Computation Tree Logic Is Equivalent to Failure Trace Testing

by

A. F. M. Nokib Uddin

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

Bishop's University
Sherbrooke, Quebec, Canada
July 2015

Abstract

The two major systems of formal verification are model checking and algebraic techniques such as model-based testing. Model checking is based on some form of temporal logic such as linear temporal logic (LTL) or computation tree logic (CTL). CTL in particular is capable of expressing most interesting properties of processes such as liveness and safety. Algebraic techniques are based on some operational semantics of processes (such as traces and failures) and its associated preorders. The most fine-grained practical preorder is based on failure traces. The particular algebraic technique for formal verification based on failure traces is failure trace testing.

It was shown earlier [8] that CTL and failure trace testing are equivalent; that is, for any failure trace test there exists a CTL formula equivalent to it, and the other way around. Both conversions are constructive and algorithmic. The proof of the conversion from failure trace tests to CTL formulae and implicitly the associated algorithm is however incorrect [6].

We now provide a correct proof for the existence of a conversion from failure trace tests to CTL formulae. We also offer intuitive support for our proof by providing worked examples related to the examples used earlier [9] to support the conversion the other way around, thus going full circle not only with the conversion but also with our examples. The revised proof continues to be constructive and so the conversion continues to be algorithmic.

Our corrected proof completes an algorithmic solution that allows the free mix of logic

and algebraic specifications for any system, thus offering increased flexibility and convenience in system specification for formal verification.

Acknowledgments

In writing this thesis I have been greatly assisted in terms of encouragement, time, and guidance by a number of individuals. I would like to thank my supervisor Stefan D. Bruda for his timely assistance during our meetings, for his constant evaluation of the various drafts of my work, and for lots of helpful ideas and technical support. Without him I would not be in the position of writing this thesis. Then I would like to acknowledge Professor Nelly Khouzam for her friendship and support throughout my MSc studies.

An equally important contribution has come from my mother Satal Moni, my father Md. Abdus Sattar, my close friend Kaniz Afrin, my sisters, and my friends here at Bishop's University.

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

Contents

List of Figures

Chapter 1

Introduction

Computing systems are already ubiquitous in our everyday life, from entertainment systems at home, to telephone networks and the Internet, and even to health care, transportation, and energy infrastructure. Ensuring the correct behaviour of software and hardware has been one of the goals of Computer Science since the dawn of computing. Since then computer use has skyrocketed and so has the need for assessing correctness.

Historically the oldest verification method, which is still widely used today, is empirical testing [23, 28]. This is a non-formal method which provides input to a system, observes the output, and verifies that the output is the one expected given the input. Such testing cannot check all the possible input combinations and so it can disprove correctness but can never prove it. Deductive verification [17, 19, 26] is chronologically the next verification method developed. It consists of providing proofs of program correctness manually, based on a set of axioms and inference rules. Program proofs provide authoritative evidence of correctness but are time consuming and require highly qualified experts.

Various techniques have been developed to automatically perform program verification with the same effect as deductive reasoning but in an automated manner. These efforts are grouped together in the general field of formal methods. The general technique is to verify a system automatically against some formal specification. Model-based testing and model checking are the two approaches to formal methods that became mainstream.

Their roots can be traced to simulation and deductive reasoning, respectively. These formal methods however are sound, complete, and to a large extent automatic. They have proven themselves through the years and are currently in wide use throughout the computing industry.

In model-based testing [4, 15, 30] the specification of a system is given algebraically, with the underlying semantics given in an operational manner as a labeled transition system (LTS for short), or sometimes as a finite automaton (a particular, finite kind of LTS). Such a specification is usually an abstract representation of the system's desired behaviour. The system under test is modeled using the same formalism (either finite or infinite LTS). The specification is then used to derive systematically and formally tests, which are then applied to the system under test. The way the tests are generated ensures soundness and completeness. In this thesis we focus on arguably the most powerful method of model-based testing, namely failure trace testing [21]. Failure trace testing also introduces a smaller set (of sequential tests) that is sufficient to assess the failure trace relation.

By contrast, in model checking [11, 12, 25] the system specification is given in some form of temporal logic. The specification is thus a (logical) description of the desired properties of the system. The system under test is modeled as Kripke structures, another formalism similar to transition systems. The model checking algorithm then determines whether the initial states of the system under test satisfy the specification formulae, in which case the system is deemed correct. There are numerous temporal logic variants used in model checking, including CTL*, CTL and LTL. In this thesis we focus on CTL.

There are advantages as well as disadvantages to each of these formal methods techniques. Model checking is a complete verification technique, which has been widely studied and also widely used in practice. The main disadvantage of this technique is that it is not compositional. It is also the case that model checking is based on the system under test being modeled using a finite state formalism, and so does not scale very well with the size of the system under test. By contrast, model-based testing is compositional by definition

(given its algebraic nature), and so has better scalability. In practice however it is not necessarily complete given that some of the generated tests could take infinite time to run and so their success or failure cannot be readily ascertained. The logical nature of specification for model checking allows us to only specify the properties of interest, in contrast with the labeled transition systems or finite automata used in model-based testing which more or less require that the whole system be specified.

Some properties of a system may be naturally specified using temporal logic, while others may be specified using finite automata or labeled 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. However, such a mixed specification cannot be verified. Parts of it can be model checked and some other parts can be verified using model-based testing. However, no global algorithm for the verification of the whole system exists. Before even thinking of verifying such a specification we need to convert one specification to the form of the other.

Precisely such a conversion was investigated relatively recently and published as a technical report [8]. An equivalence between labeled transition systems (the semantic model used in model-based testing) and Kripke structures (the semantic model used in model checking) was first established, and then it was shown that for each failure trace test there exists an equivalent CTL formula and the other way around. The conversion from tests to CTL formulae was further published [9]. It was discovered however that the conversion the other way around (from CTL formulae to tests) is incorrect as presented earlier [8]. The purpose of this thesis is to fix this conversion.

We prove that though the original construction of the conversion function [8] was incorrect, the existence of such a function does hold. Specifically, we show that for each CTL formula there exists an equivalent failure trace test, finally showing that the two (algebraic and logic) formalisms are equivalent. Like all the earlier proofs [8, 9] our proof is constructive, so that implementing back and forth automated conversions is an immediate

consequence of our result combined with the earlier ones.

We believe that we are thus 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 the all-or-nothing model-checking lacks).

The thesis continues as follows: We introduce the basic concepts used in our work including model checking, temporal logic, model-based testing, and failure trace testing in Chapter 2. Previous work is reviewed in Chapter 3. In particular the work on labeled transition system and Kripke structure equivalence and on converting failure trace tests to CTL formulae [8, 9] is presented in detail in Sections 3.1 and 3.2, respectively. Chapter 4 then presents our conversion from CTL formulae to failure trace tests. We discuss the significance and consequences of our work (as a whole together with the conversion the other way [9]) in Chapter 5. For the remainder of this thesis results proved elsewhere are introduced as Propositions, while original results are stated as Theorems, Lemmata, and Corollaries.

Chapter 2

Preliminaries

This chapter is dedicated to introducing the necessary background information on model checking, temporal logic, and failure trace testing. For technical reasons we also introduce here the language TLOTOS, a process algebra used for describing algebraic specifications, tests, and systems under test. The reason for using this particular language is that earlier work on failure trace testing uses this language as well.

Our preliminaries are necessarily the same to the preliminaries used in the work we are fixing [8]. For this reason the content of this section is largely identical to the earlier work, though we have streamlined and somehow shortened the presentation.

Given a set of symbols A we use as usual A^* to denote exactly all the strings of symbols from A. The empty string, and only the empty string is denoted by ε. We use ω to refer to $|\mathbb{N}|$, the cardinality of the set of natural numbers $\mathbb{N}$. The power set of a set A is denoted as usual by 2^A.

2.1 Temporal Logic and Model Checking

A specification suitable for model checking is described by a temporal logic formula. The system under test is given as 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 of the

respective Kripke structure satisfy the logic formula.

Formally, a *Kripke structure* [12] K over a set AP of atomic propositions is a tuple $(S, S_0, \rightarrow, L)$, where S is a set of states, $S_0 \subseteq S$ is the set of initial states, $\rightarrow \subseteq S \times S$ is the transition relation, and $L : S \rightarrow 2^{\mathsf{AP}}$ is a function that assigns to each states exactly all the atomic propositions that are true in that state. As usual we write $s \rightarrow t$ instead of $(s, t) \in \rightarrow$. It is usually assumed [12] that $\rightarrow$ is total, meaning that for every state $s \in S$ there exists a state $t \in S$ such that $s \rightarrow t$. Such a requirement can however be easily established by creating a "sink" state that has no atomic proposition assigned to it, is the target of all the transitions from states with no other outgoing transitions, and has one outgoing "self-loop" transition back to itself.

A *path* π in a Kripke structure is a 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. Any state may be the start of multiple paths. It follows that all the paths starting from a given state s_0 can be represented together as a computation tree with nodes labeled with states. Such a tree is rooted at s_0 and (s, t) is an edge in the tree if and only if $s \rightarrow t$. Some temporal logics reason about computation paths individually, while some other temporal logics reason about whole computation trees.

There are several temporal logics currently in use. We will focus in this thesis on the CTL* family [12, 16] and more precisely on the CTL variant. CTL* is a general temporal logic which is usually restricted for practical considerations. One such a restriction is the linear-time temporal logic or LTL [12, 24], which is an example of temporal logic that represents properties of individual paths. Another restriction is the computation tree logic or CTL [10, 12], which represents properties of computation trees.

In CTL* the properties of individual paths are represented using five temporal operators: X (for a property that has to be true in the next state of the path), F (for a property that has to eventually become true along the path), G (for a property that has to hold in every state along the path), U (for a property that has to hold continuously along a path until another property becomes true and remains true for the rest of the path), and R (for

a property that has to hold along a path until another property becomes true and releases the first property from its obligation). These path properties are then put together so that they become state properties using the quantifiers A (for a property that has to hold on all the outgoing paths) and E (for a property that needs to hold on at least one of the outgoing paths).

CTL is a subset of CTL*, with the additional restriction that the temporal constructs X, F, G, U, and R must be immediately preceded by one of the path quantifiers A or E. More precisely, the syntax of CTL formulae is defined as follows:

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

where $a \in \mathsf{AP}$, and f, f_1, f_2 are all state formulae.

CTL formulae are interpreted over states in Kripke structures. Specifically, the CTL semantics is given by the operator $\models$ such that $K, s \models f$ means that the formula f is true in the state s of the Kripke structure K. All the CTL formulae are state formulae, but their semantics is defined using the intermediate concept of path formulae. In this context the notation $K, \pi \models f$ means that the formula f is true along the path π in the Kripke structure K. The operator $\models$ is defined inductively as follows:

1. $K, s \models \top$ is true and $K, s \models \bot$ is false for any state s in any Kripke structure K.
2. $K, s \models a, a \in \mathsf{AP}$ if and only if $a \in L(s)$.
3. $K, s \models \neg f$ if and only if $\neg(K, s \models f)$ for any state formula f.
4. $K, s \models f \wedge g$ if and only if $K, s \models f$ and $K, s \models g$ for any state formulae f and g.
5. $K, s \models f \vee g$ if and only if $K, s \models f$ or $K, s \models g$ for any state formulae f and g.

6. $K, s \vDash \mathsf{E}\ f$ for some path formula f if and only if there exists a path $\pi = s \to s_1 \to s_2 \to \cdots \to s_i, i \in \mathbb{N} \cup \{\omega\}$ such that $K, \pi \vDash f$.

7. $K, s \vDash \mathsf{A}\ f$ for some path formula f if and only if $K, \pi \vDash f$ for all paths $\pi = s \to s_1 \to s_2 \to \cdots \to s_i, i \in \mathbb{N} \cup \{\omega\}$.

We use π^i to denote the i-th state of a path π, with the first state being π^0. The operator $\vDash$ for path formulae is then defined as follows:

1. $K, \pi \vDash \mathsf{X}\ f$ if and only if $k, \pi^1 \vDash f$ for any state formula f.

2. $K, \pi \vDash f\ \mathsf{U}\ g$ for any state formulae f and g if and only if there exists $j \geq 0$ such that $K, \pi^k \vDash g$ for all $k \geq j$, $K, \pi^i \vDash f$ for all $i < j$.

3. $K, \pi \vDash f\ \mathsf{R}\ g$ for any state formulae f and g if and only if for all $j \geq 0$, if $K, \pi^i \nvDash f$ for every $i < j$ then $K, \pi^j \vDash g$.

2.2 Labeled Transition Systems and Stable Failures

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 labeled transition system (LTS), where the labels (or actions) are associated with the transitions instead.

An LTS [20] 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, and A is a countable set of actions. The actions in A are called visible (or observable), by contrast with the special, unobservable action $\tau \notin A$ (also called internal action). The relation $\to \subseteq S \times (A \cup \{\tau\}) \times S$ is the transition relation; we use $p \xrightarrow{a} q$ instead of $(p, a, q) \in \to$. A transition $p \xrightarrow{a} q$ means that state p becomes state q after performing the (visible or internal) action a.

The notation $p \xrightarrow{a}$ stands for $\exists p' : p \xrightarrow{a} p'$. The sets of states and transitions can also be considered global, in which case 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". Given that $\rightarrow$ is a relation rather than a function, and also given the existence of the internal action, an LTS defines a nondeterministic process.

A *path* (or *run*) π starting from state p' is a sequence $p' = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots p_{k-1} \xrightarrow{a_k} p_k$ with $k \in \mathbb{N} \cup \{\omega\}$ such that $p_{i-1} \xrightarrow{a_i} p_i$ for all $0 < i \leq k$. We use $|\pi|$ to refer to k, the length of π. If $|\pi| \in \mathbb{N}$, then we say that π is finite. The trace of π is the sequence $\text{trace}(\pi) = (a_i)_{0<i\leq|\pi|, a_i \neq \tau} \in A^*$ of all the visible actions that occur in the run listed in their order of occurrence and including duplicates. Note in particular that internal actions do not appear in traces. The set of finite traces of a process p is defined as $\text{Fin}(p) = \{tr \in \text{traces}(p) : |tr| \in \mathbb{N}\}$. If we are not interested in the intermediate states of a run then we use the notation $p \stackrel{w}{\Longrightarrow} q$ to state that there exists a run π starting from state p and ending at state q such that $\text{trace}(\pi) = w$. We also use $p \stackrel{w}{\Longrightarrow}$ instead of $\exists p' : p \stackrel{w}{\Longrightarrow} p'$.

A process p that has no outgoing internal action cannot make any progress unless it performs a visible action. We say that such a process is *stable* [27]. We write $p \downarrow$ whenever we want to say that process p is stable. Formally, $p \downarrow = \neg(\exists p' \neq p : p \stackrel{\varepsilon}{\Longrightarrow} p')$. A stable process p responds predictably to any set of actions $X \subseteq A$, in the sense that its response depends exclusively on its outgoing transitions. Whenever there is no action $a \in X$ such that $p \xrightarrow{a}$ we say that p *refuses* the set X. Only stable processes are able to refuse actions; unstable processes refuse actions "by proxy": they refuse a set X whenever they can internally become a stable process that refuses X. Formally, p refuses X (written p ref X) if and only if $\forall a \in X : \neg(\exists p' : (p \stackrel{\varepsilon}{\Longrightarrow} p') \wedge p' \downarrow \wedge p' \xrightarrow{a})$.

To describe the behaviour of a process in terms of refusals we need to record each refusal together with the trace that causes that refusal. An observation of a refusal plus the trace that causes it is called a *stable failure* [27]. Formally, (w, X) is a stable failure of process p if and only if $\exists p^w : (p \stackrel{w}{\Longrightarrow} p^w) \wedge p^w \downarrow \wedge (p^w \text{ ref } X)$. The set of stable failures of p is then $\mathcal{SF}(p) = \{(w, X) : \exists p^w : (p \stackrel{w}{\Longrightarrow} p^w) \wedge p^w \downarrow \wedge (p^w \text{ ref } X)\}$.

Several preorder relations (that is, binary relations that are reflexive and transitive but

not necessarily symmetric or antisymmetric) can be defined over processes based on their observable behaviour (including traces, refusals, stable failures, etc.) [5]. Such preorders can then be used in practice as implementation relations, which in turn create a process-oriented specification technique. The *stable failure preorder* is defined based on stable failures and is one of the finest such preorders (but not the absolute finest) [5].

Let p and q be two processes. The stable failure preorder $\sqsubseteq_{\mathrm{SF}}$ is defined as $p \sqsubseteq_{\mathrm{SF}} q$ if and only if $\mathrm{Fin}(p) \subseteq \mathrm{Fin}(q)$ and $\mathcal{SF}(p) \subseteq \mathcal{SF}(q)$. Given the preorder $\sqsubseteq_{\mathrm{SF}}$ one can naturally define the stable failure equivalence $\simeq_{\mathcal{SF}}$: $p \simeq_{\mathcal{SF}} q$ if and only if $p \sqsubseteq_{\mathrm{SF}} q$ and $q \sqsubseteq_{\mathrm{SF}} p$.

2.3 Failure Trace Testing

In model-based testing [4] a test runs in parallel with the system under test and synchronizes 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. The outcome of such a run is either success ($\top$) or failure ($\bot$). The precise definition of synchronization, success, and failure depends on the particular type of tests being considered. We will present below such definitions for the particular framework of failure trace testing.

Given the nondeterministic nature of LTS there may be multiple runs for a given process p and a given test t and so a set of outcomes is necessary to give the results of all the possible runs. We denote by $\mathrm{Obs}(p, t)$ the set of exactly all the possible outcomes of all the runs of p and t. Given the existence of such a set of outcomes, two definitions of a process passing a test are possible. More precisely, a process p *may* pass a test t whenever some run is successful (formally, p may t if and only if $\top \in \mathrm{Obs}(p, t)$), while p *must* pass t whenever all runs are successful (formally, p must t if and only if $\{\top\} = \mathrm{Obs}(p, t)$).

In what follows we use the notation $\mathrm{init}(p) = \{a \in A : p \stackrel{a}{\Longrightarrow}\}$. A failure trace f [21] is a string of the form $f = A_0 a_1 A_1 a_2 A_2 \ldots a_n A_n$, $n \geq 0$, with $a_i \in A^*$ (sequences of actions) and $A_i \subseteq A$ (sets of refusals). Let p be a process such that $p \stackrel{\varepsilon}{\Longrightarrow} p_0 \stackrel{a_1}{\Longrightarrow} p_1 \stackrel{a_2}{\Longrightarrow} \cdots \stackrel{a_n}{\Longrightarrow} p_n$;

$f = A_0 a_1 A_1 a_2 A_2 \ldots a_n A_n$ is then a failure trace of p whenever the following two conditions hold:

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

In other words, we obtain a failure trace of p by taking a trace of p and inserting refusal sets after stable states.

Systems and tests can be concisely described using the testing language TLOTOS [3, 21], which will also be used in this thesis. 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 or processes. The syntax of TLOTOS is then defined as follows:

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

The semantics of TLOTOS is then the following:

1. inaction (stop): no rules.
2. action prefix: $a; t_1 \xrightarrow{a} t_1$ and $\mathbf{i}; t_1 \xrightarrow{\tau} t_1$
3. deadlock detection: $\theta; t_1 \xrightarrow{\theta} t_1$.
4. successful termination: pass $\xrightarrow{\gamma}$ stop.
5. choice: with $g \in A \cup \{\gamma, \theta, \tau\}$,

$$\frac{t_1 \xrightarrow{g} t_1'}{\begin{array}{c} t_1 \,\Box\, t_2 \xrightarrow{g} t_1' \\ t_2 \,\Box\, t_1 \xrightarrow{g} t_1' \end{array}}$$

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

$$\frac{t_1 \xrightarrow{g} t_1'}{\Sigma(\{t_1\} \cup t) \xrightarrow{g} t_1'}$$

Failure trace tests are defined in TLOTOS using the special actions γ which signals the successful completion of a test, and θ which is the deadlock detection label (the precise behaviour will be given later). Processes (or LTS) can also be described as TLOTOS processes, but such a description does not contain γ or θ. 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. One can then distinguish the possibility and the inevitability of success for a test as mentioned earlier: p may t if and only if $\top \in \text{Obs}(p, t)$, and p must t if and only if $\{\top\} = \text{Obs}(p, t)$.

The set $\mathcal{ST}$ of sequential tests is defined as follows [21]: 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 [21]. For a sequential test t the failure trace ftr(t) is defined inductively as follows: ftr(pass) $= \emptyset$, ftr($a; t'$) $= a$ ftr(t'), and ftr($\Sigma\{a; \text{stop} : a \in A'\} \,\square\, \theta; t'$) $= A'$ ftr(t'). Conversely, let f be a failure trace. Then we

can inductively define the sequential test $\mathrm{st}(f)$ as follows: $\mathrm{st}(\emptyset) = \mathrm{pass}$, $\mathrm{st}(af) = a\ \mathrm{st}(f)$, and $\mathrm{st}(Af) = \Sigma\{a; \mathrm{stop} : a \in A\} \,\Box\, \theta; \mathrm{st}(f)$. For all failure traces f we have that $\mathrm{ftr}(\mathrm{st}(f)) = f$, and for all tests t we have $\mathrm{st}(\mathrm{ftr}(t)) = t$. We then define the failure trace preorder $\sqsubseteq_{\mathrm{FT}}$ as follows: $p \sqsubseteq_{\mathrm{FT}} q$ if and only if $\mathrm{ftr}(p) \subseteq \mathrm{ftr}(q)$.

The above bijection effectively shows that the failure trace preorder (which is based on the behaviour of processes) can be readily converted into a testing-based preorder (based on the outcomes of tests applied to processes). Indeed there exists a successful run of p in parallel with the test t, if and only if f is a failure trace of both p and t. Furthermore, these two preorders are equivalent to the stable failure preorder introduced earlier:

Proposition 2.1 [21] *Let p be a process, t a sequential test, and f a failure trace. Then p* may *t if and only if $f \in \mathrm{ftr}(p)$, where $f = \mathrm{ftr}(t)$.*

Let p_1 and p_2 be processes. Then $p_1 \sqsubseteq_{\mathrm{SF}} p_2$ if and only if $p_1 \sqsubseteq_{\mathrm{FT}} p_2$ if and only if p_1 may *$t \implies p_2$* may *t for all failure trace tests t if and only if $\forall t' \in \mathcal{ST} : p_1$* 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 if and only if $\exists t' \in T(t) : p$* may *t'.*

We note in passing that unlike other preorders, $\sqsubseteq_{\mathrm{SF}}$ (or equivalently $\sqsubseteq_{\mathrm{FT}}$) can be in fact characterized in terms of may testing only; the must operator needs not be considered any further.

Chapter 3

Previous Work

The investigation into connecting logical and algebraic frameworks of formal specification and verification has not been pursued in too much depth. To our knowledge the only substantial investigation on the matter is based on linear-time temporal logic (LTL) and its relation with Büchi automata [29]. Such an investigation started with timed Büchi automata [1] approaches to LTL model checking [12, 15, 18, 31, 32].

An explicit equivalence between LTL and the may and must testing framework of De Nicola and Hennessy [15] was developed as a unified semantic theory for heterogeneous system specifications featuring mixtures of labeled transition systems and LTL formulae [13]. This theory uses Büchi automata [29] rather than LTS as underlying semantic formalism. The Büchi must-preorder for a certain class of Büchi process was first established by means of trace inclusion. Then LTL formulae were converted into Büchi processes whose languages contain the traces that satisfy the formula.

The relation between may and must testing and temporal logic mentioned above [13] was also extended to the timed (or real-time) domain [7, 14]. Two refinement timed preorders similar to may and must testing were introduced, together with behavioural and language-based characterizations for these relations (to show that the new preorders are extensions of the traditional preorders). An algorithm for automated test generation out of formulae written in a timed variant of LTL called Timed Propositional Temporal Logic

(TPTL) [2] was then introduced.

To our knowledge a single effort that considers the equivalence of CTL and algebraic specification exists [8, 9]. This work analyzes the equivalence between labeled transition systems and Kripke structures, and then establishes the conversion of failure trace tests into equivalent CTL formulae. An attempt at establishing the conversion the other way around also exists [8] but is incorrect [6]. Our work fixes this incorrect conversion. The remainder of this chapter is therefore dedicated to an in-depth presentation of the equivalence between LTS and Kripke structures, followed by the same treatment for the conversion of failure trace tests into CTL formulae [9].

Our presentation below follows faithfully the original technical report [8], with no change in the content but possible slight changes in the language being used. We include the original examples but for brevity we exclude the proofs. The only exception is the actual construction of the conversion function, which is shown originally in a proof; for completeness we therefore include a sketch of that proof in our manuscript which outlines the construction but omits the argument of correctness for that construction (see Proposition 3.2 in Section 3.2).

3.1 An Equivalence between LTS and Kripke Structures

A CTL satisfaction operator over LTS was defined in the same manner as for Kripke structures [8, 9]. In essence, the elementary propositions that hold in an LTS state are the actions that state can perform. Satisfaction for the rest of the CTL formulae is then defined inductively as usual.

Definition 3.1 SATISFACTION FOR PROCESSES [8, 9]: *A process p satisfies $a \in A$, written by abuse of notation $p \vDash a$, if and only if $p \xrightarrow{a}$. 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. $p \vDash \neg f$ *if and only if* $\neg(p \vDash f)$.

3. $p \vDash f \wedge g$ *if and only if* $p \vDash f$ *and* $p \vDash g$.

4. $p \vDash f \vee g$ *if and only if* $p \vDash f$ *or* $p \vDash g$.

5. $p \vDash \mathsf{E}\, f$ *for some path formula* f *if and only if there exists 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 *if and only if* $\pi \vDash f$ *for all paths* $\pi = p \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$.

As usual π^i *denotes 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. $\pi \vDash \mathsf{X}\, f$ *if and only if* $\pi^1 \vDash f$.

2. $\pi \vDash f\ \mathsf{U}\ g$ *if and only if 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. $\pi \vDash f\ \mathsf{R}\ g$ *if and only if for all* $j \geq 0$*, if* $\pi^i \nvDash f$ *for every* $i < j$ *then* $\pi^j \vDash g$.

A weaker satisfaction operator for CTL was also introduced. This operator is defined over sets of states rather than single states.

Definition 3.2 SATISFACTION OVER SETS OF STATES [8, 9]: *Consider a Kripke structure* $K = (S, S_0, R, L)$ *over* AP*. For some set* $Q \subseteq S$ *and some CTL state formula* f*,* $K, Q \vDash f$ *is defined 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. $K, Q \vDash a$ *if and only if* $a \in L(s)$ *for some* $s \in Q$*,* $a \in \mathsf{AP}$.

3. $K, Q \vDash \neg f$ *if and only if* $\neg(K, Q \vDash f)$.

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

5. $K, Q \vDash f \vee g$ *if and only if* $K, Q \vDash f$ *or* $K, Q \vDash g$.

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

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

Then the following equivalence between Kripke structures and LTS was introduced. One nice property of this equivalence is that a Kripke structure equivalent to a given LTS can be constructed algorithmically.

Definition 3.3 EQUIVALENCE BETWEEN KRIPKE STRUCTURES AND LTS [8, 9]: *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* $p \simeq K, Q$*), if and only if for any CTL* formula* f $K, Q \vDash f$ *if and only if* $p \vDash f$.

Proposition 3.1 [8, 9] *There exists an algorithmic function* $\mathbb{K}$ *which converts a labeled transition system* p *into a Kripke structure* K *and a set of states* Q *such that* $p \simeq (K, Q)$.

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

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

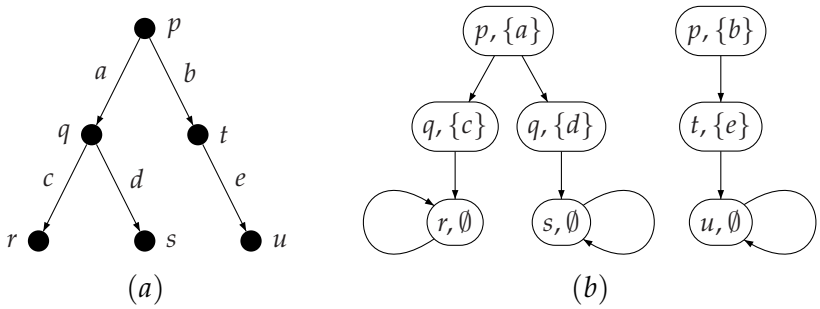
2. $Q = \{\langle s_0, x\rangle \in S'\}$.

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* $n \in N$, $s \overset{n}{\Longrightarrow} t$,

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

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



Each state of the Kripke structure (b) is labeled with the LTS state it came from, and the set of propositions that hold in that Kripke state.

Figure 3.1: A conversion of an LTS (a) to an equivalent Kripke structure (b) [8, 9].

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

A notable property of the conversion function $\mathbb{K}$ is that a single LTS state can result in multiple states in the resulting Kripke structure. This happens whenever an LTS state evolves differently after performing different actions. This is the case for the states p and q of the LTS shown in Figure 3.1(a), whose equivalent Kripke structure is shown in Figure 3.1(b). Such a "split" is also the reason for needing the weaker satisfaction operator (Definition 3.2).

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.

Proposition 3.2 [8, 9] *There exists a function* $\mathbb{F} : \mathcal{T} \to \mathcal{F}$ *such that* p may t *if and only if* $\mathbb{K}(p) \vDash \mathbb{F}(t)$ *for any* $p \in \mathcal{P}$.

Proof sketch [8, 9]. The function $\mathbb{F}$ is defined inductively. The basis is naturally defined as follows:

1. $\mathbb{F}(\text{pass}) = \top$

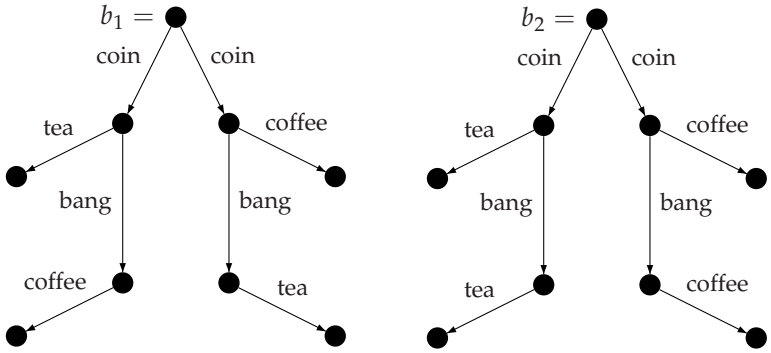


Figure 3.2: Two coffee machines.

2. $\mathbb{F}(\text{stop}) = \bot$

The inductive constructs are then defined as follows:

1. $\mathbb{F}(\mathbf{i}; t) = \mathbb{F}(t)$
2. $\mathbb{F}(a; t) = a \wedge \mathsf{EX}\, \mathbb{F}(t)$
3. $\mathbb{F}(\Sigma T) = \bigvee_{t \in T} \mathbb{F}(t)$, where $\bigvee_{t \in T = \{t_1, \ldots, t_n\}} t$ is the usual shorthand for $t_1 \vee \cdots \vee t_n$
4. $\mathbb{F}(\Sigma T \,\Box\, \theta; t) = (\mathbb{F}(t_1) \vee \mathbb{F}(t_2) \vee \cdots \vee \mathbb{F}(t_n)) \vee (\neg(\mathbb{F}(t_1) \wedge \mathbb{F}(t_2) \wedge \cdots \wedge \mathbb{F}(t_n)) \wedge \mathbb{F}(t))$, where $T = \{t_1, t_2, \ldots, t_n\}$.

Several constructs were not considered since they can be expressed using the above constructs. For example $t_1 \,\Box\, t_2$ is equivalent to $\Sigma\{t_1, t_2\}$ and so the $\Box$ operator does not need individual consideration. The function $\mathbb{F}$ is also defined by assuming without loss of generality that there is at most one deadlock (θ) branch in every choice. The interested reader is directed to the original proof [8, 9] for details. $\Box$

We conclude our description of previous work with the example used originally [9] to illustrate the conversion of failure trace tests into CTL formulae. This example does illustrate this conversion so it is relevant by itself. However, the primary purpose of including it in this manuscript is that we will follow the same example later (see Example 3 at the

end of Chapter 4) to illustrate our conversion function from CTL formulae to failure trace tests, thus going full circle.

Example 1 HOW TO TELL LOGICALLY THAT YOUR COFFEE MACHINE IS WORKING [9].:

The coffee machines b_1 and b_2 below were famously introduced to illustrate the limitations of the may and must testing framework of De Nicola and Hennessy. Indeed, they have been found [21] to be equivalent under testing preorder [15] but not equivalent under stable failure preorder [21].

$$b_1 = \text{coin}; (\text{tea} \,\Box\, \text{bang}; \text{coffee}) \;\Box\; \text{coin}; (\text{coffee} \,\Box\, \text{bang}; \text{tea})$$

$$b_2 = \text{coin}; (\text{tea} \,\Box\, \text{bang}; \text{tea}) \;\Box\; \text{coin}; (\text{coffee} \,\Box\, \text{bang}; \text{coffee})$$

These machines are also shown graphically (as LTS) in Figure 3.2.

The first machine accepts a coin and then dispenses either tea or coffee nondeterministically. One can however hit the machine to change its original choice. The second machine gives either tea or coffee, just as b_1. By contrast with b_1 however the beverage offered will not be changed by hits. One failure trace test that differentiate these machines [21] is

$$t = \text{coin}; (\text{coffee}; \text{pass} \,\Box\, \theta; \text{bang}; \text{coffee}; \text{pass})$$

The conversion of the failure trace test t into a CTL formula as per Proposition 3.2 yields [9]:

$$\mathbb{F}(t) = \text{coin} \wedge \mathsf{EX}\,(\text{coffee} \vee \neg\text{coffee} \wedge \text{bang} \wedge \mathsf{EX}\,\text{coffee})$$

The meaning of this formula is clearly equivalent to the meaning of t, as it literally reads “a coin is expected, and in the next state either coffee is offered,

or coffee is not offered but a bang is available and then the next state will offer coffee."

$\mathbb{F}(t)$ holds for both the initial states of $\mathbb{K}(b_1)$ (where coffee is offered from the outset or follows a hit on the machine) but holds in only one of the initial states of $\mathbb{K}(b_2)$ (the one that dispenses coffee).

Chapter 4

CTL Is Equivalent to Failure Trace Testing

Let as before $\mathcal{P}$ be the set of all processes, $\mathcal{T}$ the set of all failure trace tests, and $\mathcal{F}$ the set of all CTL formulae.

This section presents the main contribution of this thesis. We go the other way around and show that CTL formulae can be converted into failure trace tests. Combined with earlier work in the matter [8, 9] we thus establish that CTL formulae and failure trace tests are equivalent:

Theorem 4.1 *For some $t \in \mathcal{T}$ and $f \in \mathcal{F}$, whenever p* may *t if and only if $\mathbb{K}(p) \vDash f$ for any $p \in \mathcal{P}$ we say that t and f are equivalent. Then, for every failure trace test there exists an equivalent CTL formula and the other way around. Furthermore a failure trace test can be algorithmically converted into its equivalent CTL formula and the other way around.*

Proof. Immediate from Proposition 3.2 (in Section 3.2) and Theorem 4.5 (below). The algorithmic nature of the conversion is shown implicitly in the proofs of these two results. □

We find it convenient to show first how to construct logical (but not temporal) combinations of tests.

Lemma 4.2 *For any test $t \in \mathcal{T}$ there exists a test $\bar{t} \in \mathcal{T}$ such that p may t if and only if $\neg(p \text{ may } \bar{t})$ for any $p \in \mathcal{P}$.*

Proof. We follow the original construction for negation [8, 9], which continues to work.

Specifically, we modify t to produce $\bar{t}$ as follows: We first force all the success states to become deadlock states by eliminating the outgoing action γ from t completely. If the action that leads to a state thus converted is θ then this action is removed as well (indeed, the "fail if nothing else works" phenomenon thus eliminated is implicit in testing). Finally, we add to all the states having their outgoing transitions θ or γ eliminated in the previous steps one outgoing transition labeled θ followed by one outgoing transition labeled γ.

The test $\bar{t}$ must fail every time the original test t succeeds. The first step (eliminating γ transitions) has exactly this effect.

In addition, we must ensure that $\bar{t}$ succeeds in all the circumstances in which t fails. The extra θ outgoing actions (followed by γ) ensure that whevever the end of the run reaches a state that was not successful in t (meaning that it had no outgoing γ transition) then this run is extended in $\bar{t}$ via the θ branch to a success state, as desired. □

Lemma 4.3 *For any two tests $t_1, t_2 \in \mathcal{T}$ there exists a test $t_1 \vee t_2 \in \mathcal{T}$ such that p may $(t_1 \vee t_2)$ if and only if $(p \text{ may } t_1) \vee (p \text{ may } t_2)$ for any $p \in \mathcal{P}$.*

Proof. We construct such a disjunction on tests by induction over the structure of tests.

For the base case it is immediate that pass $\vee\, t = t \vee$ pass $=$ pass and stop $\vee\, t = t \vee$ stop $= t$ for any structure of the test t.

For the induction step we consider without loss of generality that t_1 and t_2 have the following structure:

$$\begin{aligned} t_1 &= \Sigma\{b_1; t_1(b_1) : b_1 \in B_1\} \,\Box\, \theta; t_{N1} \\ t_2 &= \Sigma\{b_2; t_2(b_2) : b_2 \in B_2\} \,\Box\, \theta; t_{N2} \end{aligned}$$

Indeed, all the other possible structures of t_1 (and t_2) are covered by such a form since θ not appearing on the top level of t_1 (or t_2) is equivalent to $t_{N1} = \text{stop}$ (or $t_{N2} = \text{stop}$), while not having a choice on the top level of the test is equivalent to B_1 (or B_2) being an appropriate singleton.

We then construct $t_1 \vee t_2$ for the form of t_1 and t_2 mentioned above under the inductive assumption that the disjunction between any of the "inner" tests $t_1(b_1)$, t_{N1} $t_2(b_2)$, and t_{N2} is known. We have:

$$\begin{array}{lll} \Sigma\{b_1; t_1(b_1) : b_1 \in B_1\} \,\Box\, \theta; t_{N1} \quad \vee \quad \Sigma\{b_2; t_2(b_2) : b_2 \in B_2\} \,\Box\, \theta; t_{N2} & & \\ = \quad \Sigma\{b; (t_1(b) \vee t_2(b)) : b \in B_1 \cap B_2\} & \Box & \\ \quad\;\; \Sigma\{b; (t_1(b) \vee [t_{N2}]_b) : b \in B_1 \setminus B_2\} & \Box & \\ \quad\;\; \Sigma\{b; (t_2(b) \vee [t_{N1}]_b) : b \in B_2 \setminus B_1\} & \Box & \\ \quad\;\; \theta; (t_{N1} \vee t_{N2}) & & \end{array} \tag{4.1}$$

where $[t]_b$ is the test t restricted to performing b as its first action and so is inductively constructed as follows:

1. If $t = \text{stop}$ then $[t]_b = \text{stop}$.
2. If $t = \text{pass}$ then $[t]_b = \text{pass}$.
3. If $t = b; t'$ then $[t]_b = t'$.
4. If $t = a; t'$ with $a \neq b$ then $[t]_b = \text{stop}$.
5. If $t = \mathbf{i}; t'$ then $[t]_b = [t']_b$.
6. If $t = t' \,\Box\, t''$ such that neither t' nor t'' contain θ in their topmost choice then $[t]_b = [t']_b \,\Box\, [t'']_b$.
7. If $t = b; t' \,\Box\, \theta; t''$ then $[t]_b = t'$.
8. If $t = a; t' \,\Box\, \theta; t''$ with $a \neq b$ then $[t]_b = [t'']_b$.

If the test $t_1 \vee t_2$ is offered an action b that is common to the top choices of the two tests t_1 and t_2 ($b \in B_1 \cap B_2$) then the disjunction succeeds if and only if b is performed

and then at least one of the tests $t_1(b)$ and $t_2(b)$ succeeds (meaning that $t_1(b) \vee t_2(b)$ succeeds inductively) afterward. The first term of the choice in Equation (4.1) represents this possibility.

If the test is offered an action b that appears in the top choice of t_1 but not in the top choice of t_2 ($b \in B_1 \setminus B_2$) then the disjunction succeeds if and only if $t_1(b)$ succeeds after b is performed, or t_{N2} performs b and then succeeds; the same goes for $b \in B_2 \setminus B_1$ (only in reverse). The second and the third terms of the choice in Equation (4.1) represent this possibility.

Finally whenever the test $t_1 \vee t_2$ is offered an action b that is in neither the top choices of the two component tests t_1 and t_2 (that is, $b \notin B_1 \cup B_2$), then the disjunction succeeds if and only if at least one of the tests t_{N1} or t_{N2} succeeds (or equivalently $t_{N1} \vee t_{N2}$ succeeds inductively). This is captured by the last term of the choice in Equation (4.1). Indeed, $b \notin B_1 \cup B_2$ implies that $b \notin (B_1 \cap B_2) \cup (B_1 \setminus B_2) \cup (B_2 \setminus B_1)$ and so such an action will trigger the deadlock detection (θ) choice.

There is no other way for the test $t_1 \vee t_2$ to succeed so the construction is complete. $\square$

We believe that an actual example of how disjunction is constructed is instructive. The following is therefore an example to better illustrate disjunction over tests. A further example (incorporating temporal operators and also negation) will be provided later (see Example 3).

Example 2 A DISJUNCTION OF TESTS:

Consider the construction $t_1 \vee t_2$, where:

$$\begin{aligned} t_1 \;=\; & (\text{bang}; \text{tea}; \text{pass}) \;\square \\ & (\text{coin}; (\text{coffee}; \text{stop} \;\square\; \theta; \text{pass})) \end{aligned}$$

$$\begin{aligned} t_2 \;=\; & (\text{coin}; \text{pass}) \,\Box \\ & (\text{nudge}; \text{stop}) \,\Box \\ & (\theta; (\text{bang}; \text{water}; \text{pass} \,\Box\, \text{turn}; \text{stop})) \end{aligned}$$

Using the notation from Equation (4.1) we have $B_1 = \{\text{coin}, \text{bang}\}$, $B_2 = \{\text{coin}, \text{nudge}\}$ and so $B_1 \cap B_2 = \{\text{coin}\}$, $B_1 \setminus B_2 = \{\text{bang}\}$, and $B_2 \setminus B_1 = \{\text{nudge}\}$. We further note that $t_1(\text{coin}) = \text{coffee}; \text{stop} \,\Box\, \theta; \text{pass}$, $t_1(\text{bang}) = \text{tea}; \text{pass}$, $t_{N1} = \text{stop}$, $t_2(\text{coin}) = \text{pass}$, $t_2(\text{nudge}) = \text{stop}$, and $t_{N2} = \text{bang}; \text{water}; \text{pass} \,\Box\, \text{turn}; \text{stop}$. Therefore we have:

$$\begin{aligned} t_1 \vee t_2 \;=\; & (\text{coin}; (t_1(\text{coin}) \vee t_2(\text{coin}))) \,\Box \\ & (\text{bang}; (t_1(\text{bang}) \vee [t_{N2}]_{\text{bang}})) \,\Box \\ & (\text{nudge}; (t_2(\text{nudge}) \vee [t_{N1}]_{\text{nudge}})) \,\Box \\ & \theta; (t_{N1} \vee t_{N2}) \\ =\; & (\text{coin}; \text{pass}) \,\Box \\ & (\text{bang}; (t_1(\text{bang}) \vee [t_{N2}]_{\text{bang}})) \,\Box \\ & (\text{nudge}; \text{stop}) \,\Box \\ & (\theta; (\text{bang}; \text{water}; \text{pass} \,\Box\, \text{turn}; \text{stop})) \end{aligned}$$

Indeed, $t_1(\text{coin}) = \text{pass}$ so $t_1(\text{coin}) \vee t_2(\text{coin}) = \text{pass}$; $[t_{N1}]_{\text{nudge}} = \text{stop}$; and $t_{N1} = \text{stop}$ so $t_{N1} \vee t_{N2} = t_{N2}$.

We further have $[t_{N2}]_{\text{bang}} = \text{water}; \text{pass}$, and therefore $t_1(\text{bang}) \vee [t_{N2}]_{\text{bang}} = (\text{tea}; \text{pass}) \vee (\text{water}; \text{pass})$. We proceed inductively as above, except that in this degenerate case $t_{N1} = t_{N2} = \text{stop}$ and $B_1 \cap B_2 = \emptyset$, so the result is a simple choice between the components: $t_1(\text{bang}) \vee [t_{N2}]_{\text{bang}} = (\text{tea}; \text{pass}) \,\Box\, (\text{water}; \text{pass})$. Overall we reach the following result:

$$
\begin{aligned}
t_1 \vee t_2 \quad = \quad & (\text{coin}; \text{pass}) \,\Box \\
& (\text{bang}; ((\text{tea}; \text{pass}) \,\Box\, (\text{water}; \text{pass}))) \,\Box \\
& (\text{nudge}; \text{stop}) \,\Box \\
& (\theta; (\text{bang}; \text{water}; \text{pass} \,\Box\, \text{turn}; \text{stop}))
\end{aligned} \tag{4.2}
$$

Intuitively, t_1 specifies that we can have tea if we hit the machine, and if we put a coin in we can get anything except coffee. Similarly t_2 specifies that we can put a coin in the machine, we cannot nudge it, and if none of the above happen then we can either hit the machine (case in which we get water) or we can turn it upside down. On the other hand, the disjunction of these tests as shown in Equation (4.2) imposes the following specification:

1. If a coin is inserted then the test succeeds. Indeed, this case from t_2 supersedes the corresponding special case from t_1: the success of the test implies that the machine is allowed to do anything afterward, including not dispensing coffee.
2. We get either tea or water after hitting the machine. Getting tea comes from t_1 and getting water from t_2 (where the bang event comes from the θ branch).
3. If we nudge the machine then the test fails immediately; this comes directly from t_2.
4. In all the other cases the test behaves like the θ branch of t_2. This behaviour makes sense since there is no such a branch in t_1.

We can thus see how the disjunction construction from Lemma 4.3 makes intuitive sense.

This all being said, note that we do not claim that either of the tests t_1 and

t_2 are useful in any way, and so we should not be held responsible for the behaviour of any machine built according to the disjunctive specification shown above.

Corollary 4.4 *For any tests $t_1, t_2 \in \mathcal{T}$ there exists a test $t_1 \wedge t_2 \in \mathcal{T}$ such that p may $(t_1 \wedge t_2)$ if and only if $(p$ may $t_1) \wedge (p$ may $t_2)$ for any $p \in \mathcal{P}$.*

Proof. Immediate from Lemmata 4.2 and 4.3 using the De Morgan rule $a \wedge b = \neg(\neg a \vee \neg b)$. □

We are now ready to show that any CTL formula can be converted into an equivalent failure trace test.

Theorem 4.5 *There exists a function $\mathbb{T} : \mathcal{F} \to \mathcal{T}$ such that $\mathbb{K}(p) \vDash f$ if and only if p may $\mathbb{T}(f)$ for any $p \in \mathcal{P}$.*

Proof. The proof is done by structural induction over CTL formulae. As before, the function $\mathbb{T}$ will also be defined recursively in the process.

We have naturally $\mathbb{T}(\top) =$ pass and $\mathbb{T}(\bot) =$ stop. Clearly any Kripke structure satisfies $\top$ and any process passes pass, so it is immediate that $\mathbb{K}(p) \vDash \top$ if and only if p may pass $= \mathbb{T}(\top)$. Similarly $\mathbb{K}(p) \vDash \bot$ if and only if p may stop is immediate (neither is ever true).

To complete the basis we have $\mathbb{T}(a) = a$; pass, which is an immediate consequence of the definition of $\mathbb{K}$. Indeed, the construction defined in Proposition 3.1 ensures that for every outgoing action a of an LTS process p there will be an initial Kripke state in $\mathbb{K}(p)$ where a holds and so p may a; pass if and only if $\mathbb{K}(p) \vDash a$.

The constructions for non-temporal logical operators have already been presented in Lemma 4.2, Lemma 4.3, and Corollary 4.4. We therefore have $\mathbb{T}(\neg f) = \overline{\mathbb{T}(f)}$ with $\overline{\mathbb{T}(f)}$ as constructed in Lemma 4.2, while $\mathbb{T}(f_1 \vee f_2) = \mathbb{T}(f_1) \vee \mathbb{T}(f_2)$ and $\mathbb{T}(f_1 \wedge f_2) =$

$\mathbb{T}(f_1) \wedge \mathbb{T}(f_2)$ with test disjunction and conjunction as in Lemma 4.3 and Corollary 4.4, respectively. That these constructions are correct follow directly from the respective lemmata and corollary.

We then move to the temporal operators. Their conversion comes largely (but not completely) unmodified from the original proof [8].

We have $\mathbb{T}(\mathsf{EX}\, f) = \Sigma\{a; \mathbb{T}(f) : a \in A\}$. When applied to some process p the resulting test performs any action $a \in A$ and then (in the next state p' such that $p \xrightarrow{a} p'$) gives control to $\mathbb{T}(f)$. $\mathbb{T}(f)$ will always test the next state (since there is no θ in the top choice), and there is no restriction as to what particular next state p' is expected (since any action of p is accepted by the test).

Concretely, suppose that p may $\Sigma\{a; \mathbb{T}(f) : a \in A\}$. Then there exists some action $a \in A$ such that p performs a, becomes p', and p' may $\mathbb{T}(f)$ (by definition of may testing). It follows by inductive assumption that $\mathbb{K}(p') \vDash f$ and so $\mathbb{K}(p) \vDash \mathsf{X}\, f$ (since $\mathbb{K}(p')$ is one successor of $\mathbb{K}(p)$). Conversely, by the definition of $\mathbb{K}$ all successors of $\mathbb{K}(p)$ have the form $\mathbb{K}(p')$ such that $p \xrightarrow{a} p'$ for some $a \in A$. Suppose then that $\mathbb{K}(p) \vDash \mathsf{X}\, f$. Then there exists a successor $\mathbb{K}(p')$ of $\mathbb{K}(p)$ such that $\mathbb{K}(p') \vDash f$, which is equivalent to p' may $\mathbb{T}(f)$ (by inductive assumption), which in turn implies that $(p = a; p')$ may $\Sigma\{a; \mathbb{T}(f) : a \in A\}$ (by the definition of may testing), as desired.

We then have $\mathbb{T}(\mathsf{EF}\, f) = t'$ such that $t' = \mathbb{T}(f) \,\square\, (\Sigma(a; t' : a \in A))$. The test t' is shown graphically in Figure 4.1(a). It specifies that at any given time the system under test has a choice to either pass $\mathbb{T}(f)$ or perform some (any) action and then pass t' anew. Repeating this description recursively we conclude that to be successful the system under test can pass $\mathbb{T}(f)$, or perform an action and then pass $\mathbb{T}(f)$, or perform two actions and then pass $\mathbb{T}(f)$, and so on. The overall effect (which is shown in Figure 4.1(b)) is that exactly all the processes p that perform an arbitrary sequence of actions and then pass $\mathbb{T}(f)$ at the end of this sequence will pass $t' = \mathbb{T}(\mathsf{EF}\, f)$. Given the inductive assumption that $\mathbb{T}(f)$ is equivalent to f this is equivalent to $\mathbb{K}(p)$ being the start of an arbitrary path to some state

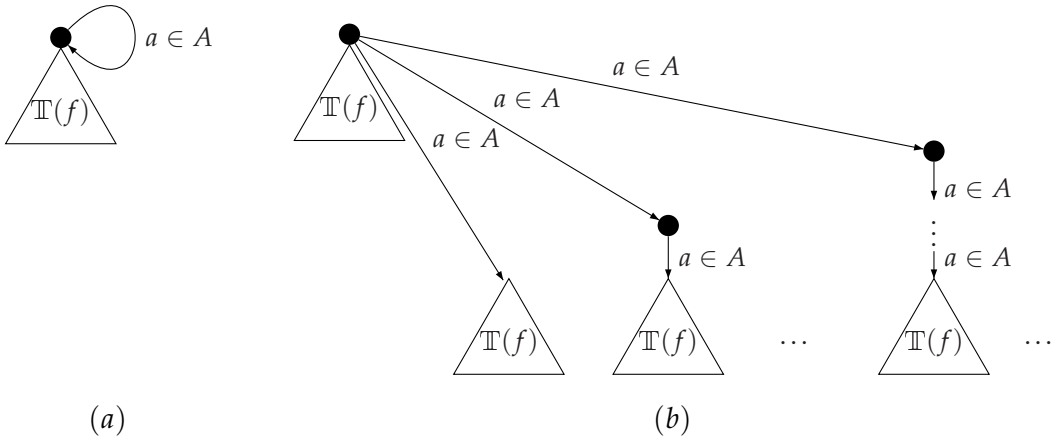


Figure 4.1: Test equivalent to the CTL formula $\mathsf{EF}\ f$ (a) and its unfolded version (b).

that satisfies f, which is precisely the definition of $\mathbb{K}(p) \vDash \mathsf{EF}\ f$, as desired.

Following a similar line of thought we have $\mathbb{T}(\mathsf{EG}\ f) = \mathbb{T}(f) \wedge (\mathbb{T}(\mathsf{EX}\ f') \,\Box\, \theta; \mathrm{pass})$, with $f' = f \wedge \mathsf{EX}\ f'$. Suppose that p may $\mathbb{T}(\mathsf{EG}\ f)$. This implies that p may $\mathbb{T}(f)$. This also imply that p may $\mathbb{T}(\mathsf{EX}\ f')$ but only unless $p = \mathrm{stop}$; indeed, the θ choice appears in conjunction with a multiple choice that offers all the possible alternatives (see the conversion for EX above) and so can only be taken if no other action is available.

By inductive assumption p may $\mathbb{T}(f)$ if and only if $\mathbb{K}(p) \vDash f$. By the conversion of EX (see above) p may $\mathbb{T}(\mathsf{EX}\ f')$ if and only if $\mathbb{K}(p') \vDash f'$ where $p \xrightarrow{a} p'$ for some $a \in A$ and so $\mathbb{K}(p')$ is the successor of $\mathbb{K}(p)$ on some path. We thus have p may $\mathbb{T}(\mathsf{EG}\ f)$ if and only if $\mathbb{K}(p) \vDash f$ and $\mathbb{K}(p') \vDash f'$ for some successor p' of p. Repeating the reasoning above recursively (starting from p', etc.) we conclude that p may $\mathbb{T}(\mathsf{EG}\ f)$ if and only if $\mathbb{K}(p_1) \vDash f$ for all the states $\mathbb{K}(p_1)$ on some path that starts from $\mathbb{K}(p)$, which is clearly equivalent to $\mathbb{K}(p) \vDash \mathsf{EG}\ f'$. The recursive reasoning terminates at the end of the path, when the LTS state p becomes stop, and the process is therefore released from its obligation to have states in which f holds (since no states exist any longer). In this case $\mathbb{K}(p)$ is a "sink" state with no successor (according to Item 3c in Proposition 3.1) and so the corresponding Kripke path is also at an end (and so there are no more states for f to hold in).

Finally we have $\mathbb{T}(\mathsf{E}\ f_1\ \mathsf{U}\ f_2) = (\mathbb{T}(f_1) \wedge (\mathbb{T}(\mathsf{EX}\ f') \ \Box\ \theta; \text{pass})) \ \Box\ \mathbf{i}; (\mathbb{T}(f_2) \wedge (\mathbb{T}(\mathsf{EX}\ f'') \ \Box\ \theta; \text{pass}))$, with $f' = f_1 \wedge \mathsf{EX}\ f'$ and $f'' = f_2 \wedge \mathsf{EX}\ f''$. Following the same reasoning as above (in the EG case) the fact that a process p follows the test $\mathbb{T}(f_1) \wedge \mathbb{T}(\mathsf{EX}\ f')$ without deadlocking until some arbitrary state p' is reached is equivalent to $\mathbb{K}(p)$ featuring a path of arbitrary length ending in state $\mathbb{K}(p')$ whose states $\mathbb{K}(p_1)$ will all satisfy f_1. At the arbitrary (and nondeterministically chosen) point p' the test $\mathbb{T}(\mathsf{E}\ f_1\ \mathsf{U}\ f_2)$ will exercise its choice and so p' has to pass $\mathbb{T}(f_2) \wedge \mathbb{T}(\mathsf{EX}\ f'')$ in order for p to pass $\mathbb{T}(\mathsf{E}\ f_1\ \mathsf{U}\ f_2)$. We follow once more the same reasoning as in the EG case and we thus conclude that this is equivalent to $\mathbb{K}(p_1) \vDash f$ for all the states $\mathbb{K}(p_1)$ on some path that starts from $\mathbb{K}(p')$. Putting the two phases together we have that p may $\mathbb{T}(\mathsf{E}\ f_1\ \mathsf{U}\ f_2)$ if and only if $\mathbb{K}(p)$ features a path along which f_1 holds up to some state and f_2 holds from that state on, which is equivalent to $\mathbb{K}(p) \vDash \mathsf{E}\ f_1\ \mathsf{U}\ f_2$, as desired. The θ; pass choices play the same role as in the EG case (namely, they account for the end of a path since they can only be taken when no other action is available).

Thus we complete the proof and the conversion between CTL formulae and sequential tests. Indeed, note that EX, EF, EG, and EU is a minimal set of temporal operators for CTL [12], so all the remaining CTL constructs can be rewritten using only the constructs discussed above. □

Example 3 HOW TO TEST THAT YOUR COFFEE MACHINE IS WORKING:

We turn our attention again to the coffee machines b_1 and b_2 below, as presented earlier in Example 1, and also graphically as LTS in Figure 3.2:

$$b_1 = \text{coin}; (\text{tea} \ \Box\ \text{bang}; \text{coffee}) \ \Box\ \text{coin}; (\text{coffee} \ \Box\ \text{bang}; \text{tea})$$

$$b_2 = \text{coin}; (\text{tea} \ \Box\ \text{bang}; \text{tea}) \ \Box\ \text{coin}; (\text{coffee} \ \Box\ \text{bang}; \text{coffee})$$

Also recall that the following CTL formula was found to differentiate between

the two machines:

$$\phi = \text{coin} \wedge \mathsf{EX}\,(\text{coffee} \vee \neg\text{coffee} \wedge \text{bang} \wedge \mathsf{EX}\,\text{coffee})$$

Indeed, the formula holds for both the initial states of $\mathbb{K}(b_1)$ (where coffee is offered from the outset or follows a hit on the machine) but holds in only one of the initial states of $\mathbb{K}(b_2)$ (the one that dispenses coffee).

Let $C = \{\text{coin}, \text{tea}, \text{bang}, \text{coffee}\}$ be the set of all actions. We will henceforth convert silently all the tests $\theta;\text{pass}$ into pass as long as these tests do not participate in a choice. We will also perform simplifications of the intermediate tests as we go along in order to simplify (and so clarify) the presentation.

The process of converting the formula ϕ to a test suite $\mathbb{T}(\phi)$ then goes as follows:

1. We convert first $\text{bang} \wedge \mathsf{EX}\,\text{coffee} = \neg(\neg\text{bang} \vee \neg\mathsf{EX}\,\text{coffee})$.

   We have $\mathbb{T}(\neg\text{bang}) = \text{bang};\text{stop} \;\Box\; \theta;\text{pass}$. On the other hand $\mathbb{T}(\mathsf{EX}\,\text{coffee}) = \Sigma\{a;\text{coffee};\text{pass} : a \in C\}$ and so $\mathbb{T}(\neg\mathsf{EX}\,\text{coffee}) = \Sigma\{a;\text{coffee};\text{stop} \;\Box\; \theta;\text{pass} : a \in C\} \;\Box\; \theta;\text{pass} = \Sigma\{a;\text{coffee};\text{stop} \;\Box\; \theta;\text{pass} : a \in C\}$ (we ignore the topmost θ branch since the rest of the choice covers all the possible actions).

   Now we compute the disjunction $\mathbb{T}(\neg\text{bang}) \vee \mathbb{T}(\neg\mathsf{EX}\,\text{coffee})$. With the notations used in the proof of Lemma 4.3 we have $B_1 = \{\text{bang}\}$, $B_2 = C$, $t_1(\text{bang}) = \text{stop}$, $t_{N1} = \text{pass}$, $t_2(b) = \text{coffee};\text{stop} \;\Box\; \theta;\text{pass}$ for all $b \in B_2$, and $t_{N2} = \text{pass}$. We therefore have $\mathbb{T}(\neg\text{coffee}) \vee \mathbb{T}(\neg\mathsf{EX}\,\text{coffee}) = \text{bang};(\text{coffee};\text{stop} \;\Box\; \theta;\text{pass}) \;\Box\; \Sigma\{a;\text{pass} : a \in C \setminus \{\text{bang}\}\} \;\Box\; \theta;\text{pass}$. Therefore:

$$\begin{aligned} \mathbb{T}(\neg(\text{bang} \wedge \mathsf{EX}\,\text{coffee})) \;=\; & \text{bang};(\text{coffee};\text{stop} \;\Box\; \theta;\text{pass}) \\ & \Box\; \theta;\text{pass} \end{aligned} \tag{4.3}$$

For brevity we integrated the $C \setminus \{\text{bang}\}$ into the θ branch. Negating this test yields:

$$\mathbb{T}(\text{bang} \wedge \mathsf{EX}\ \text{coffee}) \quad = \quad \text{bang}; \text{coffee}; \text{pass}$$

Note in passing that that the negated version as shown in Equation (4.3) will suffice. The last formula is only provided for completeness and also as a checkpoint in the conversion process. Indeed, the equivalence between bang $\wedge$ EX coffee and bang; coffee; pass can be readily ascertained intuitively.

2. We move to $\neg\text{coffee} \wedge \text{bang} \wedge \mathsf{EX}\ \text{coffee} = \neg(\text{coffee} \vee \neg(\text{bang} \wedge \mathsf{EX}\ \text{coffee}))$. By Equation (4.3) we have $\mathbb{T}(\text{coffee} \vee \neg(\text{bang} \wedge \mathsf{EX}\ \text{coffee})) = \text{coffee}; \text{pass} \vee (\text{bang}; (\text{coffee}; \text{stop} \,\Box\, \theta; \text{pass}) \,\Box\, \theta; \text{pass})$. This time $B_1 = \{\text{coffee}\}$, $B_2 = \{\text{bang}\}$, $t_1(b) = \text{pass}$, $t_2(b) = \text{coffee}; \text{stop} \,\Box\, \theta; \text{pass}$, $t_{N1} = \text{stop}$, and $t_{N2} = \text{pass}$. Therefore $\mathbb{T}(\text{coffee} \vee \neg(\text{bang} \wedge \mathsf{EX}\ \text{coffee})) = \text{coffee}; \text{pass} \,\Box\, \text{bang}; (\text{coffee}; \text{stop} \,\Box\, \theta; \text{pass}) \,\Box\, \theta; \text{pass}$ (note that $B_1 \cap B_2 = \emptyset$). Negating this test yields:

$$\begin{aligned} \mathbb{T}(\neg\text{coffee} \wedge \text{bang} \wedge \mathsf{EX}\ \text{coffee}) \quad = \quad & \text{coffee}; \text{stop} \,\Box \\ & \text{bang}; \text{coffee}; \text{pass} \end{aligned} \tag{4.4}$$

This is yet another checkpoint in the conversion, as the equivalence above can be once more easily ascertained.

3. The conversion of coffee $\vee \neg$coffee $\wedge$ bang $\wedge$ EX coffee combines in a disjunction the test coffee; pass and the test from Equation (4.4). None of these tests feature a θ branch in their top choice and so the combination is a simple choice between the two:

$$\begin{aligned} & \mathbb{T}(\text{coffee} \vee \neg\text{coffee} \wedge \text{bang} \wedge \mathsf{EX}\ \text{coffee}) \\ = \quad & \text{coffee}; \text{pass} \,\Box\, \text{bang}; \text{coffee}; \text{pass} \end{aligned} \tag{4.5}$$

In what follows we use for brevity $\phi' = \text{coffee} \vee \neg\text{coffee} \wedge \text{bang} \wedge \mathsf{EX}\ \text{coffee}$.

4. We have $\mathbb{T}(\mathsf{EX}\ \phi') = \Sigma\{a; \mathbb{T}(\phi') : a \in C\}$ and therefore

$$\begin{aligned} \mathbb{T}(\mathsf{EX}\ \phi') \quad = \quad & \Sigma\{a; (\text{coffee}; \text{pass} \\ & \qquad \Box\ \text{bang}; \text{coffee}; \text{pass}) : a \in C\} \end{aligned} \tag{4.6}$$

We will actually need in what follows the negation of this formula, which is the following:

$$\begin{aligned} \mathbb{T}(\neg\mathsf{EX}\ \phi') \quad = \quad & \Sigma\{a; (\text{coffee}; \text{stop} \\ & \qquad \Box\ \text{bang}; (\text{coffee}; \text{stop} \Box\ \theta; \text{pass}) \\ & \qquad \Box\ \theta; \text{pass}) : a \in C\} \\ & \Box\ \theta; \text{pass} \end{aligned} \tag{4.7}$$

At this point our test becomes complex enough so that we can use it to illustrate in more detail the negation algorithm (Lemma 4.2). We thus take this opportunity to explain in detail the conversion between $\mathbb{T}(\mathsf{EX}\ \phi')$ from Equation (4.6) and $\overline{\mathbb{T}(\mathsf{EX}\ \phi')} = \mathbb{T}(\neg\mathsf{EX}\ \phi')$ shown in Equation (4.7).

(a) The test $\mathbb{T}(\mathsf{EX}\ \phi')$ is shown as an LTS in Figure 4.2(a). For convenience all the states are labeled t_i, $1 \leq i \leq 7$ so that we can easily refer to them.

(b) We then eliminate all the success (γ) transitions. The states t_3 and t_6 are thus converted from pass to stop.

(c) All the states that were not converted in the previous step (that is, states t_1, t_2, and t_4) gain a θ; pass branch. The result is the negation of the original test and is shown in Figure 4.2(b).

(d) If needed, the conversion the other way around would proceed as follows: The success transitions are eliminated (this affects t_8, t_9, and

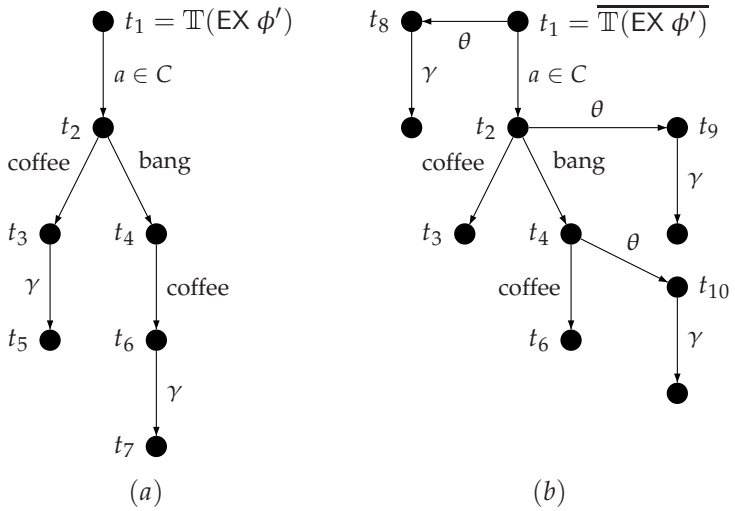


Figure 4.2: Conversion of the test $\mathbb{T}(\mathsf{EX}\ \phi')$ (a) into its negation $\overline{\mathbb{T}(\mathsf{EX}\ \phi')}$ (b).

t_{10}). The preceding θ transitions are also eliminated (which eliminates t_8, t_9, t_{10} and affects the states t_1, t_2, and t_4). The states unaffected by this process are t_3 and t_6 so they both gain a θ; pass branch; however, such a branch is the only outgoing one for both t_3 and t_6 so it is equivalent to a simple pass in both cases. The result is precisely the test $\mathbb{T}(\mathsf{EX}\ \phi')$ as shown in Figure 4.2(a).

5. We finally reach the top formula ϕ. Indeed, $\phi = \text{coin} \wedge \mathsf{EX}\ \phi' = \neg(\neg\text{coin} \vee \neg\mathsf{EX}\ \phi')$. We thus need to combine in a disjunction the test coin; stop $\Box$ θ; pass with the test shown in Equation (4.7). We have:

$$\begin{aligned}\mathbb{T}(\neg\text{coffee} \vee \neg\mathsf{EX}\ \phi') \quad = \quad & \text{coin}; (\text{coffee}; \text{stop} \\ & \Box\ \text{bang}; (\text{coffee}; \text{stop} \Box\ \theta; \text{pass}) \\ & \Box\ \theta; \text{pass}) \\ & \Box\ \Sigma\{b; \text{pass} : b \in C \setminus \{\text{coin}\}\} \\ & \Box\ \theta; \text{pass}\end{aligned}$$

Indeed, $B_1 = \{\text{coin}\}$, $B_2 = C$, $t_1(b) = \text{stop}$, $t_2(b) =$

coffee; stop $\Box$ bang; (coffee; stop $\Box$ θ; pass) $\Box$ θ; pass, and $t_{N1} = t_{N2} =$ pass.

To reduce the size of the expression we combine the $C \setminus \{\text{coin}\}$ and θ choices and so we obtain:

$$\begin{aligned}\mathbb{T}(\neg\text{coffee} \vee \neg\mathsf{EX}\,\phi') \quad = \quad & \text{coin}; (\text{coffee}; \text{stop} \\ & \Box\, \text{bang}; (\text{coffee}; \text{stop} \Box\, \theta; \text{pass}) \\ & \Box\, \theta; \text{pass}) \\ & \Box\, \theta; \text{pass}\end{aligned}$$

Negating the above expression results in the test equivalent to the original formula:

$$\mathbb{T}(\phi) \quad = \quad \text{coin}; (\text{coffee}; \text{pass} \Box\, \text{bang}; \text{coffee}; \text{pass})$$

Recall now that the test we started from in Example 1 was slightly different, namely:

$$t = \text{coin}; (\text{coffee}; \text{pass} \Box\, \theta; \text{bang}; \text{coffee}; \text{pass})$$

We argue however that these two tests are in this case equivalent. Indeed, both tests succeed whenever coin is followed by coffee. Suppose now that coin does happen but the next action is not coffee. Then t will follow on the deadlock detection branch, which will only succeed if the next action is bang. On the other hand $\mathbb{T}(\phi)$ does not have a deadlock detection branch in the choice following coin; however, the only alternative to coffee in $\mathbb{T}(\phi)$ is bang, which is precisely the same alternative as for t (as shown above). We thus conclude that t and $\mathbb{T}(\phi)$ are indeed equivalent.

Chapter 5

Conclusions

Our work builds on the previous work summarized earlier in Sections 3.1 and 3.2 [8, 9]. This work consists of the definition of an equivalence between a process or a labeled transition system and a Kripke structure (Definition 3.3), the development of an algorithmic function $\mathbb{K}$ that converts any labeled transition system into an equivalent Kripke structure (Proposition 3.1), and then the development of an algorithmic function $\mathbb{F}$ that converts the failure trace tests to equivalent CTL formulae (Proposition 3.2). A function for the conversion the other way around was also developed [8], but one of the authors of the original paper subsequently realized that this conversion is incorrect [6].

The purpose of the present work was thus to fix this conversion function. We therefore constructed an algorithmic function $\mathbb{T}$ that converts any CTL formula into an equivalent failure trace test (Theorem 4.5). Combined with the previous work on the matter we have shown that failure trace tests and CTL formulae are equivalent (Theorem 4.1).

Given the nature of our work, the conclusions of the original paper apply, while no substantial new conclusions are introduced. The reminder of this chapter is therefore dedicated to a revision of the previous conclusions [8] and so covers the whole (and now fully correct) work.

As already mentioned, the function $\mathbb{K}$ [8] creates a Kripke structure that may have multiple initial states, and so a weaker satisfaction operator (over sets of states rather than

states) was needed. It was further noted [8] that this issue only happens for those LTS that start with a choice of multiple visible actions (a phenomenon that was deemed somehow incorrectly "initial nondeterminism" in the original paper). It follows that in order to eliminate the need for a new satisfaction operator (over sets of states) one can simply create an extra LTS state which becomes the initial state and performs an artificial "start" action to give control to the original initial state. It was originally claimed that when this is done the proofs of Proposition 3.1, Proposition 3.2, and Theorem 4.5 all revert to the normal satisfaction operator (over states), and so the results are without loss of generality.

This being said, the claim mentioned above still needs to be verified. Alternatively (and ideally) we believe that the investigation into a conversion between LTS and Kripke structures that preserves the nice properties (used subsequently) of the conversion described in Definition 3.3 and which also copes directly with the so-called initial nondeterminism is worth pursuing.

We believe that our results (providing a combined, logical and algebraic method of system verification) has unquestionable advantages. To emphasize this consider the scenario of a network communication protocol between two end points and through some communication medium being formally specified. The two end points are likely to be algorithmic (or even finite state machines) and so the natural way of specifying them is algebraic. The communication medium on the other hand has a far more loose specification. Indeed, it is likely that not even the actual properties are fully known at specification time, since they can vary widely when the protocol is actually deployed (between say, the properties of a 6-foot direct Ethernet link and the properties of a nondeterministic and congested Internet route between Afghanistan and Zimbabwe). The properties of the communication medium are therefore more suitable for logic specification. Such a scenario is also applicable to systems with components at different levels of maturity (some being fully implemented already while others being at the prototype stage of even not being implemented at all and so less suitable for being specified algebraically). Our work enables

precisely this kind of mixed specification. In fact no matter how the system is specified we enable the application of either model checking of model-based testing (or even both) on it, depending on suitability or even personal taste.

The original paper [8] identified several directions of further investigation. We believe that these directions continue to be pertinent and interesting.

First, the issue of tests taking an infinite time to complete is an ever-present issue in model-based testing. Our conversion of CTL formulae is no exception, as the tests resulting from the conversion of expressions that use EF, EG, and EU fall all into this category. Furthermore Rice's theorem [22] (which states that any non-trivial and extensional property of programs is undecidable) guarantees that tests that take an infinite time to complete will continue to exist no matter how much we refine our conversion algorithms. We therefore believe that it is very useful to investigate methods and algorithms for partial (or incremental) application of tests. Such methods will offer increasingly stronger guarantees of correctness as the test progresses, and total correctness at the limit (when the test completes).

It continues to be potentially interesting to extend this work to other temporal logics (such as CTL*) and whatever testing framework turns out to be equivalent to it (in the same sense as used in our work).

Finally, it is worth noting that the original paper [8] did mention a substantial disadvantage of the conversion of tests to CTL formulae. Specifically, the resulting $\mathbb{F}(t)$ are almost surely not in their simplest form, as the conversion does not really exploit the expressiveness of temporal operators. In fact the resulting formulae may even have infinite length. This thesis did not modify anything in this conversion and so this issue is outside the scope of our work, but bringing these formulae to more compact forms is definitely an immediate open problem.

Bibliography

[1] R. ALUR AND D. L. DILL, *A theory of timed automata*, Theoretical Computer Science, 126 (1994), pp. 183–235.

[2] P. BELLINI, R. MATTOLINI, AND P. NESI, *Temporal logics for real-time system specification*, ACM Computing Surveys, 32 (2000), pp. 12–42.

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

[4] M. BROY, B. JONSSON, J.-P. KATOEN, M. LEUCKER, AND A. PRETSCHNER, eds., *Model-Based Testing of Reactive Systems: Advanced Lectures*, vol. 3472 of Lecture Notes in Computer Science, Springer, 2005.

[5] S. D. BRUDA, *Preorder relations*, in Broy et al. [4], pp. 117–149.

[6] ——, *Private communication*, 2014.

[7] S. D. BRUDA AND C. DAI, *A testing theory for real-time systems*, International Journal of Computers, 4 (2010), pp. 97–106.

[8] S. D. BRUDA AND Z. ZHANG, *Model checking is refinement: Computation tree logic is equivalent to failure trace testing*, Tech. Rep. 2009-002, Bishop's University, Department of Computer Science, aug 2009.

[9] ——, *Refinement is model checking: From failure trace tests to computation tree logic*, in Proceedings of the 13th IASTED International Conference on Software Engineering and Applications (SEA 09), Cambridge, MA, Nov. 2009.

[10] 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.

[11] 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.

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

[13] R. CLEAVELAND AND G. LÜTTGEN, *Model checking is refinement—Relating Büchi testing and linear-time temporal logic*, Tech. Rep. 2000-14, ICASE, Langley Research Center, Hampton, VA, Mar. 2000.

[14] C. DAI AND S. D. BRUDA, *A testing framework for real-time specifications*, in Proceedings of the 9th IASTED International Conference on Software Engineering and Applications (SEA 08), Orlando, Florida, Nov. 2008.

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

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

[17] R. W. FLOYD, *Assigning meanings to programs*, in Mathematical Aspects of Computer Science, J. T. Schwartz, ed., vol. 19 of Proceedings of Symposia in Applied Mathematics, American Mathematical Society, 1967, pp. 19–32.

[18] R. GERTH, D. PELED, M. Y. VARDI, AND P. WOLPER, *Simple on-the-fly automatic verification of linear temporal logic*, in Proceedings of the IFIP symposium on Protocol Specification, Testing and Verification (PSTV 95), Warsaw, Poland, 1995, pp. 3–18.

[19] C. A. R. HOARE, *An axiomatic basis for computer programming*, Communications of the ACM, 12 (1969), pp. 576–580 and 583.

[20] J.-P. KATOEN, *Labelled transition systems*, in Broy et al. [4], pp. 615–616.

[21] 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.

[22] H. R. LEWIS AND C. H. PAPADIMITRIOU, *Elements of the Theory of Computation*, Prentice-Hall, 2nd ed., 1998.

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

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

[25] J. P. QUEILLE AND J. SIFAKIS, *Fairness and related properties in transition systems — a temporal logic to deal with fairness*, Acta Informatica, 19 (1983), pp. 195–220.

[26] 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.

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

[28] 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.

[29] W. THOMAS, *Automata on infinite objects*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., vol. B, North Holland, 1990, pp. 133–191.

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

[31] M. VARDI AND P. WOLPER, *An automata-theoretic approach to automatic program verification*, in Proceedings of the First Annual Symposium on Logic in Computer Science (LICS 86), 1986, pp. 332–344.

[32] M. Y. VARDI AND P. WOLPER, *Reasoning about Infinite Computations*, Academic Press, 1994.