Technical Report 2009-002
Model Checking Is Refinement: Computation Tree Logic Is Equivalent to Failure Trace Testing*

Stefan D. Bruda and Zhiyu Zhang
Department of Computer Science
Bishop's University
Sherbrooke, Quebec J1M 1Z7, Canada
email: {bruda|zzhang}@cs.ubishops.ca

22 August 2009

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 computation tree 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 its associated preorders. The most fine-grained preorder beside bisimulation (mostly of theoretical importance) is based on failure traces. We show that these two most powerful variants are equivalent; that is, we show that for any failure trace test there exists a CTL formula equivalent to it, and the other way around. Our result allows for parts of a large system to be specified logically while other parts are specified algebraically, thus combining the best of the two (logic and algebraic) worlds.

Keywords: model checking, model-based testing, stable failure, failure trace, failure trace preorder, temporal logic, computation tree logic, labelled transition system, Kripke structure.

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, the Internet, and telephone networks. Highway and air traffic control 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

*This research was supported by the Natural Sciences and Engineering Research Council of Canada. Part of this research was also supported by Bishop's University.

is acceptable. Indeed, 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 four kinds of methods of verification: Testing (or simulation) [9, 13], deductive verification [11], formal testing [6, 14] and model checking [4, 5]. Simulation is a traditional, non-formal method of verification and it is still the most widely used: We inject signals into the system, run it, and then observe the resulting signals; we then 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. Deductive verification uses axioms and proof rules to prove mathematically the correctness of the system; it is a time consuming process that can only be performed by experts with education in logical reasoning and considerable experience. (Formal) testing is a method in which test cases are derived systematically and automatically 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. The specification of the system is described by temporal logic formulae. We then construct a model of the system and we label all the states in the model where the formulae hold to see whether the initial states of the model satisfy the formulae.

Model checking is a complete verification technique, but 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. In addition, model checking is 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 a potentially 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 represented as a labelled transition system or finite automaton, so one has to more or less specify the whole system.

In model checking we specify the system using temporal logic, which come in in three varieties: CTL*, CTL and LTL. In this paper, we concentrate on CTL. We also concentrate of arguably the most powerful method of model-based testing, namely failure trace testing [8]. Beside power, failure trace testing also introduces a set (of sequential tests) that is sufficient to assess the failure trace relation.

Some properties of a system may be naturally specified using temporal logic, while others may be specified using finite automata or labelled transition systems. Such a mixed specification could be given by somebody else, but most often algebraic specifications are just more convenient for some components while logic specifications are more suitable for others. Consider some properties that are expressed as CTL formulae 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. The result is not automatically correct. We do not even 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 and then check it at once. We describe in this paper precisely such a conversion. Indeed, we show that for each CTL formula, there is an equivalent failure trace test suite, and the other way around.

We are opening the domain of combined (algebraic and logic) methods of formal system verification. The advantages of such a combined method stem from the above considerations but also from the lack of compositionality of model checking (which can thus be side-stepped by switching to algebraic specifications), from the lack of completeness of model-based testing (which can be side-stepped by switching to model checking), and from the potentially attractive feature of model-based testing of incremental application of a test suite insuring correctness to a certain degree (which model-checking lacks).

2 Preliminaries

Temporal Logic and Model Checking. A specification can be described by a temporal logic formula. The model of the system under test (that should have the same properties as the system) is a Kripke structure. The goal of model checking is then to find the set of all states in the Kripke structure that satisfy the given logic formula. The system then satisfies the specification provided that all the (designated) initial states satisfy the logic formula.

Formally, a *Kripke structure* K over a set AP or atomic propositions is a tuple $(S, S_0, \rightarrow, L)$, where S is the set of states, S_0 is the set of initial states, $\rightarrow \subseteq S \times S$ is the transition relation, and $L : S \rightarrow 2^{\mathsf{AP}}$ is a valuation that specifies which atomic propositions are true in each state. We usually write $s \rightarrow t$ instead of $(s,t) \in \rightarrow$. It is required that $\rightarrow$ be total, so for all $s \in S$ there exists $t \in S$ such that $s \rightarrow t$, but such a requirement is easily established using a "sink" state. A *path* π in a Kripke structure is a nonempty finite or infinite sequence $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$ such that $s_i \rightarrow s_{i+1}$ for all $i \geq 0$. The path starts from state s_0. For any Kripke structure K and state s_0 we can build a computation tree with nodes labelled with states and the root being labelled s_0, such that (s,t) is an edge in the tree iff $s \rightarrow t$.

CTL* [7], CTL [3] (computation tree logic) and LTL [10] (linear-time temporal logic) are all temporal logics. CTL* is the most general. Both LTL and CTL are strict subsets of CTL*. CTL* formulae are used to describe the properties of computation trees. CTL* formulae consist of path quantifiers and temporal operators. There are two path quantifiers: A and E, that describe the branching structure of the computation tree as the states in the 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", F "eventually", G "always" or "globally", U "until", and R "releases".

We have two kinds of formulae in CTL*: state formulae and path formulae. State formulae hold in a specific state, while path formulae hold along a specific path and are put together using path quantifiers. CTL, a sub-logic of 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 one of the path quantifiers A or E. The syntax of CTL state formulae can thus be defined as follows: With a ranging over AP, and f, f_1, f_2 ranging over state formulae,

$$\begin{aligned} f \;=\; & \top \mid \bot \mid a \mid \neg f \mid f_1 \wedge f_2 \mid f_1 \vee f_2 \mid \\ & \mathsf{AX}\, f \mid \mathsf{AF}\, f \mid \mathsf{AG}\, f \mid \mathsf{A}\, f_1\, \mathsf{U}\, f_2 \mid \mathsf{A}\, f_1\, \mathsf{R}\, f_2 \mid \\ & \mathsf{EX}\, f \mid \mathsf{EF}\, f \mid \mathsf{EG}\, f \mid \mathsf{E}\, f_1\, \mathsf{U}\, f_2 \mid \mathsf{E}\, f_1\, \mathsf{R}\, f_2 \end{aligned}$$

The truth value of a CTL formula is defined in terms of Kripke structures. The notation $K, s \vDash f$ means that in a Kripke structure K, formula f is true in state s. The notation $K, \pi \vDash f$ means that in a Kripke structure K, formula f is true along the path π. The meaning of $\vDash$ is defined inductively. f and g are state formulae unless stated otherwise.

1. $K, s \vDash \top$ is true and $K, s \vDash \bot$ is false for any state s in any Kripke structure K.
2. $K, s \vDash a, a \in \mathsf{AP}$ iff $a \in L(s)$.
3. $K, s \vDash \neg f$ iff $\neg(K, s \vDash f)$.
4. $K, s \vDash f \wedge g$ iff $K, s \vDash f$ and $K, s \vDash g$.
5. $K, s \vDash f \vee g$ iff $K, s \vDash f$ or $K, s \vDash g$.
6. $K, s \vDash \mathsf{E}\ f$ for some path formula f iff there is a path $\pi = s \to s_1 \to s_2 \to \cdots \to s_i$ such that $K, \pi \vDash f$.
7. $K, s \vDash \mathsf{A}\ f$ for some path formula f iff $K, \pi \vDash f$ for all paths $\pi = s \to s_1 \to s_2 \to \cdots \to s_i$.

We use π^i to denote the i-th state of a path π (with the first state being state 0, or π^0). The definition of $\vDash$ for path formulae is:

1. $K, \pi \vDash \mathsf{X}\ f$ iff $k, \pi^1 \vDash f$.
2. $K, \pi \vDash f\ \mathsf{U}\ g$ iff there exists $j \geq 0$ such that $K, \pi^j \vDash g$ and $K, \pi^k \vDash g$ for all $k \geq j$, and for all $i < j$ $K, \pi^i \vDash f$.
3. $K, \pi \vDash f\ \mathsf{R}\ g$ iff for all $j \geq 0$, if $K, \pi^i \nvDash f$ for every $i < j$ then $K, \pi^j \vDash g$.

Stable Failures. CTL semantics is defined over Kripke structures, where each state is labelled with atomic propositions. By contrast, the common model used for system specifications in model-based testing is the labelled transition system (LTS), where the labels or formulae are associated with the transitions instead. In model-based testing [6, 14] test cases 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 process we need to check for correctness. The model, an LTS, is usually an abstract representation of the system's desired behaviours. Some times finite automata are used instead, which are a particular, finite case of LTS.

An LTS is a tuple $M = (S, A, \to, s_0)$ 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 visible (or observable) events (or actions). $\to \subseteq S \times (A \cup \{\tau\}) \times S$ 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 $(p, a, q) \in \to$. The sets of states and transitions can also be considered global, case in which an LTS is completely defined by its initial state. We therefore blur whenever convenient the distinction between an LTS and a state, calling them both "processes".

A relevant set of tests run parallel with the system under test and synchronize with it over visible actions. A run of a test t and a process p represents a possible sequence of states and

actions of t and p running synchronously. We have in fact a set of possible runs $\text{Obs}(t, p)$. If we have $r \in \text{Obs}(t, p)$ then the result of t testing p may be the run r. We take the outcomes of a particular run of a test as being success ($\top$) or failure ($\bot$). The outcome of run r is $\bot$ if r is unsuccessful, or r contains a state q such that $q \uparrow$ (q diverges) and q is not preceded by a successful state; $q \uparrow$ iff there exists $q_1, q_2, \ldots q_k, \ldots$, such that $q \xrightarrow{\tau} q_1 \xrightarrow{\tau} q_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} q_k \xrightarrow{\tau} \cdots$; that is, an infinite path of τ actions starts from the state q.

Since some tests or processes 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. The may powerdomain identifies that a process may pass a test in order to be regarded as successful, and the must powerdomain identifies tests that must be successful.

To analyze processes, it is necessary to consider those sequences of events that can be observed as the interface of the processes. A trace 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 over $A^{\checkmark} = A \cup \{\checkmark\}$ that might possibly be recorded. A represents the set of actions of the process. The $\checkmark$ (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. The set of all possible traces of a process p is denoted by $\text{traces}(p)$.

A *path* (or *run*) π starting from state p' is a sequence $p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots p_{k-1} \xrightarrow{a_k} p_k$ with $k \in \mathbb{N}$ such that $k = 0$, or $p_0 = p'$ and $p_{i-1} \xrightarrow{a_i} p_i$ for all $0 < i \leq k$. We use $|\pi|$ to refer to k, the length of π. If $|\pi| < \infty$, then we say that π is finite. The trace of π is the sequence $\text{trace}(\pi) = (a_i)_{i \in \{a_i : 0 < i \leq |\pi|, a_i \neq \tau\}} \in A^*$. 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.

The notation $p \stackrel{w}{\Longrightarrow} p'$ states that there exists a sequence of transitions whose initial state is p, whose final state is p', and whose visible transitions form the sequence w. Since the termination can only occur at the end of an execution, the tick $\checkmark$ may happen in w only at the end. The notation $p \stackrel{w}{\Longrightarrow}$ stands for $\exists p' : p \stackrel{w}{\Longrightarrow} p'$. The traces of a process p then can be defined as $\text{traces}(p) = \{w : p \stackrel{w}{\Longrightarrow}\}$. In all, the finite traces of a process p can be defined as $\text{Fin}(p) = \{tr \in \text{traces}(p) : \delta(tr) \subseteq A^{\checkmark} \wedge |tr| \in \mathbb{N} \wedge \checkmark \notin \delta(\text{init}(tr))\}$, where $\text{init}(tr)$ contains exactly all the proper prefixes of tr and $\delta(tr)$ refers to the set of all the elements in the trace tr. $|tr|$ refers to the length of the trace tr.

A process p which can make no internal progress (that is, it has no outgoing internal actions) is said to be *stable* [12], defined as $p \downarrow = \neg(\exists p' \neq p : p \stackrel{\varepsilon}{\Longrightarrow} p')$. A stable process p can always respond in some way to the offer of a set of event $X \subseteq A^{\checkmark}$ if there is at least one $a \in X$ that p can execute. If there is no such an a then p will *refuse* the entire set X, denoted by p ref X. Generally, $p \text{ ref } X = \forall a \in X : \neg(\exists p' : p \stackrel{\varepsilon}{\Longrightarrow} p' \wedge p' \downarrow \wedge p' \xrightarrow{a})$.

The refusal of an offer set X might happen during an execution of the process p. Such a refusal 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 the refusal happens. The observation (w, X) is called a *stable failure* [12] of p whenever $\exists p^w : p \stackrel{w}{\Longrightarrow} p^w \wedge p^w \downarrow \wedge p^w \text{ ref } X$. The set of stable failures of p is then $\text{SF}(p) = \{(w, X) : \exists p^w : p \stackrel{w}{\Longrightarrow} p^w \wedge p^w \downarrow \wedge p^w \text{ ref } X\}$. 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. Let p and q be two processes. Then $p \sqsubseteq_{\text{SF}} q$ iff $\text{Fin}(p) \subseteq \text{Fin}(q)$

and $\mathrm{SF}(p) \subseteq \mathrm{SF}(q)$. We call $\sqsubseteq_{\mathrm{SF}}$ the *stable failure preorder*: p implements q iff the set of finite traces of p is included in the set of finite traces of q, and the set of stable failures of p is also included in the set of stable failures of q. Given the preorder, one can easily define the stable failure testing equivalence $\simeq_{\mathrm{SF}}$: $p \simeq_{\mathrm{SF}} q$ iff $p \sqsubseteq_{\mathrm{SF}} q$ and $q \sqsubseteq_{\mathrm{SF}} p$. Other preorders can be defined; in general, $\sqsubseteq_{\mathrm{SF}}$ is one of the finest preorders [2].

Failure Trace Testing. In what follows we use the notation $\mathrm{init}(p) = \{a \in A : p \stackrel{a}{\Longrightarrow}\}$. A failure trace f is a string of the form $f = A_0 a_1 A_1 a_2 \ldots A_n a_n$, $n \geq 0$, with $a_i \in A^*$ and $A_i \subseteq A$ (the sets of refusals). Let p be a process such that $p \stackrel{\varepsilon}{\Longrightarrow} p_0 \stackrel{a_1}{\Longrightarrow} p_1 \stackrel{a_2}{\Longrightarrow} \cdots \stackrel{a_n}{\Longrightarrow} p_n$; $f = A_0 a_1 A_1 a_2 \ldots A_n a_n$ is then a failure trace of p whenever:

- If $p_i \stackrel{\tau}{\longrightarrow}$ then then $A_i = \emptyset$, which means that p_i is not a stable state and thus refuses an empty set of events by definition.
- If $\neg(p_i \stackrel{\tau}{\longrightarrow})$, then $A_i \subseteq (A \setminus \mathrm{init}(p_i))$; for a stable state the failure trace refuses any set of events that cannot be performed in that state (including the empty set).

In all, we get a failure trace of p by taking a trace of p and then putting into it refusal sets after stable states.

Systems and tests can be concisely described using the testing language TLOTOS [1, 8], which will also be used in this paper. A is the countable set of observable actions, ranged over by a. The set of processes or tests is ranged over by t, t_1 and t_2, while T ranges over the sets of tests. The syntax of TLOTOS is defined as follows:

$$t = \mathrm{stop} \mid a; t_1 \mid \mathbf{i}; t_1 \mid \theta; t_1 \mid \mathrm{pass} \mid t_1 \,\Box\, t_2 \mid \Sigma T.$$

The semantics of TLOTOS is then defined as follows:

1. inaction (stop): no rules.
2. action prefix: $a; t_1 \stackrel{a}{\longrightarrow} t_1$ and $\mathbf{i}; t_1 \stackrel{\tau}{\longrightarrow} t_1$
3. deadlock detection: $\theta; t_1 \stackrel{\theta}{\longrightarrow} t_1$.
4. successful termination: $\mathrm{pass} \stackrel{\gamma}{\longrightarrow} \mathrm{stop}$.
5. choice: with $g \in A \cup \{\gamma, \theta, \tau\}$,
$$\frac{t_1 \stackrel{g}{\longrightarrow} t_1'}{t_1 \,\Box\, t_2 \stackrel{g}{\longrightarrow} t_1' \qquad t_2 \,\Box\, t_1 \stackrel{g}{\longrightarrow} t_1'}$$
6. generalized choice: with $g \in A \cup \{\gamma, \theta, \tau\}$,
$$\frac{t_1 \stackrel{g}{\longrightarrow} t_1'}{\Sigma(\{t_1\} \cup t) \stackrel{g}{\longrightarrow} t_1'}$$

The symbol γ signals the successful completion of a test, while θ is the deadlock detection label. Any process (or LTS) is also described as a TLOTOS process not containing γ and θ. A failure trace test on the other hand is a full TLOTOS process, i.e., may contain γ and δ. A test runs in parallel with the system under test according to the parallel composition operator $\|_\theta$. This operator also defines the semantics of θ as the lowest priority action:

$$\frac{p \xrightarrow{\tau} p'}{p\|_\theta t \xrightarrow{\tau} p'\|_\theta t} \qquad \frac{t \xrightarrow{\tau} t'}{p\|_\theta t \xrightarrow{\tau} p'\|_\theta t}$$

$$\frac{t \xrightarrow{\gamma} \text{stop}}{p\|_\theta t \xrightarrow{\gamma} \text{stop}} \qquad \frac{p \xrightarrow{a} p' \qquad t \xrightarrow{a} t'}{p\|_\theta t \xrightarrow{a} p'\|_\theta t'}\ a \in A$$

$$\frac{t \xrightarrow{\theta} t' \qquad \neg\exists x \in A \cup \{\tau, \gamma\} : p\|_\theta t \xrightarrow{x}}{p\|_\theta t \xrightarrow{\theta} p\|_\theta t'}$$

Given that both processes and tests can be nondeterministic we have a set $\Pi(p\|_\theta t)$ of possible runs of a process and a test. The outcome of a particular run $\pi \in \Pi(p\|_\theta t)$ of a test t and a process under test p is success ($\top$) whenever the last symbol in trace(π) is γ, and failure ($\bot$) otherwise. All the possible outcomes of all the runs in $\Pi(p\|_\theta t)$ are denoted by $\text{Obs}(p, t)$. One distinguishes the possibility and the inevitability of success for a test. Indeed, we write p may t iff $\top \in \text{Obs}(p, t)$, and p must t iff $\{\top\} = \text{Obs}(p, t)$.

The set $\mathcal{ST}$ of of sequential tests is defined as follows: pass $\in \mathcal{ST}$, if $t \in \mathcal{ST}$ then $a; t \in \mathcal{ST}$ for any $a \in A$, and if $t \in \mathcal{ST}$ then $\Sigma\{a; \text{stop} : a \in A'\} \,\square\, \theta; t \in \mathcal{ST}$ for any $A' \subseteq A$.

A bijection between failure traces and sequential tests exists. For a sequential test t the failure trace $\text{ftr}(t)$ is defined inductively as follows: $\text{ftr}(\text{pass}) = \emptyset$, $\text{ftr}(a; t') = a\ \text{ftr}(t')$, and $\text{ftr}(\Sigma\{a; \text{stop} : a \in A'\} \,\square\, \theta; t') = A\ \text{ftr}(t')$. Conversely, let f be a failure trace. Then we can inductively define the sequential test $\text{st}(f)$ as follows: $\text{st}(\emptyset) = \text{pass}$, $\text{st}(af) = a\ \text{st}(f)$, and $\text{st}(Af) = \Sigma\{a; \text{stop} : a \in A\} \,\square\, \theta; \text{st}(f)$. For all failure traces f we have that $\text{ftr}(\text{st}(f)) = f$, and for all tests t we have $\text{st}(\text{ftr}(t)) = t$.

We can thus convert a process-oriented testing equivalence into a testing-based testing equivalence. Indeed there exists a successful run of p in parallel with the test t, iff f is a failure trace of both p and t. We define $p \sqsubseteq_{\text{FT}} q$ iff $\text{ftr}(p) \subseteq \text{ftr}(q)$. The process-based testing equivalences on both stable failures and failure traces express exactly the same idea.

Proposition 1 [8] *Let p be a process, t a sequential test, and f a failure trace. Then p may t iff $f \in \text{ftr}(p)$, where $f = \text{ftr}(t)$.*

Let p_1 and p_2 be processes. Then $p_1 \sqsubseteq_{\text{SF}} p_2$ iff $p_1 \sqsubseteq_{\text{FT}} p_2$ iff p_1 may $t \implies p_2$ may t for all failure trace tests t iff $\forall t' \in \mathcal{ST} : p_1$ may $t' \implies p_2$ may t'.

Let t be a failure trace test. Then there exists $T(t) \subseteq \mathcal{ST}$ such that p may$t$ iff $\exists t' \in T(t) : p$ may t'.

Note in passing that unlike other preorders, $\sqsubseteq_{\text{SF}}$ can be in fact characterized in terms of may testing only; the must operator needs not be considered any further.

3 CTL Is Equivalent to Failure Trace Testing

This section presents the main contribution of this paper. Indeed, we analyze the equivalence between labelled transition systems and Kripke structures and then we establish the equivalence between failure trace tests and CTL formulae. We establish such equivalence by constructing two functions that convert failure trace tests to CTL* formulae and the other way around.

3.1 Equivalence between LTS and Kripke Structures

We first define an LTS satisfaction operator similar to the one on Kripke structures in a natural way. Informally, the outgoing transitions of an LTS state are the propositions that hold in that state.

Definition 1 SATISFACTION FOR PROCESSES: *A process p satisfies $a \in A$, written by abuse of notation $p \vDash f$, iff $p \xrightarrow{a}$. That p satisfies some (general) CTL* state formula is defined inductively as follows: Let f and g be some state formulae unless stated otherwise; then,*

1. $p \vDash \top$ *is true and* $p \vDash \bot$ *is false for any process* p.
2. $p \vDash \neg f$ *iff* $\neg(p \vDash f)$.
3. $p \vDash f \wedge g$ *iff* $p \vDash f$ *and* $p \vDash g$.
4. $p \vDash f \vee g$ *iff* $p \vDash f$ *or* $p \vDash g$.
5. $p \vDash \mathsf{E}\ f$ *for some path formula* f *iff there is a path* $\pi = p \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$ *such that* $\pi \vDash f$.
6. $p \vDash \mathsf{A}\ f$ *for some path formula* f *iff* $p \vDash f$ *for all paths* $\pi = p \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$.

We use π^i to denote the i-th state of a path π (with the first state being state 0, or π^0). The definition of $\vDash$ for LTS paths is:

1. $\pi \vDash \mathsf{X}\ f$ *iff* $\pi^1 \vDash f$.
2. $\pi \vDash f\ \mathsf{U}\ g$ *iff there exists* $j \geq 0$ *such that* $\pi^j \vDash g$ *and* $\pi^k \vDash g$ *for all* $k \geq j$, *and* $\pi^i \vDash f$ *for all* $i < j$.
3. $\pi \vDash f\ \mathsf{R}\ g$ *iff for all* $j \geq 0$, *if* $\pi^i \nvDash f$ *for every* $i < j$ *then* $\pi^j \vDash g$.

We also need to define a weaker satisfaction operator for CTL. Such an operator is similar to the original, but is defined over a set of states rather than a single state. By abuse of notation we denote this operator by $\vDash$ as well.

Definition 2 SATISFACTION OVER SETS OF STATES: *Consider a Kripke structure $K = (S, S_0, R, L)$ over* AP. *For some set $Q \subseteq S$ and some CTL state formula f we define $K, Q \vDash f$ as follows, with f and g state formulae unless stated otherwise:*

1. $K, Q \vDash \top$ *is true and* $K, Q \vDash \bot$ *is false for any set* Q *in any Kripke structure* K.
2. $K, Q \vDash a$ *iff* $a \in L(s)$ *for some* $s \in Q$, $a \in \mathsf{AP}$.

3. $K, Q \vDash \neg f$ *iff* $\neg(K, Q \vDash f)$.

4. $K, Q \vDash f \wedge g$ *iff* $K, Q \vDash f$ *and* $K, Q \vDash g$.

5. $K, Q \vDash f \vee g$ *iff* $K, Q \vDash f$ *or* $K, Q \vDash g$.

6. $K, Q \vDash \mathsf{E}\, f$ *for some path formula* f *iff for some* $s \in Q$ *there exists a path* $\pi = s \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_i$ *such that* $K, \pi \vDash f$.

7. $K, Q \vDash \mathsf{A}\, f$ *for some path formula* f *iff for some* $s \in Q$ *it holds that* $K, \pi \vDash f$ *for all paths* $\pi = s \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_i$.

With these definition we can introduce the following equivalence relation between Kripke structures and LTS:

Definition 3 EQUIVALENCE BETWEEN KRIPKE STRUCTURES AND LTS: *Given a Kripke structure* K *and a set of states* Q *of* K*, the pair* K, Q *is equivalent to a process* p*, written* $K, Q \simeq p$ *(or* $p \simeq K, Q$*), iff for any CTL* formula* f $K, Q \vDash f$ *iff* $p \vDash f$.

It is easy to see that the relation $\simeq$ is indeed an equivalence relation. This equivalence has the useful property that given an LTS it is easy to construct its equivalent Kripke structure.

Theorem 1 *There exists an algorithmic function* ξ *which converts a labelled transition system* p *into a Kripke structure* K *and a set of states* Q *such that* $p \simeq (K, Q)$.

Specifically, for any labelled transition system $p = (S, A, \rightarrow, s_0)$*, its equivalent Kripke structure* K *is defined as* $K = (S', Q, R', L')$ *where:*

1. $S' = \{\langle s, x\rangle : s \in S, x \subseteq \text{init}(s)\}$.

2. $Q = \{\langle s_0, x\rangle \in S'\}$.

3. R' *contains exactly all the transitions* $(\langle s, N\rangle, \langle t, O\rangle)$ *such that* $\langle s, N\rangle, \langle t, O\rangle \in S'$*, and*

   (a) *for any* $n \in N$, $s \stackrel{n}{\Longrightarrow} t$,

   (b) *for some* $q \in S$ *and for any* $o \in O$, $t \stackrel{o}{\Longrightarrow} q$*, and*

   (c) *if* $N = \emptyset$ *then* $O = \emptyset$ *and* $t = s$ *(these loops ensure that the relation* R' *is complete).*

4. $L' : S' \rightarrow 2^{\mathsf{AP}}$ *such that* $L'(s, x) = x$*, where* $\mathsf{AP} = A$.

Figure 1 illustrates graphically how we convert a labelled transition system to its equivalent Kripke structure. As illustrated in the figure, we combine each state in the labelled 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 labelled transition system. The labeling function in the equivalent Kripke structure links the actions to their relevant states. An LTS states in split into multiple Kripke states whenever it can evolve differently by performing different actions (like state p in the figure).

Using such a conversion we can define the semantics of CTL* formulae with respect to a process rather than Kripke structure. One problem—that required the new satisfaction operator for

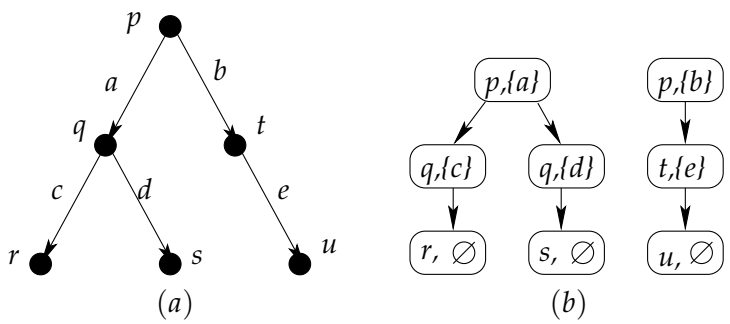

Figure 1: The conversion from LTS (a) to its equivalent Kripke structure (b). Each state of the Kripke structure (b) is labelled with the LTS state it came from, and the set of propositions that hold in that Kripke state (the loops for states satisfying no proposition are not shown).

sets of Kripke states as defined in Definition 2—is introduced by the fact that one state of a process can generate multiple initial Kripke states. We believe that the weaker satisfaction operator from Definition 2 is introduced without loss of generality and can be worked around by such mechanisms as considering processes with one outgoing transition (a "start" action) followed by their normal behaviour.

Proof of Theorem 1. The proof relies on the properties of the syntax and semantics of CTL* formulae and is done by structural induction.

For the basis of the induction, we note that $\top$ is true for any process and for any state in any Kripke structure. $p \vDash \top$ iff $K, Q \vDash \top$ is therefore immediate. The same goes for $\bot$ (no process and no state in any Kripke structure satisfy $\bot$). $p \vDash a$ iff $\xi(p) = K, Q \vDash a$ by the definition of ξ; indeed, $a \in \text{init}(p)$ (so that $p \vDash a$) iff $a \in x$ for some state $\langle s, x\rangle \in Q$ that is, $a \in L'(\langle s, x\rangle)$.

On to the inductive step. $p \vDash f'$ iff $\xi(p) \vDash f'$ for any formula f' by induction hypothesis, so we take $f' = \neg f$ and so $p \vDash \neg f$ iff $\xi(p) \vDash \neg f$.

Suppose that $p \vDash f$ and [or] $p \vDash g$ (so that $p \vDash f \wedge g$ [$p \vDash f \vee g$]). This is equivalent by induction hypothesis to $\xi(p) \vDash f$ and [or] $\xi(p) \vDash g$, that is, $\xi(p) \vDash f \wedge g$ [$\xi(p) \vDash f \vee g$], as desired.

Let now π be a path $\pi = p \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ starting from a process p. According to the definition of ξ, all the equivalent paths in the Kripke structure $\xi(p)$ have the form $\pi' = \langle p, A_0\rangle \rightarrow \langle s_1, A_1\rangle \rightarrow \langle s_2, A_2\rangle \rightarrow \cdots \rightarrow \langle s_n, A_n\rangle$, such that $a_i \in A_i$ for all $0 \leq i < n$. Clearly, such a path π' exists. Moreover, given some path of form π', a path of form π also exists (because no path in the Kripke structure comes out of the blue; instead all of them come from paths in the original process). By abuse of notation we write $\xi(\pi) = \pi'$, with the understanding that this incarnation of ξ is not necessarily a function (it could be a relation) but is complete (there exists a path $\xi(\pi)$ for any path π). With this notion of equivalent paths we can now proceed to path formulae.

Consider the formula $\mathsf{X}f$ such that some path π satisfies it. Whenever $\pi \vDash \mathsf{X}f$, $\pi^1 \vDash f$ and therefore $\xi(\pi)^1 \vDash f$ (by inductive assumption, for indeed f is a state, not a path formula) and therefore $\xi(\pi) \vDash \mathsf{X}f$, as desired. Conversely, $\xi(\pi) \vDash \mathsf{X}f$, that is, $\xi(\pi)^1 \vDash f$ means that $\pi^1 \vDash f$ by inductive assumption, and so $\pi \vDash \mathsf{X}f$.

The proof for F, G, U, and R operators proceed similarly. Whenever $\pi \vDash \mathsf{F}\, f$, there is a state π^i such that $\pi^i \vDash f$. By induction hypothesis then $\xi(\pi)^i \vDash f$ and so $\xi(\pi) \vDash \mathsf{F}\, f$. The other way (from $\xi(\pi)$ to π) is similar. The G operator requires that all the states along π satisfy f, which imply that

all the states in any $\xi(\pi)$ satisfy f, and thus $\xi(\pi) \vDash \mathsf{G}\ f$ (and again things proceed similarly in the other direction). In all, the induction hypothesis established a bijection between the states in π and the states in (any) $\xi(\pi)$. This bijection is used in the proof for U and R just as it was used in the above proof for F and G. Indeed, the states along the path π will satisfy f or g as appropriate for the respective operator, but this translates in the same set of states satisfying f and g in $\xi(\pi)$, so the whole formula (using U or R holds in π iff it holds in $\xi(\pi)$.

Finally, given a formula $\mathsf{E}\ f$, $p \vDash \mathsf{E}\ f$ implies that there exists a path π starting from p that satisfies f. By induction hypothesis there is then a path $\xi(\pi)$ starting from $\xi(p)$ that satisfies f (there is at least one such a path) and thus $\xi(p) \vDash \mathsf{E}\ f$. The other way around is similar, and so is the proof for $\mathsf{A}\ f$ (all the paths π satisfy f so all the path $\xi(\pi)$ satisfy f as well; there are no supplementary paths, since all the paths in $\xi(p)$ come from the paths in p). □

3.2 From Failure Trace Tests to CTL Formulae

Let $\mathcal{P}$ be the set of all processes, $\mathcal{T}$ the set of all tests, and $\mathcal{F}$ the set of all CTL formulae. We are now ready to show that any failure trace test can be converted to an equivalent CTL formula.

Theorem 2 *There exists a function $\psi : \mathcal{T} \to \mathcal{F}$ such that for any process p, p may t iff $\xi(p) \vDash \psi(t)$.*

Proof. The proof is done by induction on tests. In the process we also construct (inductively) the function ψ.

We put $\psi(\text{pass}) = \top$. Any process passes pass and any Kripke structure satisfies $\top$, thus it is immediate that p may pass iff $\xi(p) \vDash \top = \psi(\text{pass})$. Similarly, we put $\psi(\text{stop}) = \bot$. No process passes stop and no Kripke structure satisfies $\bot$.

On to the induction steps now. We put $\psi(\mathbf{i}; t) = \psi(t)$: an internal action in a test is not seen by the process under test by definition. We then put $\psi(a; t) = a \wedge \mathsf{EX}\ \psi(t)$. We note that p may $(a; t)$ iff p may a and p' may t for some $p \xrightarrow{a} p'$. Now, p may a iff $\xi(p) \vDash a$ by the construction of ξ, and also p' may t iff $\xi(p') \vDash \psi(t)$ by induction hypothesis. By Theorem 1, when we convert p to an equivalent Kripke structure $\xi(p)$ we take as new states the original states together with their outgoing actions. So once we are (in $\xi(p)$) in a state that satisfies a, all the next states of that state correspond to the states following p after executing a. Therefore, $\mathsf{X}(\psi(t))$ is satisfied in exactly those states in which t must succeed. Thus p may $a; t$ iff $\xi(p) \vDash a \wedge \mathsf{EX}\psi(t)$. For illustration purposes note that in Figure 1 the initial state p becomes two initial states $(p, \{a\})$ and $(p, \{b\})$; the next state of the state satisfying the property a in the Kripke structure contains only q (and never t).

Note now that $\Box$ is just syntactical sugar, for indeed $t_1 \Box t_2$ is perfectly equivalent with $\Sigma\{t_1, t_2\}$. We put[1] $\psi(\Sigma T) = \bigvee_{t \in T} \psi(t)$. p may ΣT iff p may t for at least one $t \in T$ iff $\xi(p) \vDash \psi(t)$ for at least one $t \in T$ (by induction hypothesis) iff $\xi(p) \vDash \bigvee_{t \in T} \psi(t)$.

We finally get to consider θ. Note first that whenever θ does not participate in a choice it behaves exactly like $\mathbf{i}$, so we assume without loss of generality that θ appears only in choice constructs. We also assume without loss of generality that every choice contains at most one top-level θ, for indeed $\theta; t_1 \Box \theta; t_2$ is equivalent with $\theta; (t_1 \Box t_2)$. For convenience let $T = \{t_1, t_2, \ldots, t_n\}$. We put $\psi(\Sigma T \Box \theta; t) = (\psi(t_1) \vee \psi(t_2) \vee \cdots \vee \psi(t_n)) \vee (\neg(\psi(t_1) \wedge \psi(t_2) \wedge \cdots \wedge \psi(t_n)) \wedge \psi(t))$. We establish that p may $\Sigma T \Box \theta; t$ iff $\xi(p) \vDash \psi(\Sigma T \Box \theta; t)$ in this particular case by induction over n, the size of T.

[1] As usual $\bigvee_{t \in T = \{t_1, \ldots, t_n\}} t$ is a shorthand for $t_1 \vee \cdots \vee t_n$.

For $n = 1$, that is, $T = \{t_1\}$, p may $t_1 \,\Box\, \theta; t$ iff p may t_1 or it is not the case that p may t_1 but p may t. This is equivalent to $\xi(p) \vDash \psi(t_1) \vee ((\xi(p) \nvDash \psi(t_1)) \wedge (\xi(p) \vDash \psi(t)))$, that is, $\xi(p) \vDash \psi(t_1) \vee ((\neg\psi(t_1)) \wedge \psi(t))$, as desired. We now assume that the property holds for $T = \{t_1, \ldots, t_n\}$ and consider a new $b \notin T$. We have p may $\Sigma(T \cup \{b\}) \,\Box\, \theta; t$ iff p may $\Sigma T \,\Box\, b \,\Box\, \theta; t$ iff p may ΣT or p may b, or it is not the case that p may ΣT or p may b, but p may t iff $\xi(p) \vDash (\psi(t_1) \vee \ldots \vee \psi(t_n)) \vee \xi(p) \vDash \psi(b) \vee (\xi(p) \nvDash (\psi(t_1) \vee \ldots \vee \psi(t_n)) \wedge \xi(p) \nvDash \psi(b) \wedge \xi(p) \vDash \psi(t))$ (by induction hypothesis for n and also for 1) iff $\xi(p) \vDash (\psi(t_1) \vee \ldots \vee \psi(t_n) \vee \psi(b)) \vee (\xi(p) \nvDash (\psi(t_1) \vee \ldots \vee \psi(t_n) \vee \psi(b)) \wedge \xi(p) \vDash \psi(t))$ iff $\xi(p) \vDash (\psi(t_1) \vee \ldots \vee \psi(t_n) \vee \psi(b)) \vee (\neg(\psi(t_1) \vee \ldots \vee \psi(t_n) \vee \psi(b)) \wedge \psi(t))$ iff $\xi(p) \vDash \psi(\Sigma(T \cup \{b\}) \,\Box\, \theta; t)$, as desired.

Both inductions (over the size of T and over the structure of tests) are complete. □

3.3 From CTL Formulae to Failure Trace Tests

We go now the other way around and show that CTL formulae can be converted into failure trace tests. Recall that $\mathcal{P}$ is the set of all processes, $\mathcal{T}$ the set of all tests, and $\mathcal{F}$ the set of all CTL formulae. By abuse of notation we write p may A for some $A \subseteq \mathcal{T}$ iff p may t for all $t \in A$.

Theorem 3 *There exists a function $\omega : \mathcal{F} \to 2^{\mathcal{T}}$ such that for any process p, $\xi(p) \vDash f$ iff p may $\omega(f)$.*

Proof. The proof is done by structural induction over CTL formulae. As before, the function ω will also be defined inductively in the process.

We put $\omega(\top) = \{\text{pass}\}$. Any Kripke structure satisfies $\top$ and any process passes pass, so it is immediate that $\xi(p) \vDash \top$ iff p may $\{\text{pass}\} = \omega(\top)$. Similarly it is immediate that $\xi(p) \vDash \bot$ iff p may $\{\text{stop}\}$ (namely, never!), so we put $\omega(\bot) = \{\text{stop}\}$. We then put $\omega(a) = \{a; \text{pass}\}$, noting that $\xi(p) \vDash a$ iff p may a; pass by the definition of ξ.

For exactly all the tests $t \in \omega(f)$ we add t' to $\omega(\neg f)$, where t' is generated out of t using the following algorithm: We first force all the successful states to become deadlock states (that is, we eliminate the outgoing action γ from t completely); whenever the test would have reached a successful state, it now reaches a deadlock state and fails. Then we introduce an action θ followed by an action γ (success) to all the states except the ones that were success states originally; this way, t' can succeed in any state other than the original success states.

We put $\omega(f_1 \wedge f_2) = \omega(f_1) \cup \omega(f_2)$. Indeed, $\xi(p) \vDash f_1 \wedge f_2$ iff $\xi(p) \vDash f_1$ and $\xi(p) \vDash f_2$ iff p may $\omega(f_1)$ and p may $\omega(f_2)$ (by induction hypothesis) iff p may $\omega(f_1) \cup \omega(f_2)$. Whenever $\xi(p) \vDash f_1 \wedge f_2$ the process p should pass all of the tests in $\omega(f_1)$ and $\omega(f_2)$ (and the other way around).

A direct form of $\omega(f_1 \vee f_2)$ is exceedingly complicated, so we retort instead to the De Morgan rules for logical operators and we put $\omega(f_1 \vee f_2) = \omega(\neg(\neg f_1 \wedge \neg f_2))$. Given the De Morgan rules (namely, that $\neg(f_1 \vee f_2)$ is equivalent to $\neg f_1 \wedge \neg f_2$) and the previous definitions of $\omega(\neg f)$ and $\omega(f_1 \wedge f_2)$, the correctness of the construction is immediate.

Now we put $\omega(\mathsf{EX}\ f) = \{\Sigma(a; t : a \in A) : t \in \omega(f)\}$. As shown in Figure 2, the test suite is generated by combining a choice of action from all the available actions and the tests from $\omega(f)$. p may $\omega(\mathsf{EX}\ f)$ iff p passes each of the test cases above, which is equivalent to p being able to perform an action (any action!) and then pass the tests in $\omega(f)$. By the inductive assumption that $\omega(f)$ is equivalent to f, the above is equivalent to $\xi(p) \vDash \mathsf{EX}\ f$ (any action then satisfy f).

We then have $\omega(\mathsf{AX}\ f) = \{a; t \,\Box\, \theta; \text{pass} : a \in A, t \in \omega(f)\}$. As shown in Figure 3, the test suite is generated by combining an action and a test from $\omega(f)$. When the action is not provided,

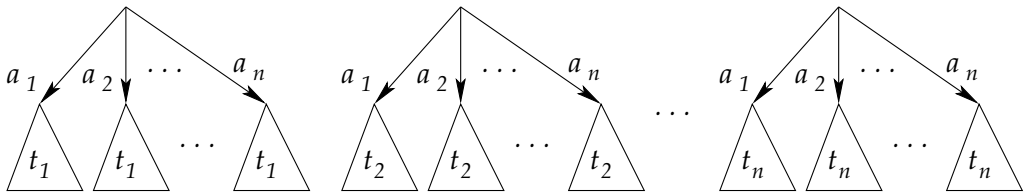

Figure 2: Test suite for the CTL formula EX f.

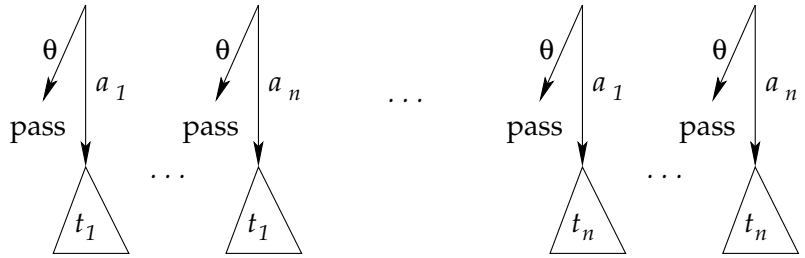

Figure 3: Test suite for the CTL formula AX f.

a deadlock detection transition will take place and lead to a pass state (so that particular test does no play any role). The test suite is generated by providing all the possible actions; the system under test however does not necessarily have to perform all the possible 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, we get a deadlock whenever the respective action is not encountered in the system; this leads to a pass state and then we continue to check the results of the other runs with the other test cases. p may $\omega(\mathsf{AX}\ f)$ iff p passes each of the test cases above, that is, whenever p can perform an action then after performing it (in the next state) it passes all the tests in $\omega(f)$ (which by inductive assumption is equivalent to the formula f).

We put $\omega(\mathsf{EF}\ f) = \{t' = t \,\Box\, \mathbf{i}; (\Sigma(a; t') : a \in A)) : t \in \omega(f)\}$. Another way of depicting $\omega(\mathsf{EF}\ f)$ is shown graphically in Figure 4. The test suite is generated by combining a choice of 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 until the last layer of the tree of the Kripke structure. The resulting test suite is nondeterministic. p satisfies the formula EF f iff p passes the tests in $\omega(f)$, or p can perform one action and at the next state it passes the tests in $\omega(f)$, or p can perform two

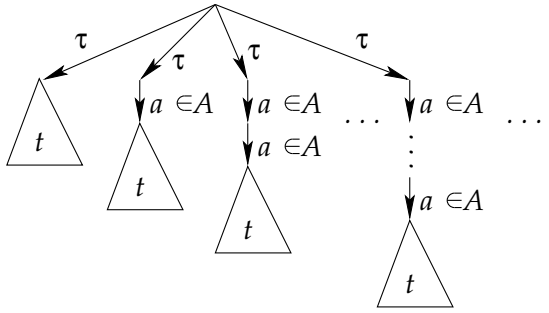

Figure 4: Test suite for the CTL formula EF f.

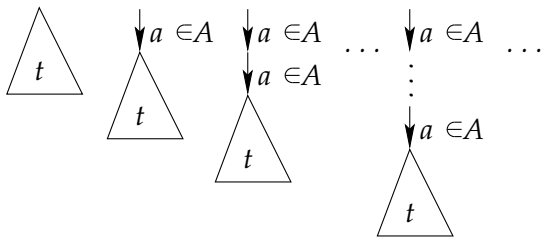

Figure 5: Test suite for CTL formula AG f.

actions and then it passes the tests in $\omega(f)$ and so on. Clearly this corresponds to the formula EF f given the induction hypothesis that $\omega(f)$ is equivalent to f.

Let $\omega(\mathsf{AF}\ f) = \{\omega(f) \,\square\, \omega(\mathsf{AX}\ f') : f' = f \vee \mathsf{AX}\ f'\}$. Here f' is a recursive definition. Unfolding this formula yields f or AX $(f \vee \mathsf{AX}\ f')$ or AX $(f \vee \mathsf{AX}\ (f \vee \mathsf{AX}\ f'))$, and so on. The process should satisfy any of the unfolded formulae 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 labelled transition system, the process need to pass the tests in $\omega(f)$ or it performs some action to the second layer states and then the process need to pass the tests in $\omega(f)$, and so on. These are clearly equivalent.

Similarly, we put $\omega(\mathsf{EG}\ f) = \{\omega(f) \cup \omega(\mathsf{EX}\ f') : f' = f \wedge \mathsf{EX}\ f'\}$. When we unfold f' we get f and EX $(f \wedge \mathsf{EX}\ f')$ and EX $(f \wedge \mathsf{EX}\ (f \wedge \mathsf{EX}\ f'))$ and so on. The process should satisfy all of the unfolded formulae in order for the 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 then some successive states in the second layer need to satisfy the formula f and so on. In terms of labelled transition system, the process need to pass the tests in $\omega(f)$ and then perform some action to the second layer states and then pass $\omega(f)$ and so on. Again, these are equivalent.

Once more similarly we put $\omega(\mathsf{AG}\ f) = \{\omega(f) \cup \omega(\mathsf{AX}\ f') : f' = f \wedge \mathsf{AX}\ f'\}$. Unfolding f' yields f and AX $(f \wedge \mathsf{AX}\ f')$ and AX $(f \wedge \mathsf{AX}\ (f \wedge \mathsf{AX}\ f'))$ and so on. Figure 5 illustrates that the test suite is generated by combining a choice of action and the tests in $\omega(f)$, then a choice of two actions followed by the tests in $\omega(f)$, and so on until the last layer of the tree of the Kripke structure. p satisfies the formula AG f iff p passes the tests in $\omega(f)$, or p can perform an action and then at the next states pass the tests in $\omega(f)$, or or p can perform two actions then it pass the tests from $\omega(f)$, and so on.

Finally we put $\omega(\mathsf{E}\ f_1\ \mathsf{U}\ f_2) = \{(\omega(f_1) \cup \omega(\mathsf{EX}\ f')) \,\square\, \mathbf{i}; (\omega(f_2) \cup \omega(\mathsf{EX}\ f'')) : f' = f_1 \wedge \mathsf{EX}\ f', f'' = f_2 \wedge \mathsf{EX}\ f''\}$. This is similar to the EG f case: f' and f'' are recursive definitions. Unfolding the formula yields f_1 and EX $(f_1 \wedge \mathsf{EX}\ f')$ and EX $(f_1 \wedge \mathsf{EX}\ (f_1 \wedge \mathsf{EX}\ f'))$ and so on, until we change via an internal action to f_2 and EX $(f_2 \wedge \mathsf{EX}\ f'')$ and EX $(f_2 \wedge \mathsf{EX}\ (f_2 \wedge \mathsf{EX}\ f''))$, etc. In a Kripke structure, the states in the first layer need to satisfy the formula f_1 and then 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 whole path. In terms of labelled transition system, the process need to pass the tests in $\omega(f_1)$ and then perform some action to the second layer of states where some state needs to pass the tests from $\omega(f_1)$ again, and so on until some point where some state passes $\omega(f_2)$; from then on, some state from every layer needs to pass $\omega(f_2)$.

Thus we complete the proof and the conversion between CTL formulae and sequential tests. Indeed, note that all the remaining CTL constructs can be rewritten using only the constructs discussed above. □

4 Conclusions and Open Problems

We have shown the equivalence between CTL formulae and failure trace tests. Our results can be summarized as follows: We defined the equivalence between a process or a labelled transition system and a Kripke structure in Definition 3, and then (Theorem 1) we built a function that converts any labelled transition system to an equivalent Kripke structure. As a result, we can define the semantics of CTL formulae with respect to a process rather than a Kripke structure. We then define two functions ψ and ω that link the failure trace tests to corresponding, equivalent CTL formulae (Theorems 2 and 3).

It is worth noting that the function ξ creates a Kripke structure that may have multiple initial states, and so we were forced to use a weaker satisfaction operator over sets of states (rather than single states) as per Definition 2. We note however that such an issue manifests itself only when the LTS being converted exhibits *initial nondeterminism*, meaning that it can perform nondeterministically more than one initial action. No other form of nondeterminism requires satisfaction over sets of states. Therefore a sufficient condition to eliminate satisfaction over sets of states is that the initial state of the LTS being converted is stable and all its actions lead to one next state; indeed, in such a case the initial state in never split in the equivalent Kripke structure. Initial nondeterminism can thus be eliminated by creating a new start state that performs a "start" action and then gives control to the original initial state. Our results are thus without loss of generality; indeed, it is easy to see that in the absence of initial nondeterminism the proofs of Theorems 1, 2, and 3 revert to the normal satisfaction operator (over Kripke states).

This being said, the existence of a conversion between LTS and Kripke structures that preserves the nice properties (used subsequently) of the conversion described in Definition 3 and also copes directly with initial nondeterminism is an interesting open problem.

In all, we have developed a combined method of system verification. We provided the means to convert failure trace tests into CTL temporal logic formulae and CTL temporal logic formulae back into failure trace tests. This opens the way toward a combined, logical and algebraic approach to formal verification. Such a combined approach has a number of advantages over traditional approaches. Model-based testing is incomplete but has the advantage of incremental or compositional application. By contrast, model checking is complete but must be applied all at once, generating the state explosion problem [5]. CTL formulae represent loose specifications, as we only specify the properties of interest; by contrast, 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 scrutiny. In our work, we convert CTL temporal logic formulae 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 globally. 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 using temporal logic formulae. The conversion therefore is very useful. Such

a conversion can also allow the use of the fastest, most suitable, or even most convenient method of verification, irrespective to the form taken by the specification.

The results of this paper are important first steps towards the ambitious goal of a unified (logic and algebraic) approach to conformance testing. We believe in particular that this paper opens several direction of future research. The main challenge in the method we introduced is dealing with the infinite-state test cases. Indeed, the test cases produced from CTL temporal logic formulae can have an infinite set of states. This is fine theoretically, but from a practical perspective it is worthy of future work to eliminate infinite states or to obtain usable algorithms than simulate runs of infinite-state test suites with the system under test and obtain useful results in finite time. The tests developed here can be combined with partial application so that another interesting research direction is to find partial application algorithms that yield total correctness at the limit and have some correctness insurance milestones along the way.

On the conversion the other way around (from tests to CTL formulae) we note that the obtained formulae $\psi(t)$ may not be in their simplest (or more concise) form possible in several respects. The formulae may even have an infinite length (this happens for infinite tests), since they don't really exploit the operators G, F, U, and R. We have strong reasons to believe that bringing it to more manageable proportions is possible, and this is one subject of our research.

Another obvious open question is whether there is an equivalence between a category of tests and the full temporal logic CTL*.

References

[1] E. BRINKSMA, G. SCOLLO, AND C. STEENBERGEN, *LOTOS specifications, their implementations and their tests*, in Proc. IFIP 6.1, 1987, pp. 349–360.

[2] S. D. BRUDA, *Preorder relations*, in Model-Based Testing of Reactive Systems: Advanced Lectures, M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, eds., vol. 3472 of Lecture Notes in Computer Science, Springer, 2005, pp. 117–149.

[3] E. M. CLARKE AND E. A. EMERSON, *Design and synthesis of synchronization skeletons using branching-time temporal logic*, in Works in Logic of Programs, 1982, pp. 52–71.

[4] 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, 8 (1986), pp. 244–263.

[5] E. M. CLARKE, O. GRUMBERG, AND D. A. PELED, *Model Checking*, MIT Press, 1999.

[6] R. DE NICOLA AND M. HENNESSY, *Testing equivalences for processes*, Theoretical Computer Science, 34 (1984), pp. 83–133.

[7] R. DE NICOLA AND F. VAANDRAGER, *Three logics for branching bisimulation*, Journal of the ACM, 42 (1995), pp. 438–487.

[8] R. LANGERAK, *A testing theory for LOTOS using deadlock detection*, in Proceedings of the IFIP WG6.1 Ninth International Symposium on Protocol Specification, Testing and Verification IX, 1989, pp. 87–98.

[9] K. PAWLIKOWSKI, *Steady-state simulation of queueing processes: survey of problems and solutions*, ACM Computing Surveys, 22 (1990), pp. 123–170.

[10] A. PNUELI, *A temporal logic of concurrent programs*, Theoretical Computer Science, 13 (1981), pp. 45–60.

[11] H. SAÏDI, *The invariant checker: Automated deductive verification of reactive systems*, in Proceedings of Computer Aided Verification (CAV 97), vol. 1254 of Lecture Notes In Computer Science, Springer, 1997, pp. 436–439.

[12] S. SCHNEIDER, *Concurrent and Real-time Systems: The CSP Approach*, John Wiley & Sons, 2000.

[13] T. J. SCHRIBER, J. BANKS, A. F. SEILA, I. STÅHL, A. M. LAW, AND R. G. BORN, *Simulation textbooks - old and new, panel*, in Winter Simulation Conference, 2003, pp. 1952–1963.

[14] J. TRETMANS, *Conformance testing with labelled transition systems: implementation relations and test generation*, Computer Networks and ISDN Systems, 29 (1996), pp. 49–79.