance Testing. University of Twente. 1992.

[24] Guang Yuan LI and Zhi Song TANG. Linear temporal logic with clocks and real-time. *Journal of software Vol 13, No. 1*. Pages 193-202. 2002.

[25] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science Volume 13*. Pages 45-60. 1981.

[26] Michael Huth and Mark Ryan. Logic in Computer Science (Second Edition). Cambridge University Press. 2004.

[27] Duminda Wijesekera, Paul Ammann, Lingya Sun, Gordon Fraser. Relating counterexamples to test cases in CTL model checking specifications. *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing.* ACM. Pages 75-84. July 2007.

[28] P. Ammann, W. Ding and D. Xu. Using a Model Checker to Test Safety Properties. In *Proceedings of the 7th International Conference on Engineering of Complex Computer Science (ICECCS 2001)*. IEEE. Pages 212-221, Skovde, Sweden. 2001.

[29] P. E. Ammann, P. E. Black and W. Majurski. Using Model Checking to Generate Tests from Specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM' 98)*. Pages 46-54, IEEE Computer Society, 1998.

[30] Stefan D. Bruda. Preorder Relations. In *Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, et al., eds., Model-Based Testing of Reactive Systems: Advanced Lectures, Springer Lecture Notes in Computer Science 3472*. Pages 117-150. 2005.

[31] Valery Tschaen. Test Generation Algorithms based on Preorder Relations. In *Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, et al., eds., Model-Based Testing of Reactive Systems: Advanced Lectures, Springer Lecture Notes in Computer Science 3472*. Pages 151-171. 2005.

[32] E. Brinksma, G. Scollo and C. Steenbergen. LOTOS Specifications, their implementations and their tests. In *G. v. Bochmann, B. Sarikaya, Proc. IFIP 6.1 Sixth St. Jovite*. Pages: 349-360. 1987.

[33] Batsayan Das, Dipankar Sarkar and Santanu Chattopadhyay. Model Checking on State Transition Diagram. *ASP-DAC' 04. Proceedings of the 2004 conference on Asia South Pacific design automation: electronic design and solution fair. IEEE Press.* Pages 412-417. January 2004.

[34] Mordechai Ben-Ari, Zohar Manna and Amir Pnueli. The temporal logic of branching time. *POPL' 81. Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Volume 20. ACM.* Pages 207-226. January 1981.

[35] Rocco De Nicola and Frits Vaandrager. Three logics for branching bisimulation. *Journal of the ACM (JACM), Volume 42, Issue 2. ACM.* Pages 438-487. March 1995.

[36] Tiejun GAO. An Analytic Semantics of CSP. *Fundam. Inform. Number 2, Volume 15.* 1991.