On the Equivalence between Computation Tree Logic and Failure Trace Testing

by

Sunita Singh

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
August 2017

Abstract

The two major systems of formal verifications are model checking and model-based testing. Model checking is based on some form of temporal logic for instance Linear Temporal Logic (LTL) or Computation Tree Logic (CTL, the focus of this thesis). Model-based testing is an algebraic technique which is based on some operational semantics of processes (such as traces and failures) and its associated preorders. The most fine-grained preorder is based on failure traces. A previous paper showed that CTL and failure trace testing are equivalent [5], in the sense that for any failure trace test there exist an equivalent CTL formula and other way around. However, the proof of the conversion from failure trace test to CTL is partially incorrect. We provide in this thesis a corrected proof. We also note that whenever a failure trace test contains cycles the CTL formula produced by the previous conversion algorithm is infinite in size. We develop a modified conversion algorithm which under certain, pretty general conditions produces finite formulae instead.

Acknowledgment

First of all, I would like to express my heartiest gratitude to my advisor Prof. Stefan D. Bruda for his guidance during my degree and related research, his timely assistance during the meetings and constant evaluations of my work. Without his support and valuable assistance, this work would not have been completed. Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Layachi Bentabet, Prof. Madjid Allili, and Prof. Juergen Dingel, for their insightful comments and questions. My appreciation also extends to the faculty of the CS Department, especially Prof. Nelly Khouzam and Prof. Lin Jensen for their valuable assistance during the course of my study.

Last but not the least, I'm thankful to God to give me strength, my family, my parents, and friends for their help, support, and encouragement throughout my graduate studies.

Contents

1 Introduction **1**

2 Preliminaries **6**

2.1 Temporal Logic and Model Checking 6

2.2 Labeled Transition Systems and Stable Failures 9

2.3 Failure Trace Testing 12

3 Previous Work **16**

3.1 A constructive Equivalence between LTS and Kripke Structures 17

3.2 From CTL formulae to Failure Trace Tests 20

4 CTL Is Equivalent to Failure Trace Testing **22**

4.1 From Failure Trace Tests to CTL Formulae 23

4.2 Converting Failure Trace Tests into Compact CTL Formulae 25

4.2.1 Example Generation of Compact CTL Formulae 32

5 Conclusions **36**

Bibliography **38**

Chapter 1

Introduction

The society today is heavily dependent on computing systems to assist us in almost every aspect of our daily life. From highly critical and complex systems to a small home appliance, everything has become digital. These systems are becoming more and more complex and massively encroaching on daily life. There are several control functions in cars, cell phones, planes, etc. which are based on embedded computing systems. Audio and video systems, medical devices, communications systems are containing huge amounts of software. Transport, highway, chemical plants, nuclear power plants, traffic control and alert systems are also increasingly using hardware and software systems, and errors in such software and hardware can have disastrous consequences. Therefore, the most challenging and important thing for the field of computer science is to ensure the correctness of hardware and software systems, and often no failure is acceptable.

Verification is the process of ensuring that a system has been built according to certain desired properties or specifications. Numerous verification methods have been proposed to provide assurance about the correctness of the systems. The traditional verification method is testing [15, 20] and still it is the most widely used technique. This non-formal

method of verification first generates input data to the system, then observes it and verifies whether the output data is according to the given input. However, there is no guarantee that this method will be fully protected from erroneous conclusions caused by the pseudorandom or nondeterministic behaviour of input data. Another shortcoming is that no complete coverage is possible, meaning that the method cannot cover all the possible situations, which means that testing can disprove correctness but rarely prove it. Deductive verification [18, 12] is a formal method of verification, also called program proving. It is the process of proving the correctness of the system mathematically using axioms and inference rules. An advantage of deductive verification is that it can be used for reasoning about infinite system states. The main disadvantage of this method is that is can never be fully automated; the needed manual intervention is very time consuming and requires highly skilled and experienced experts in logical reasoning.

Formal methods attempt to develop verification techniques that are sound, complete, and can be automated. We consider in this paper two such techniques: model-based testing [10, 22, 3] and model checking [7, 8, 17, 1]. In model-based testing the specification of a system is algebraic in nature, most often represented as a labeled transition system (LTS) or a finite automaton. Such a specification is generally an abstract description of the desired behaviour of the system under test. Test cases are generated in a systematic and a algorithmic manner from the specification and then we run all the test cases against the system under test and observe the result. The way tests are generated ensure that they are sound and complete. Model checking is a method for automatic (and algorithmic) verification of finite-state systems. Some temporal logic is used to produce and then reason about the specification of the system; indeed, the specification is a logical description of the system's behaviour. We then construct a Kripke structure (model) of the system under test and then we label all the states in the given model where the formula holds, which in turn determines whether the initial states of the model satisfy the specification formula.

If this is the case, then the system is deemed conforming to its specification.

Model checking is a complete and fully automatic verification technique but it also has some drawbacks. It is not compositional, so the user needs to model the whole system first, before verifying the correctness of the design. State explosion is another problem, for the number of states in the finite-state representation will increase exponentially. Thus the technique does not scale well to large systems. Model-based testing is compositional by its algebraic nature, therefore it can scale better to larger systems. However, there is no guarantee of completeness, since some of the test cases can take an infinite time to run. In model checking it is easy to provide partial specifications due to the logical nature of the specification, which allows us to only specify the properties of interest. In model-based testing the algebraic nature of the specification (LTS or finite automaton) requires the specification of more or less the whole system.

Various kinds of temporal logic are used to specify the system in model checking, which include CTL*, CTL and LTL. In this paper we will focus on CTL. Apart from this, we will also focus on probably the most powerful method of model-based testing which is failure trace testing. An interesting characteristic of failure trace testing is the existence of a simple testing scenario (consisting of so-called sequential tests) that is enough to evaluate the failure trace relation.

Some of the system's properties may be naturally specified by using temporal logic, whereas others can be readily specified using LTS or finite automata. Such a mixed specification could be given by somebody else, but generally for some components algebraic specifications are more convenient while for others logical specifications are more suitable. However, when this happens there is no global formal specification to verify the whole system. Thus, some part is specified by a temporal logic formula and could be model checked to insure the correctness, while for the other part of the same system the specification is given algebraically so that we can do model-based testing to ensure the

correctness of that part. Still, just by combining both of these results together there is no guarantee that the whole system is correct. The only way to ensure global correctness in this scenario is to convert one specification to the form of other, then apply the appropriate verification technique on the whole system. This paper contributes to precisely such a conversion. Earlier work [5, 23] developed an algorithmic method of converting CTL formulae into equivalent failure trace tests and the other way around. The conversion from tests to formulae however featured an incorrect proof, and also has the downside of producing potentially infinite formulae. We fix both these issues: we first correct the faulty proof, and then we show how to produce compact formulae equivalent to failure trace tests.

We believe that this effort opens the domain of combined algebraic and logical methods of formal system verification. The advantages of such a combined method stem not only from the possible combined specification as mentioned above, but also from the lack of compositionality of model checking (which can be avoided by switching to algebraic specification), from the lack of completeness of model-based testing (that can be avoided 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, being an "all or nothing" formalism).

This thesis continues as follows: In Chapter 2 we introduce model checking, model-based testing, temporal logic, and failure trace testing. In Chapter 3 we summarize the related previous work which includes the equivalence between label transition systems and Kripke structures together with a conversion method of an LTS into its equivalent Kripke structure, and the conversion of CTL formulae to failure trace tests. Finally we present our work namely, the algorithmic conversion from failure trace test to (compact) CTL formula in Chapter 4. Our conclusions are provided in Chapter 5. For the remainder of this thesis results proved elsewhere are introduced as propositions, while original

results are stated as theorems, and lemmata.

Chapter 2

Preliminaries

This section will cover temporal logic, model checking, LTS, stable failures, and failure trace testing. In several pieces of previous work the process algebra TLOTOS is used to specify the system under test as well as failure trace tests. Thus, here we will present this language as well.

Given the nature of our work, the preliminaries described in this thesis will be largely the same as the preliminaries used for the earlier work we fix and enhance [5, 23]. The content of this section will therefore be similar to the corresponding sections from the previous work.

2.1 Temporal Logic and Model Checking

Temporal logic formulae are used to describe a specification of the system. The system under test is modeled as a Kripke structure that should have identical properties as the system under test. The main objective of model checking is then to find the set of all states in the Kripke structure that satisfy the given logic formula. Some of the system's states are labeled as initial states, and then the system satisfies the specification as long as all the initial states satisfy the logic formula.

A *Kripke structure* [8] K over a set of elementary propositions AP 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 $L : S \rightarrow 2^{AP}$ is a function that labels each state with a set of atomic propositions which are true in that state. Generally we write $s \rightarrow t$ instead of $(s, t) \in \rightarrow$. The relation $\rightarrow$ is total, which means that for all $s \in S$ there exists $t \in S$ such that $s \rightarrow t$; "sink" states that have no outgoing transitions must feature a "self-loop" transition. A *path* π in a Kripke structure is a nonempty (finite or infinite) sequence of states $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$ such that $s_i \rightarrow s_{i+1}$ for all $i \geq 0$. State s_0 is the root and the path starts from that state. For all the paths starting from the root s_0 we can build a computation tree with nodes labelled with states and the root labelled as s_0, such that (s, t) is an edge in the tree iff $s \rightarrow t$ where $s, t \in S$.

Many variants of temporal logic have been proposed and are widely used. One family consists of CTL* [11, 8], CTL [8, 6] (computation tree logic) and LTL [16] (linear-time temporal logic). CTL and LTL are defined as the restricted subset of CTL*, where CTL* is the most general one. CTL is interpreted over computation trees and LTL is interpreted over runs or paths.

CTL* contains path quantifiers and temporal operators. The path quantifier A refers to all computation paths, and E refers to some of the computation paths. These two quantifiers are used to represent the branching structure of computation trees, given that states in the computation tree have several other successive states which leads to multiple paths starting from the same state. There are five temporal operators which are used to represent the individual path properties: X (requires that a property will hold in the next state of the path), F (requires that a property will hold at some state in future along the path), G (requires that a property will hold in every state along the path), U (requires that the first property will hold at every preceding state along the path until the second property becomes true and remains true afterward), and R (requires that the second property has to be true along a path up to the point where the first property becomes true, and so release

the second property from its obligation; if the first property never becomes true then the second property must remain true forever). These path properties can be preceded by the quantifiers A or E to become state properties.

There are two types of formulas in CTL* which are state formulae (that can be true or false in a specific state and use temporal operators preceded by quantifiers) and path formulae (that can be true or false along a specific path and so do not use path quantifiers). CTL is a restricted subset of CTL* where each of the temporal operators X, F, G, U, and R must be immediately preceded by the path quantifiers A, and E. Thus, we have the following syntax for CTL formulae, noting that all the CTL formulas are state formula:

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

where a is the atomic proposition ranging over AP and f, f_1, f_2 are state formulae.

The CTL semantics is defined with respect to Kripke structures. Basically we use the usual notation to specify that a state formula f is true in a state s of Kripke structure K: $K, s \models f$ means that in Kripke structure K, formula f is true at state s. If f is a path formula then $K, \pi \models f$ means that in Kripke structure K, formula f is true along the path π. We define the validation relation $\models$ inductively as follows (where f and g are state formulae):

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

6. $K, s \models \mathsf{E}\ f$ for some path formula f iff there exists a path π starting at s such that $K, s \models f$.

7. $K, s \models \mathsf{A}\ f$ for some path formula f iff $K, \pi \models f$ for all paths π starting at s.

We use π^i to denote the i-th state of a path π, with the starting state π^0 . The meaning of the relation $\models$ for path formula is the following:

1. $K, \pi \models \mathsf{X}f$ iff $K, \pi^1 \models f$ for any state formula f.

2. $K, \pi \models f\ \mathsf{U}\ g$ for state formula f and g iff there exists $j \geq 0$ such that $K, \pi^j \models g$ and $K, \pi^k \models g$ for all $k \geq j$, that means g is true at the state s_j and remain true afterwards for the later states, and for all $i < j$, $K, \pi^i \models f$, that means f is true from the initial state of π until the state s_i.

3. $K, \pi \models f\ \mathsf{R}\ g$ for any state formula f and g iff for all $j \geq 0$ if $K, \pi^i \not\models f$ and every $i < j$ then $K, \pi^j \models g$, that means f is not true from the initial state until the state s_i and g is true at the state s_j and the previous states.

2.2 Labeled Transition Systems and Stable Failures

The semantics of CTL is defined over Kripke structures, where each state is labeled with atomic propositions. In model-based testing the common models are the labeled transition system (LTS) and the finite automaton, where labels are associated with transitions instead of states. Usually an LTS describes the abstract behaviours of the system under test.

An LTS [13] is a tuple $M = (S, A, \rightarrow, s_0)$ where S is a countable, non empty set of states. $s_0 \in S$ is the initial state. A is countable set of labels which denote observable actions of a system. The internal action (which is not observable by the external environment) is denoted by τ such that $\tau \notin A$. The relation $\rightarrow \subseteq S \times (A \cup \{\tau\}) \times S$ is the transition

relation. The fact that $(p, a, q) \in \rightarrow$ is written as $p \xrightarrow{a} q$ and is interpreted as follows: there is a transition from state p to state q with label a, where the label representing any kind of visible or internal action. The set of states and its transitions can be considered global and if so then an LTS is completely defined by its initial state. We therefore blur whenever convenient the distinction between an LTS and an LTS state, calling them both "processes".

Generally, we consider a set T of relevant tests and set P of processes. In model-based testing [10, 3, 22] tests run parallel with the process (or system under test) and synchronize with it over observable actions. A run of a test t and a process p represent a possible sequence of states and actions of t and p running synchronously. Now we consider the set of exactly all the possible runs of p and t, where $p \in P$ and $t \in T$. The outcome of a run r may be true ($\top$) whenever a success state is encountered during that run, or false ($\bot$) whenever r does not contain a success state or r contains a state s such that s diverges (meaning that s engages in an infinite computation that does not produce any observable event) and its not preceded by successful state.

Given the nondeterministic nature of some tests and processes, we can have multiple runs for the given test t and process p (system under test), thus a set of outcomes is needed to provide the results of all the possible runs. Let $\text{Obs}(p, t)$ be the set of all the outcomes of the synchronized execution of process p and test t. We will have *may* and *must testing* depending on the degree of assurance that a process passes a test: A process p may pass the test t whenever there exists a run which leads to successful (that is, p may t iff $\top \in \text{Obs}(p, t)$), while p must pass the test t when all runs are successful (that is, p must t iff $\{\top\} = \text{Obs}(p, t)$).

To analyze the behaviour of the processes, we need to consider those sequence of events that can be observed at the interface of the process. There are a number of ways through which this behaviour can be analyzed. One aspect of process behaviour is that

the occurrence of certain events is in the right order. These observations are called *traces*. A trace is simply a record of events in the order they occur. Formally, traces are sequences of events over $A^{\checkmark} = A \cup \{\checkmark\}$, that might be possibly recorded. A represents the set of actions of a process, and the ✓ (tick) represents the termination. Events of the process can not occur after the termination, that means any termination event (✓) in a trace will only occur at the end. The set of all possible traces of a process p is denoted as traces(p).

A *path* (or run) π in an LTS is a sequence $p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots p_{k-1} \xrightarrow{a_k} p_k$ with $k \in \mathbb{N} \cup \{\infty\}$ such that $k = 0$, or $p_{i-1} \xrightarrow{a_i} p_i$ for all $0 < i \leq k$. We use $|\pi|$ to refer to k, which indicates the length of π. If $|\pi| \in \mathbb{N}$ then π is finite. The visible trace of π is defined as sequence trace$(\pi) = (a_i)_{0<i\leq|\pi|,a_i\neq\tau} \in A^*$. Internal actions are not recorded in traces, so we only consider the observable actions and transitions. The visible transitions are denoted by a specific notation $p \overset{w}{\Longrightarrow} p'$ which says that there is a sequence of transitions whose initial state is p, final state is p' and whose visible transitions form the sequence w. The notation $p \overset{w}{\Longrightarrow}$ is shorthand for $\exists p' : p \overset{w}{\Longrightarrow} p'$. We then define the traces of process p as traces$(p) = \{w : p \overset{w}{\Longrightarrow}\}$. The set of finite traces of process p is defined as Fin$(p) = \{tr \in \text{traces}(p) : |tr| \in \mathbb{N}\}$ where $|tr|$ refers to the length of trace tr. A process is said to be *stable* [19] when it does not make any internal progress (meaning that it has no internal outgoing actions) and it is defined as $p \downarrow = \neg(\exists p' \neq p : p \overset{\varepsilon}{\Longrightarrow} p')$. A stable process p always responds in an expected way to the offer of a set of actions $X \subseteq A^{\checkmark}$, such that there is at least one $a \in X$ that p can perform. When no such an action a is available then p will *refuse* the set X. We use the following notation: p ref X iff $\forall a \in X : \neg(\exists p' : p \overset{\varepsilon}{\Longrightarrow} p' \wedge p' \downarrow \wedge p' \xrightarrow{a})$.

To describe some possible behaviour of a process in terms of refusals we will record all the refusals together with the finite sequence of events that causes that refusal. The observation (w, X) that contains a refusal set X and the trace w that causes it is called a *stable failure* of p [19] whenever $\exists p^w : p \overset{w}{\Longrightarrow} p^w \wedge p^w \downarrow \wedge p^w$ ref $X\}$, meaning that p

performs the events in w and then reaches at the stable state, from where it refuses all the events in the set X. The stable failures of p are then described as $\mathrm{SF}(p) = \{(w, X) : \exists p^w : p \stackrel{w}{\Longrightarrow} p^w \wedge p^w \downarrow \wedge p^w \text{ ref } X\}$.

Depending on the level of interaction with processes, many preorder relations can be defined (like traces, stable failure, refusal etc.). In general, preorders are more convenient and more meaningful than equivalences in comparing specifications and their implementation: if two systems are in a preorder relation with each other, then one is the implementation of other. Thus such preorders can be interpreted as implementation relations in practice. The *stable failure preorder* $\sqsubseteq_{\mathrm{SF}}$ is defined as $p \sqsubseteq_{\mathrm{SF}} q$ iff $\mathrm{Fin}(p) \subseteq \mathrm{Fin}(q)$ and $\mathrm{SF}(p) \subseteq \mathrm{SF}(q)$ for any two processes p and q. That means that p implements q iff the set of finite traces of p is included in the finite traces of q and the stable failure of p are also included in the stable failure of q. Given the preorder $\sqsubseteq_{\mathrm{SF}}$ one can define the stable failure equivalence $\simeq_{\mathrm{SF}} : p \simeq_{\mathrm{SF}} q$ iff $p \sqsubseteq_{\mathrm{SF}} q$ and $q \sqsubseteq_{\mathrm{SF}} p$. The preorder $\sqsubseteq_{\mathrm{SF}}$ is considered one of the finest preorders [4].

2.3 Failure Trace Testing

In what follows we use the notation $\mathrm{init}(p) = \{a \in A : p \stackrel{a}{\Longrightarrow}\}$. A failure trace [14] f 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). Suppose 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 a failure trace of p if the following two conditions are observed:

- If $p_i \stackrel{\tau}{\longrightarrow}$ then $A_i = \emptyset$, when p_i is not a stable state then it will refuse 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.

Generally, we find the failure trace of any process p by taking a trace of p and then place refusal sets in it after the stable states.

In this paper, we will use the testing language TLOTOS [14, 2] which describes systems and tests succinctly. Let A be the countable set of visible actions, ranged over by a. The set of processes or tests are ranged over by t, t_1, and t_2, while T ranges over sets of tests. Then the syntax of TLOTOS is 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 $g \in A \cup \{\gamma, \theta, \tau\}$,
$$\frac{t_1 \xrightarrow{g} t_1'}{\Sigma(\{t_1\} \cup t) \xrightarrow{g} t_1'}$$

TLOTOS has the ability of detecting deadlock using θ (the deadlock detection label). The special action γ signals the successful termination of a test. Any process (or LTS) can be defined as a TLOTOS process not containing γ and θ. On the other hand, failure trace

tests are full TLOTOS processes, and thus may contain γ and θ. According to the parallel composition operator $\|_\theta$, a test runs in parallel with the system under test. 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 the processes and tests can be nondeterministic then we have a set $\Pi(p\|_\theta t)$ of possible runs of a process and a test. The success and failure of a test t and a process p under test depends on their outcome of a particular run $\pi \in \Pi(p\|_\theta t)$: whenever the last symbol in trace(π) is γ then the test t succeeds on process p ($\top$), otherwise it is not successful ($\bot$). All the possible outcomes of all the runs in $\Pi(p\|_\theta t)$ are denoted by $\text{Obs}(p, t)$. Then one can differentiate as usual the possibility and certainty of success for a test: 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 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 [14]. For a sequential test t the failure trace $\text{ftr}(t)$ is defined inductively as follows: $\text{ftr}(pass) = \emptyset$, $\text{ftr}(a; t') = a\ \text{ftr}(t')$, 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 inductively define the sequential test $\text{st}(f)$ as follows: $\text{st}(\emptyset) = pass$, $\text{st}(af) = a\ \text{st}(f)$, 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 $\text{st}(\text{ftr}(t)) = t$.

By the given bijection we can convert the failure trace preorder into a testing based preorder. 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 then define failure trace preorder $\sqsubseteq_{\text{FT}}$ as follows: $p \sqsubseteq_{\text{FT}}$ q iff $\text{ftr}(p) \subseteq \text{ftr}(q)$. This preorder is equivalent to the stable failure preorder.

Proposition 2.1 [14] *Let p be a process, t be a sequential test, and f be 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 *$t'$.*

We note that unlike other preorders, $\sqsubseteq_{\text{SF}}$ can be characterized in terms of may testing only; the must operator does not need to be considered any further.

Chapter 3

Previous Work

The body of research analyzing combined, logical and algebraic formal specification and verification is not very large. The only work directly relevant to our work is the effort of studying LTL and its relationship with Büchi automata [21].

Büchi automata were used as semantic basis for reasoning about combined logical and algebraic specification namely, LTL and the DeNicola and Hennessy testing preorders [9]. A unified semantic theory for heterogeneous system specifications featuring a mixture of LTS and LTL formulas was developed. First the Büchi must-preorder is described for a certain class of Büchi processes by means of trace inclusion. Then Büchi processes were constructed using a conversion of LTL formulae, such that the languages of the Büchi processes contain exactly all the traces that satisfy the respective formulae.

To the best of our knowledge the only investigation on the equivalence between CTL and algebraic specification is the work which is continued here [5, 23]. This work introduces a constructive conversion of LTS into equivalent Kripke structures, and then it constructs the conversion of failure trace tests into CTL formulae and the other way around. However, the earlier conversion of failure trace tests into equivalent CTL formulae [5] is partially incorrect.

In this chapter we will describe the aforementioned equivalence between LTS and

Kripke structures (see Proposition 3.1 below). We will also mention the conversion of CTL formulae into failure trace tests but only briefly (see Proposition 3.2 in section 3.2). This presentation is largely unchanged from the original report [5].

3.1 A constructive Equivalence between LTS and Kripke Structures

The LTS satisfaction operator is defined with the same formalism and in the same spirit as the CTL satisfaction operators over Kripke structures [5]. Basically, the actions available in an LTS state are propositions that hold in that state.

Definition 3.1 SATISFACTION FOR PROCESS [5]*: A process p satisfies $a \in A$, written by abuse of notation $p \models f$, iff $p \xrightarrow{a}$. That p satisfies some (general) CTL state formula is defined inductively as follows. Let f and g are some state formulae unless stated otherwise; then:*

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

As before, the notation π^i denotes the i-th state of a path π (with the first state being π^0). The definition of $\models$ for LTS path is:

1. *$\pi \models \mathsf{X}\, f$ iff $\pi^1 \models f$.*

2. $\pi \models f \ \mathsf{U}\ g$ *iff there exists* $j \geq 0$ *such that* $\pi^j \models g$ *and* $\pi^k \models g$ *for all* $k \geq j$*, and* $\pi^i \models f$ *for all* $i < j$*.*

3. $\pi \models f \ \mathsf{R}\ g$ *iff for all* $j \geq 0$*, if* $\pi^i \not\models f$ *for every* $i < j$ *then* $\pi^j \models g$*.*

We also introduce a weaker satisfaction operator for CTL. This operator is like the original, but it is defined over a set of states rather than a single state. By abuse of notation we also use $\models$ for this operator.

Definition 3.2 SATISFACTION OVER SETS OF STATES [5]: *Suppose a Kripke structure* $K = (S, S_0, R, L)$ *over AP. For some set* $Q \subseteq S$ *and some CTL state formula* f *is defined as follows;* $K, Q \models f$ *with* f *and* g *state formulae unless stated otherwise:*

1. $K, Q \models \top$ *is true and* $K, Q \models \bot$ *is false for any set* Q *in any Kripke structure* K*.*

2. $K, Q \models a$ *iff* $a \in L(s)$ *for some* $s \in Q$*,* $a \in \mathsf{AP}$*.*

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

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

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

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

7. $K, Q \models \mathsf{A}f$ *for some path formula* f *iff for some(any)* $s \in Q$ *it holds that* $K, \pi \models f$ *for all path* $\pi = s \to s_1 \to s_2 \to \cdots \to s_i$

Based on the above definitions the following equivalence relation between Kripke structures and LTS is introduced:

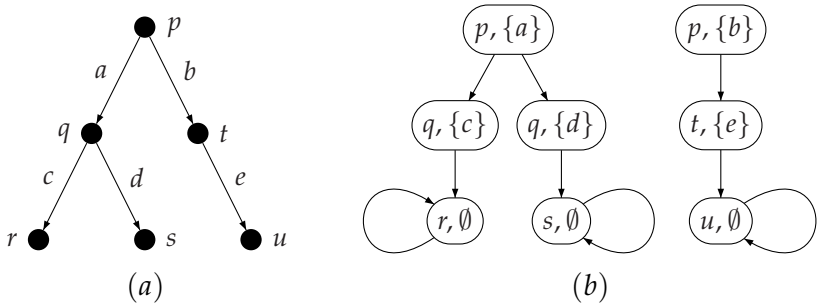
Definition 3.3 EQUIVALENCE BETWEEN KRIPKE STRUCTURES AND LTS [5]: *Given a Kripke structure K and a set of states Q of K, the pair K, Q is equivalent to a process p, written as $K, Q \simeq p$ (or $p \simeq K, Q$), iff for any CTL formula f $K, Q \models f$ iff $p \models f$.*

Proposition 3.1 [5] *There exist an algorithmic function ξ, which converts an LTS p into a Kripke structure K and a set of states Q such that $p \simeq (K, Q)$.*

Specifically, for any LTS $p = (S, A, \rightarrow, s_0)$, then we define its equivalent Kripke structure K as $K = (S', Q, R', L')$ where:

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

The process described in Proposition 3.1 is best described graphically. Refer for this purpose to Figure 3.1 [5]. Specifically, the function ξ converts the LTS given in Figure $3.1(a)$ into the equivalent Kripke structure shown in Figure $3.1(b)$. By combining each state with its corresponding actions in the LTS, we produce new states in its equivalent Kripke structure. Whenever an LTS state behaves differently by performing different actions, that state will split into multiple Kripke states (which implies between other things that the resulting Kripke structure may have multiple initial states). This generation of



Every state of the Kripke structure (b) is formed with the LTS(a) states and their corresponding outgoing action.

Figure 3.1: Illustration of the conversion from LTS (a) to its equivalent Kripke structure (b).

multiple initial Kripke states out of a single LTS state requires the weaker satisfaction operator defined in Definition 3.2. The reader is directed to the original proof [5] for further details.

3.2 From CTL formulae to Failure Trace Tests

In what follow $\mathcal{P}$ is the set of all processes, $\mathcal{T}$ is the set of all failure trace tests, and $\mathcal{F}$ is the set of all CTL formulae.

Proposition 3.2 [23] *There exist a function* $\mathbb{T} : \mathcal{F} \to \mathcal{T}$ *such that* $\xi(p) \models f$ *iff* p may $\mathbb{T}(f)$ *for any* $p \in \mathcal{P}$.

Brief outline of proof: The proof is done by structural induction over CTL formulae and the function $\mathbb{T}$ is also defined inductively at the same time. The basis is as follows:

1. $\mathbb{T}(\top) = \text{pass}$
2. $\mathbb{T}(\bot) = \text{stop}$
3. $\mathbb{T}(a) = a; \mathit{pass}$

The induction for non-temporal operators goes as follows:

1. $\mathbb{T}(\neg f) = \overline{\mathbb{T}(f)}$, where $\overline{\mathbb{T}(f)}$ is the complement of $\mathbb{T}(f)$
2. $\mathbb{T}(f_1 \vee f_2) = \mathbb{T}(f_1) \vee \mathbb{T}(f_2)$
3. $\mathbb{T}(f_1 \wedge f_2) = \mathbb{T}(f_1) \wedge \mathbb{T}(f_2)$

The definition of complement, conjunction, and disjunction of tests is given elsewhere [23].

The temporal operators are converted as follows:

1. $\mathbb{T}(\mathsf{EX}\, f) = \Sigma\{a; \mathbb{T}(f) : a \in A\}$
2. $\mathbb{T}(\mathsf{EF}\, f) = t'$ such that $t' = \mathbb{T}(f) \,\Box\, (\Sigma(a; t' : a \in A))$.
3. $\mathbb{T}(\mathsf{EG}\, f) = \mathbb{T}(f) \wedge (\mathbb{T}(\mathsf{EX}\, f') \,\Box\, \theta; \mathrm{pass})$, with $f' = f \wedge \mathsf{EX}\, f'$.
4. $\mathbb{T}(\mathsf{E}\, f_1\, \mathsf{U}\, f_2) = (\mathbb{T}(f_1) \wedge (\mathbb{T}(\mathsf{EX}\, f') \,\Box\, \theta; \mathrm{pass})) \,\Box\, \mathbf{i}; (\mathbb{T}(f_2) \wedge (\mathbb{T}(\mathsf{EX}\, f'') \,\Box\, \theta; \mathrm{pass}))$, with $f' = f_1 \wedge \mathsf{EX}\, f'$ and $f'' = f_2 \wedge \mathsf{EX}\, f''$.

For the more detailed information and for the full proof of the conversion of CTL formulae to failure trace tests, the reader is invited to follow the original proof [23]. □

Chapter 4

CTL Is Equivalent to Failure Trace Testing

The original proof of equivalence between CTL and failure trace testing [5] also went the other way around showing how failure trace tests can be converted into equivalent CTL formulae. However, there are two problems with this result: First, one part of the proof (together with the resulting conversion function) turns out to be incorrect. Secondly, the conversion does not take advantage of all but one temporal operator and so the resulting formulae are infinite whenever the test being converted contains cycles. This section corrects these two shortcomings and is thus the main contribution of our work.

First, we fix the original proof mentioned above [5] and so we effectively prove the following result:

Theorem 4.1 *For some $t \in \mathcal{T}$ and $f \in \mathcal{F}$, whenever p* may *t if and only if $\xi(p) \models f$ 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. One direction is given by Proposition 3.2 (see Section 3.2). The other direction is given by Lemma 4.2 (see Section 4.1 below), which was stated elsewhere [5] with an

incorrect proof (which we fix here). As before, the algorithmic nature of the conversion is shown implicitly in the proof of these two results. □

Afterward we show how to produce a compact CTL formula equivalent to a given failure trace test (see Section 4.2).

4.1 From Failure Trace Tests to CTL Formulae

Lemma 4.2 *There exists a function* $\mathbb{F} : \mathcal{T} \to \mathcal{F}$ *such that* p may t *if and only if* $\xi(p) \models \mathbb{F}(t)$ *for any* $p \in \mathcal{P}$.

Proof. The proof is by structural induction over tests. In the process we construct the function $\mathbb{F}$ which is also defined inductively.

We establish the basis as follows: $\mathbb{F}(\text{pass}) = \top$ and $\mathbb{F}(\text{stop}) = \bot$. Clearly any process passes pass and any Kripke structure satisfies $\top$, so it is immediate that p may pass iff $\xi(p) \models \mathbb{F}(\text{pass})$. Similarly, no kripke structure satisfies $\bot$ and no process passes stop.

We put $\mathbb{F}(\mathbf{i}; t) = \mathbb{F}(t)$: by definition, an internal action is not seen by the external environment of the system under test. Then $\mathbb{F}(a; t) = a \wedge \mathsf{EX}\, \mathbb{F}(t)$. p may $(a; t)$ iff p may a (p performed a and became p') and p' may t for some $p \xrightarrow{a} p'$. Now, p may a iff $\xi(p) \models a$. p' may t iff $\xi(p') \models \mathbb{F}(t)$ according to the inductive hypothesis. By Proposition 3.1 in Section 3.1, while converting any process p to an equivalent Kripke structure $\xi(p)$, we combine original states together with their corresponding outgoing actions to produce new states. Thus once we are in a state that satisfies a, all the next states of that state correspond to the states following p after executing a. Therefore, $\mathsf{EX}(\mathbb{F}(t))$ is satisfied in those states where t must succeed. For example, as we have seen in Figure 3.1, when two different LTS actions are performed by an initial state (p) then that initial state split into two initial states in its equivalent $\xi(p)$ which are $(p, \{a\})$, and $(p, \{b\})$. In the given $\xi(p)$ in Figure 3.1 (b) the state which satisfies the a and next state to that state of $\xi(p)$ is only

q. So $\xi(p') \models t$ which is equivalent to p' may t (here we write p' in place of q). Therefore $\xi(p) \models a \wedge \mathsf{EX}\mathbb{F}(t)$ that means p satisfies the property a and the next state to p which is p' only, which satisfies the formula $\mathbb{F}(t)$. Thus p may $(a;t)$ iff $\xi(p) \models a \wedge \mathsf{EX}\mathbb{F}(t)$.

We note that $\Box$ is just a syntactic sugar, for indeed $t_1 \Box t_2$ is completely equivalent with $\Sigma\{t_1, t_2\}$. We put $\mathbb{F}(\Sigma T) = \bigvee \mathbb{F}(t) : t \in T$. p may ΣT iff p may t for at least one $t \in T$ iff $\xi(p) \models \mathbb{F}(t)$ for at least one $t \in T$(by induction hypothesis) iff $\xi(p) \models \bigvee \mathbb{F}(t) : t \in T$.

We note in passing that so far the proof is the same as the previous one [5]. The θ case has been however stated previously [5] in an incorrect manner as follows:

> 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 $\mathbb{F}(\Sigma T \Box \theta; t) = (\mathbb{F}(t_1) \vee \mathbb{F}(t_2) \vee \cdots \vee \mathbb{F}(t_n)) \vee (\neg(\mathbb{F}(t_1) \wedge \mathbb{F}(t_2) \wedge \cdots \wedge \mathbb{F}(t_n)) \wedge \mathbb{F}(t))$.

This construction does not capture the fact that θ only operates at the top level. Indeed, $\mathbb{F}(\Sigma T \Box \theta; t)$ considers the θ branch if and only if the *tests* $t_1, \ldots, t_n$ all fail. However, the θ branch should only be considered based on the *initial actions* of $t_1, \ldots, t_n$, regardless of the outcomes of these tests. For example let $t_1 \Box \theta; t$ be applied to a process p such that $p \stackrel{a}{\Longrightarrow}$, $a \in \mathrm{init}(t_1)$. Suppose further that t_1 fails on p but t succeeds on p. Given that $a \in \mathrm{init}(t_1)$ is available the θ branch is forbidden and thus the test $t_1 \Box \theta; t$ fails on p. The formula $\mathbb{F}(t_1 \Box \theta; t)$ however is true! Indeed, under the assumption that $\mathbb{F}(t_1)$ is false and $\mathbb{F}(t)$ is true we have that $\neg\mathbb{F}(t_1) \wedge \mathbb{F}(t)$ is true and so $\mathbb{F}(t_1) \vee (\neg\mathbb{F}(t_1) \wedge \mathbb{F}(t)) = \mathbb{F}(t_1 \Box \theta; t)$ is true as well.

We now provide a correct proof for the θ case. It continue to be the case that when θ does not participate in a choice then it exactly behaves like an internal action $\mathbf{i}$, so we

can assume without loss of generality that θ appears only in choice construct. As in the previous proof we consider that every choice contains at most one top-level θ, for indeed $\theta; t_1 \Box \theta; t_2$ is equivalent with $\theta; (t_1 \Box t_2)$. We then have $\mathbb{F}(t_1 \Box \theta; t) = ((\bigvee init(t_1)) \wedge \mathbb{F}(t_1)) \vee (\neg(\bigvee init(t_1)) \wedge \mathbb{F}(t))$.

According to TLOTOS definition of $\|_\theta$ if common action is available for both p and t then the deadlock detection action θ will not play any role. In other words, whenever $p \stackrel{a}{\Longrightarrow}$ such that $a \in init(t_1)$ we have p may $t_1 \Box \theta; t$ iff p may t_1. We further note that $p \stackrel{a}{\Longrightarrow}$ is equivalent to $\xi(p) \models a$ and so given the inductive hypothesis (that p' may t_1 iff $\xi(p') \models \mathbb{F}(t_1)$ for any process p') we concluded that:

$$(p \text{ may } t_1 \Box \theta; t) \wedge (p \stackrel{a}{\Longrightarrow} \wedge a \in init(t_1)) \; iff \xi(p) \models (\bigvee init(t_1)) \wedge \mathbb{F}(t_1) \tag{4.1}$$

whenever it is not the case that $p \stackrel{a}{\Longrightarrow}$ and $a \in init(t_1)$ (equivalent to $\xi(p) \not\models \bigvee init(t_1)$), then the deadlock detection transition θ of $t_1 \Box \theta; t$ will fire and then the test will succeed iff t succeeds. Given once more the inductive hypothesis that p' may t iff $\xi(p') \models \mathbb{F}(t)$ for any process p' we have:

$$(p \text{ may } t_1 \Box \theta; t) \wedge \neg(p \stackrel{a}{\Longrightarrow} \wedge a \in init(t_1)) \; iff \xi(p) \models \neg(\bigvee init(t_1)) \wedge \mathbb{F}(t) \tag{4.2}$$

Taking the disjunction of both sides of Relations 4.1 and 4.2 we obtain the desired property:

$$(p \text{ may } t_1 \Box \theta; t) \; iff \xi(p) \models (\bigvee init(t_1)) \wedge \mathbb{F}(t_1) \vee \xi(p) \models \neg(\bigvee init(t_1)) \wedge \mathbb{F}(t)$$

the induction is thus complete now. □

4.2 Converting Failure Trace Tests into Compact CTL Formulae

Whenever all the runs of a test are finite then the conversion shown in Lemma 4.2 will produce a reasonable CTL formula. That formula is however not in its simplest form. In

particular, the conversion algorithm follows the run of the test step by step, so whenever the test has one or more cycles (and thus features potentially infinite runs) the resulting formula has an infinite length. We now show that more compact formulae can be obtained and in particular finite formulae can be derived out of tests with potentially infinite runs.

Theorem 4.3 *There exists an algorithmic function, denoted by abuse of notation* $\mathbb{F} : \mathcal{T} \to \mathcal{F}$ *such that* p may t *if and only if* $\xi(p) \models \mathbb{F}(t)$ *for any* $p \in \mathcal{P}$ *and* $\mathbb{F}(t)$ *is finite for any test* t *provided that no loop in* t *features duplicate actions; in other words, for any loop state* t_0 *from* t *such that* $t_0 \stackrel{a_1}{\Longrightarrow} t_1 \stackrel{a_2}{\Longrightarrow} t_2 \stackrel{a_3}{\Longrightarrow} \cdots \stackrel{a_n}{\Longrightarrow} t_n = t_0$ *we have* $a_1 \neq a_2 \neq \cdots \neq a_n$.

Proof. It is enough to show how to produce a finite formula starting from a general "loop" test. Such a conversion can be then applied to all the loops one by one, relying on the original conversion function from Lemma 4.2 for the rest of the test. Given the reliance on the mentioned lemma we obtain overall an inductive construction. Therefore nested loops in particular will be converted inductively (that is, from the innermost loop to the outermost loop).

Thus to complete the proof it is enough to show how to obtain an equivalent, finite CTL formula for the following, general form of a loop test:

$$t = a_0; (t_0 \,\square\, a_1; (t_1 \,\square\, \cdots a_{n-1}; (t_{n-1} \,\square\, t) \cdots))$$

The loop itself consists of the actions $a_0, \ldots, a_{n-1}$. Each such an action a_i has the "exit" test t_i as an alternative. There is no assumption about the particular form of t_i.

Given the intended use of our function, this proof will be done within the inductive assumptions of the proof of Lemma 4.2. We will therefore consider that the formulae $\mathbb{F}(a_i)$ and $\mathbb{F}(t_i)$ exist and are finite, $0 \leq i < n$.

We have:

$$\mathbb{F}(t) = \mathsf{E}\left(\bigvee_{i=0}^{n-1} C_i\right) \;\mathsf{U}\; \left(\bigvee_{i=0}^{n-1} E_i\right)$$

where C_i represents the cycle in its various stages such that

$$C_i = \mathsf{EG}(\mathbb{F}(a_i) \wedge \mathsf{EX}(\mathbb{F}(a_{(i+1) \bmod n}) \wedge \mathsf{EX} \cdots \wedge \mathsf{EX}(\mathbb{F}(a_{(i+n-1) \bmod n})) \cdots))$$

and each E_i represents one possible exit from the cycle and so

$$E_i = \mathbb{F}(a_i) \wedge \mathsf{EX}\, \mathbb{F}(t_i)$$

Intuitively, each C_i corresponds to a_i as being available in the test loop, followed by all the rest of the loop in the correct order. It therefore models the decision of the test to perform a_i and remain in the loop. Whenever some a_i is available ("true") then the corresponding C_i is true and so the disjunction of the formulae C_i will keep being true as long as we stay in the loop. By contrast, each E_i corresponds to the action a_i being available in the test loop, followed by the exit from the loop using the test t_i. The formula E_i will become true whenever the test is in the right place (offering a_i) and the test t_i succeeds after a_i is performed. Such an event releases the loop formula from its obligations (following the semantics of the U operator), so such a path can be taken by the test and will be successful.

The formula above assumes that neither the actions in the cycle nor the top-level actions of the exit tests are θ. We introduce the deadlock detection action along the following cases, with k an arbitrary value, $0 \leq k < n$: θ may appear in the loop as a_k but not on top level of the alternate exit test $t_{k-1 \bmod n}$ (Case 1), on the top level of the test $t_{k-1 \bmod n}$ but not as alternate a_k (Case 2), or both as a_k and on the top level of the alternate $t_{k-1 \bmod n}$ (Case 3). Given that θ only affects the top level of the choice in which it participates, these cases are exhaustive.

Case 1: if any $a_k = \theta$ and $\theta \notin \mathrm{init}(t_{k-1 \bmod n})$ then we replace all occurrences of $\mathbb{F}(a_k)$ in $\mathbb{F}(t)$ with $\neg(\bigvee_{b \in \mathrm{init}(t_{k-1 \bmod n})} \mathbb{F}(b))$ in conjunction with $\bigvee_{b \in \mathrm{init}(t_k) \setminus \{\theta\}} \mathbb{F}(b)$ for the "exit" formulae and with $\mathbb{F}(a_{k+1 \bmod n})$ for the "cycle" formulae). Therefore $C_i = \mathsf{EG}(\mathbb{F}(a_i) \wedge$

$\mathsf{EX}(\cdots \wedge \mathsf{EX}(\neg(\bigvee_{b\in \mathrm{init}(t_{k-1 \bmod n})} \mathbb{F}(b)) \wedge \mathbb{F}(a_{k+1 \bmod n}) \wedge \mathsf{EX} \cdots \wedge \mathsf{EX}(\mathbb{F}(a_{(i+n-1) \bmod n})) \cdots)))$ and $E_k = \neg(\bigvee_{b\in \mathrm{init}(t_{k-1 \bmod n})} \mathbb{F}(b)) \wedge \bigvee_{b\in \mathrm{init}(t_k)\setminus\{\theta\}} \mathbb{F}(b) \wedge \mathsf{EX}(\mathbb{F}(t_k))$.

Case 2: Whenever $\theta \in \mathrm{init}(t_{k-1 \bmod n})$ and $a_k \neq \theta$ then we will change the exit formula $E_{k-1 \bmod n}$ which in this case will contain two components. If any action in $\mathrm{init}(t_{k-1 \bmod n})$ is available then such an action can be taken, so a first component is $\mathbb{F}(a_{k-1 \bmod n}) \wedge \mathsf{EX}\,(\bigvee_{b\in \mathrm{init}(t_{k-1 \bmod n})\setminus\{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_{k-1 \bmod n})$. Note that any θ top-level branch in $t_{k-1 \bmod n}$ is invalidated (since some action $b \in \mathrm{init}(t_{k-1 \bmod n})$ is available). The top-level θ branch of $t_{k-1 \bmod n}$ can be taken only if no action from $\mathrm{init}(t_{k-1 \bmod n}) \cup \{a_k\}$ is available, so the second variant is $\mathbb{F}(a_{k-1 \bmod n}) \wedge \mathsf{EX}\,\neg\mathbb{F}(a_{(k)}) \wedge \neg(\bigvee_{b\in \mathrm{init}(t_{k-1 \bmod n})\setminus\{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_{k-1 \bmod n}(\theta))$, where $t_{k-1 \bmod n} = t' \,\Box\, \theta; t_{k-1 \bmod n}(\theta)$ for some test t' (recall that we can assume without loss of generality that there exists a single top-level θ branch in $t_{k-1 \bmod n}$).

By taking the disjunction of both the above variants we have $E_{k-1 \bmod n} = \mathbb{F}(a_{k-1 \bmod n}) \wedge \mathsf{EX}(\bigvee_{b\in \mathrm{init}(t_{k-1 \bmod n})\setminus\{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_{k-1 \bmod n}) \vee \neg\mathbb{F}(a_{(k)}) \wedge \neg(\bigvee_{b\in \mathrm{init}(t_{k-1 \bmod n})\setminus\{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_{k-1 \bmod n}(\theta))$.

Case 3: If $a_k = \theta$ and $\theta \in \mathrm{init}(t_{k-1 \bmod n})$, then we must modify the cycle as well as the exit test. Let $B = \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\}$.

Whenever an action from B is available the cycle cannot continue, so we replace in C all occurrences of a_k with $\neg(\bigvee_{b\in B} \mathbb{F}(b)) \wedge \bigvee_{b\in\{a_{k+1 \bmod n}\}\cup \mathrm{init}(t_k)\cup\setminus\{\theta\}} \mathbb{F}(b)$ so that $C_i = \mathsf{EG}(\mathbb{F}(a_i) \wedge \mathsf{EX}(\cdots \wedge \mathsf{EX}(\mathbb{F}(\neg(\bigvee_{b\in \mathrm{init}(t_{k-1 \bmod n})\setminus\{\theta\}} \mathbb{F}(b))) \wedge \bigvee_{b\in\{a_{k+1 \bmod n}\}\cup \mathrm{init}(t_k)\setminus\{\theta\}} \mathbb{F}(b) \wedge \mathsf{EX} \cdots \wedge \mathsf{EX}(\mathbb{F}(a_{(i+n-1) \bmod n})) \cdots)))$.

Similarly, when actions from B are available the non-θ component of the exit test is applicable, while the θ branch can only be taken when no action from B is offered. Therefore we have $E_{k-1 \bmod n} = \mathbb{F}(a_{k-1 \bmod n}) \wedge \mathsf{EX}\ \bigvee_{b\in \mathrm{init}(t_{k-1 \bmod n})\setminus\{\theta\}} \mathbb{F}(b) \wedge \mathbb{F}(t_{k-1 \bmod n}) \vee \neg(\bigvee_{b\in \mathrm{init}(t_{k-1 \bmod n})\setminus\{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_{k-1 \bmod n}(\theta))$. As before, $t_{k-1 \bmod n}(\theta)$ is the θ-branch of $t_{k-1 \bmod n}$ that is, $t_{k-1 \bmod n} = t' \,\Box\, \theta; t_{k-1 \bmod n}(\theta)$ for some test t').

Finally, recall that originally $E_k = \mathbb{F}(a_k) \wedge \mathsf{EX}\ \mathbb{F}(t_k)$. Now however $a_k = \theta$ and so we must apply to E_k the same process that we repeatedly performed earlier. That is, we replace $\mathbb{F}(a_k)$ with $\neg(\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b))$. In addition, θ does not consume any input by definition, so the EX construction disappear. In all we have
$E_k = \neg(\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_k)$.

We now prove that the construction described above is correct. We focus first on the initial, θ-less formula.

If the common actions are available for both p, and t then $p \stackrel{a_i}{\Longrightarrow} p_1 \wedge p_1 \stackrel{a_{i+1}}{\Longrightarrow} \wedge \cdots \wedge p_{n-1} \stackrel{a_{i+n-1}}{\Longrightarrow} p$, which shows that process p performs some actions in the cycle. We further notice that these are equivalent to $\xi(p) \models a_i$ and $\xi(p_1) \models a_{i+1} \wedge \cdots \wedge \xi(p_{n-1}) \models a_{i+n-1}$, respectively. Therefore $p \stackrel{a_i}{\Longrightarrow} p_1 \wedge p_1 \stackrel{a_{i+1}}{\Longrightarrow} \wedge \cdots \wedge p_{n-1} \stackrel{a_{i+n-1}}{\Longrightarrow} p$ iff $\xi(p) \models \mathsf{EG}(\mathbb{F}(a_i) \wedge \mathsf{EX}(\mathbb{F}(a_{(i+1) \bmod n}) \wedge \mathsf{EX} \cdots \wedge \mathsf{EX}(\mathbb{F}(a_{(i+n-1) \bmod n})) \cdots))$. That is,

$$p \stackrel{a_i}{\Longrightarrow} p_1 \wedge p_1 \stackrel{a_{i+1}}{\Longrightarrow} \wedge \cdots \wedge p_{n-1} \stackrel{a_{i+n-1}}{\Longrightarrow} p \text{ iff } \xi(p) \models C_i \tag{4.3}$$

We exit from the cycle as follows: When $p \stackrel{a_i}{\Longrightarrow} p' \stackrel{a_{i+1}}{\not\Longrightarrow}$ then the process p must take the test t_i after performing a_i and pass it. If however the action a_{i+1} is available in the cycle as well as in the test t_i, then it depends on the process p whether it will continue in the cycle or will take the test t_i. That means p may take t_i and pass the test or it may decide to continue in the cycle. Eventually however the process must take one of the exit tests. Given the nature of may-testing one successful path is enough for p to pass t.

Formally, we note that $p \stackrel{a_i}{\Longrightarrow} p' \wedge p'$may t_i is equivalent to $\xi(p) \models \mathbb{F}(a_i) \wedge \mathsf{EX}(\mathbb{F}(t_i))$ and so given the inductive hypothesis (that p' may t_i iff $\xi(p') \models \mathbb{F}(t_i)$ for any process p') we concluded that:

$$(p \stackrel{a_i}{\Longrightarrow} p' \wedge p' \text{ may } t_i) \text{ iff } \xi(p) \models (\mathbb{F}(a_i) \wedge \mathsf{EX}(\mathbb{F}(t_i))) = E_i \tag{4.4}$$

Taking the disjunction of Relations (4.4) over all $0 \leq i < n$ we have

$$(p \stackrel{a_i}{\Longrightarrow} p' \wedge p' \text{ may } \Sigma_{i=0}^{n-1} t_i) \text{ iff } \xi(p) \models \bigvee_{i=0}^{n-1} E_i \tag{4.5}$$

The correctness of $\mathbb{F}(t)$ is then the direct consequences of Relations (4.3) and (4.5): We can stay in the cycle as long as one C_i remains true (Relation (4.3)), and we can exit at any time using the appropriate exit test (Relation (4.5)).

Now, we consider the possible deadlock detection action as introduced in the three cases above. We have:

Case 1: Let $a_k = \theta$, $\theta \notin \text{init}(t_{k-1 \bmod n})$ and suppose that $p \|_\theta t$ runs along such that they reach the point $p' \|_\theta t' \stackrel{a_{k-1 \bmod n}}{\longrightarrow} p'' \|_\theta t''$. Let $b \in \text{init}(t_{k-1 \bmod n})$. If $p'' \|_\theta t'' \stackrel{b}{\longrightarrow}$ then the run must exit the cycle according to the definition of $\|_\theta$. At the same time C_k is false because the disjunction $\bigvee_{b \text{init}(t_{k-1 \bmod n})} \mathbb{F}(b)$ is true and no other C_i is true, so C is false and therefore the only way for $\mathbb{F}(t)$ to be true is for E_k to be true. The two, testing and logic scenarios are clearly equivalent. On the other hand, if $p'' \|_\theta t'' \stackrel{b}{\not\longrightarrow}$ for any $b \in \text{init}(t_{k-1 \bmod n})$, then the test must take the θ branch. At the same time C_k is true and so is C, whereas E_k is false (so the formula must "stay in the cycle"), again equivalent to the test scenario.

Case 2: Now $\theta \in \text{init}(t_{k-1 \bmod n})$ and $a_k \neq \theta$. The way the process and the test perform a_k and remain in the cycle is handled by the general case so we are only considering the exit test $t_{k-1 \bmod n}$. The only supplementary consequence of a_k being available is that any θ branch in $t_{k-1 \bmod n}$ is disallowed, which is still about the exit test rather than the cycle.

There are two possible successful runs that involve the exit test $t_{k-1 \bmod n}$: Either $p \stackrel{a_{k-1 \bmod n}}{\Longrightarrow} p' \wedge \exists b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\} : p' \stackrel{b}{\Longrightarrow} \wedge p'$ may $t_{k-1 \bmod n}$, or $p \stackrel{a_{k-1 \bmod n}}{\Longrightarrow} p' \wedge \neg(\exists b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\} : p' \stackrel{b}{\Longrightarrow}) \wedge p' \stackrel{a_k}{\not\Longrightarrow} \wedge p'$ may $t_{k-1 \bmod n}(\theta)$. The first case corresponds to a common action b being available to both the process and the test (case in which the θ branch of $t_{k-1 \bmod n}$ is forbidden by the semantics of p' may $t_{k-1 \bmod n}$), while

the second case requires that the θ branch of the test is taken whenever no other action is available.

Given the inductive hypothesis (that p' may t_i iff $\xi(p') \models \mathbb{F}(t_i)$ for any process p') we have

$$\begin{aligned} p \stackrel{a_{k-1 \bmod n}}{\Longrightarrow} p' \wedge \exists b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\} : p' \stackrel{b}{\Longrightarrow} \wedge p' \text{ may } t_{k-1 \bmod n} \text{ iff} \\ \xi(p) \models \mathbb{F}(a_{k-1 \bmod n}) \wedge \mathsf{EX} \bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b) \wedge \mathbb{F}(t_{k-1 \bmod n}) \end{aligned} \tag{4.6}$$

$$\begin{aligned} p \stackrel{a_{k-1 \bmod n}}{\Longrightarrow} p' \wedge \neg(\exists b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\} : p' \stackrel{b}{\Longrightarrow}) \wedge p' \stackrel{a_k}{\not\Longrightarrow} \wedge p' \text{ may } t_{k-1 \bmod n}(\theta) \text{ iff} \\ \xi(p) \models \mathbb{F}(a_{k-1 \bmod n}) \wedge \mathsf{EX}\, \neg\mathbb{F}(a_{(k)}) \wedge \neg(\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_{k-1 \bmod n}(\theta)) \end{aligned} \tag{4.7}$$

The conjunction of Relations (4.6) and (4.7) establish this case. Indeed, the left hand sides of the two relations are the only two ways to have a successful run involving $t_{k-1 \bmod n}$ (as argued above), and $\Big(\mathbb{F}(a_{k-1 \bmod n}) \wedge \mathsf{EX} \bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b) \wedge \mathbb{F}(t_{k-1 \bmod n})\Big) \vee$ $\Big(\mathbb{F}(a_{k-1 \bmod n}) \wedge \mathsf{EX}\, \neg\mathbb{F}(a_{(k)}) \wedge \neg(\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_{k-1 \bmod n}(\theta))\Big) =$ $E_{k-1 \bmod n}$.

Case 3: Let now $a_k = \theta$ and $\theta \in \text{init}(t_{k-1 \bmod n})$. Suppose that the process under test is inside the cycle and has reached a state p such that $p \stackrel{a_{k-1 \bmod n}}{\Longrightarrow} p'$, meaning that p' is ready to either continue within the cycle or pass $t_{k-1 \bmod n}$.

Suppose first that $p' \stackrel{b}{\Longrightarrow}$ for some $b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}$. Then (a) p' cannot continue in the cycle, which is equivalent to C_k being false (since no C_i, $i \neq k$ can be true), and so (b) p' must pass $t_{k-1 \bmod n}$, which is equivalent to $\xi(p') \models \bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b) \wedge$ $\mathbb{F}(t_{k-1 \bmod n})$. That C_k is false happens because $\neg(\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b))$ is false. Note incidentally that the θ branch of $t_{k-1 \bmod n}$ is forbidden, but this is guaranteed by the semantics of p' passing $t_{k-1 \bmod n}$ (and therefore by the semantics of $\xi(p') \models \mathbb{F}(t_{k-1 \bmod n})$ by inductive hypothesis).

Suppose now that $p' \stackrel{b}{\not\Longrightarrow}$ for any $b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}$. Then the only possible continuations are (a) p' remaining in the cycle which is equivalent to C_k being true (ensured by $\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)$) being false), or (b) p' taking the θ branch of $t_{k-1 \bmod n}$, which is equivalent to $\neg(\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_k(\theta))$ by the fact that $\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)$ is false and the inductive hypothesis, , or (c) p' taking the test t_k (which falls just after $a_k = \theta$ and so it is an alternative in the deadlock detection branch), which is equivalent to E_k being true, ensured by $\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)$ being false and $\mathbb{F}(t_k)$ being true iff p' passes t_k by inductive hypothesis.

Once more, taking the disjunction of the two alternatives above establishes this case. □

4.2.1 Example Generation of Compact CTL Formulae

In this section we will illustrate the conversion of failure trace test into CTL formulae. For this purpose consider the following simple vending machines, also shown graphically in Figure 4.1(a, b):

$$\begin{aligned} P_1 &= \text{coin}; ((\text{coffee} \,\Box\, \text{water}) \,\Box\, \text{bang}; (\text{tea} \,\Box\, P_1)) \\ P_2 &= \text{coin}; ((\text{coffee} \,\Box\, \text{water}) \,\Box\, \text{bang}; (\text{coffee} \,\Box\, P_2)) \end{aligned}$$

First machine dispenses either coffee or water after accepting a coin. It can also dispense tea, but only after the customer hits it. The second machine still dispenses coffee or water, and it does not change its selection much upon hitting; in such a case coffee is dispensed.

The Kripke structures equivalent to the two machines and constructed according to Proposition 3.1 are shown in Figure 4.1(c, d), respectively.

Consider now the following test:

$$t_1 = \text{coin}; (\text{coffee}; \text{pass} \,\Box\, \theta; \text{water}; \text{pass} \,\Box\, \text{bang}; (\text{tea}; \text{pass} \,\Box\, t_1))$$

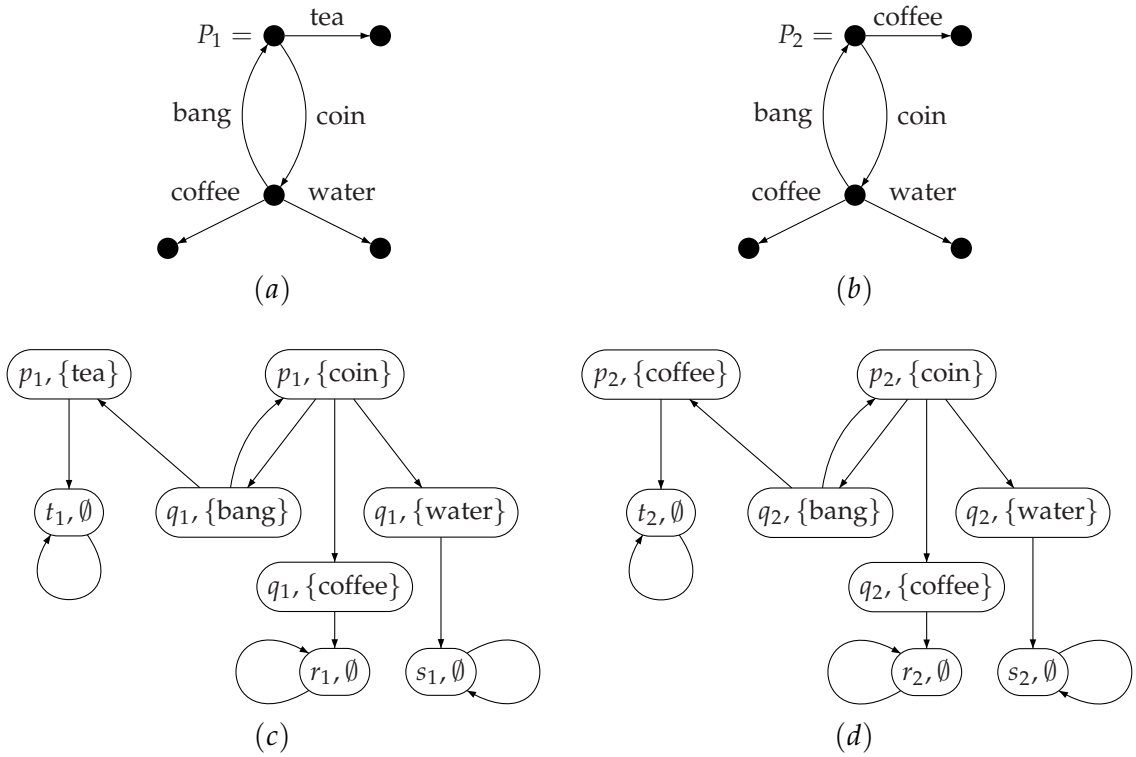


Figure 4.1: Two vending machines P_1 and P_2 (a, b) and their equivalent Kripke structures $\xi(P_1)$ and $\xi(P_2)$ (c, d).

Using Theorem 4.3 (and thus implicitly Lemma 4.2) we can convert this test into the following CTL formula $\mathbb{F}(t_1) = \mathsf{E}(\mathsf{EG}(\text{coin} \wedge \mathsf{EX}(\text{bang})) \vee (\text{bang} \wedge \mathsf{EX}(\text{coin})))\mathsf{U}(\text{coin} \wedge \mathsf{EX}(\top) \wedge \text{coffee} \wedge \mathsf{EX}(\top) \vee \neg\text{bang} \wedge \neg\text{coffee} \wedge (\text{water} \wedge \mathsf{EX}(\top))) \vee (\text{bang} \wedge \mathsf{EX}(\text{tea} \wedge \mathsf{EX}(\top)))$. By eliminating the true sub-formulae, we obtain the desired logically equivalent formula:

$$\begin{aligned}\mathbb{F}(t_1) \;=\; & \mathsf{E}(\mathsf{EG}\ (\text{coin} \wedge \mathsf{EX}\ \text{bang} \vee \text{bang} \wedge \mathsf{EX}\ \text{coin})) \\ & \mathsf{U}\ (\text{coin} \wedge \mathsf{EX}\ (\text{coffee} \vee \neg\text{bang} \wedge \neg\text{coffee} \wedge \text{water}) \vee \text{bang} \wedge \mathsf{EX}\ \text{tea})\end{aligned}$$

It is not difficult to see that the meaning of this formula is equivalent to the meaning of t_1. Indeed, the following is true for both the test t_1 as well as the formula $\mathbb{F}(t_1)$: After a coin coffee is offered, or if coffee is not available and also there is no bang after the coin then the deadlock detection will be triggered and water will be offered; on the other hand, after a coin, a bang, and if no coin is available next, then tea will be offered; or after a coin

and a bang if both coin as well as the tea are available, then two options arise: we can either continue in the cycle, or tea is offered. In all the process can remain indefinitely in the cycle, or can exit from the cycle and pass the test. One can get tea, coffee, or water at the very beginning or after a few repetitions of the cycle.

The formula $\mathbb{F}(t_1)$ holds for all the states of $\xi(p_1)$ (where either coffee or water is offered or coin follows a bang then tea is offered), but it does not hold for the states of $\xi(p_2)$ (where machine dispenses coffee after bang).

Here is another failure trace test, that would allow us to illustrate the behaviour of θ inside the cycle:

$$t_2 = \text{coin}; (\text{coffee}; \text{pass} \,\Box\, \theta; \text{water}; \text{pass} \,\Box\, \theta; (\text{tea}; \text{pass} \,\Box\, t_2))$$

Once again if we apply Theorem 4.3 we can convert this test into the following CTL formula $\mathbb{F}(t_2) = \mathsf{E}(\mathsf{EG}(\text{coin} \wedge \mathsf{EX}(\neg\text{coffee} \wedge (\text{water} \vee \text{coin}))) \vee (\neg\text{coffee} \wedge (\text{water} \vee \text{coin})) \wedge \mathsf{EX}(\text{coin})))\mathsf{U}(\text{coin} \wedge \mathsf{EX}(\top) \wedge \text{coffee} \wedge \mathsf{EX}(\top)) \vee (\neg\text{coffee} \wedge (\text{water} \wedge \mathsf{EX}(\top))) \vee (\neg\text{coffee} \wedge (\text{tea} \wedge \mathsf{EX}(\top)))$. We then eliminate the true sub-formulae as before (to enhance readability) and so we obtain the desired logically equivalent formula:

$$\begin{array}{rcl} \mathbb{F}(t_2) & = & \mathsf{E}(\mathsf{EG}\ (\ \ \text{coin} \wedge \mathsf{EX}\ \neg\text{coffee} \wedge (\text{water} \vee \text{coin}) \vee \\ & & \qquad\qquad \neg\text{coffee} \wedge (\text{water} \vee \text{coin}) \wedge \mathsf{EX}\ \text{coin})) \\ & & \qquad \mathsf{U}\ (\ \ \text{coin} \wedge \mathsf{EX}\ \text{coffee} \vee (\neg\text{coffee} \wedge (\text{water} \vee \text{tea}))) \end{array}$$

As before it is easy to see that this formula is equivalent to the meaning of t_2, for the following applies to both the test t_2 and the formula $\mathbb{F}(t_2)$: After a coin one can get coffee, or if coffee is not available then two options arise. The first option remains in the cycle whenever a coin is available next, or otherwise exit the cycle provided that tea is available. In the second option water is offered. Onece again we can have an arbitrary number of repetitions of coins and hits before the desired beverage is offered.

The formula $\mathbb{F}(t_2)$ does not hold for any of the states of $\xi(p_1)$ and $\xi(p_2)$, since after a coin nothing is dispensed by either machine running in parallel with the test, with the

continuation of the cycle also being unavailable. Thus both machines actually fail the test t_2.

Note in passing that if we apply Lemma 4.2 directly on t_1 and t_2 then we will in both cases obtain infinite CTL formulae.

Chapter 5

Conclusions

Our work described in this paper is based on the definition of equivalence between LTS and Kripke structures (Definition 3.3), the construction of an algorithmic function ξ that converts an LTS into its equivalent Kripke structure (Proposition 3.1), and then the construction of a function $\mathbb{T}$ which converts the CTL formula into its equivalent tests (Proposition 3.2). The existence of an algorithmic function $\mathbb{F}$ that converts failure trace tests into equivalent CTL formulae was not proven correctly earlier [5], so we provided here a correct and complete proof (Lemma 4.2). Together with the previous work on that matter, we have shown that CTL and failure trace tests are equivalent (Theorem 4.1).

We also noted that the function $\mathbb{F}$ produces infinite formulae whenever the test being converted features potentially infinite runs (or loops). The reason for this is that the conversion function does not take advantage of any temporal operator other than X. By using the available temporal operators to their full extent we have succeeded in converting failure trace tests with loops into compact (and most importantly, finite) CTL formulae under the assumption that the loop actions are all different from each other (Theorem 4.3). Extending this result to possibly identical loop actions is one of the arguably two remaining open problems in this area.

The other remaining open problem has already been mentioned earlier [5]. The result

of $\xi(p)$ is a Kripke structure that may have multiple initial states, which in turn requires the use of a weaker satisfaction operator. Whether this issue applies only to LTS that can perform more than a single initial action and so whether this issue can be fixed by just providing a dummy "start" action for all LTS (as claimed earlier [5]) needs to be verified.

In addition to the above the original conclusions [5] continue to apply. More significantly, we have provided a combined, algebraic and logic framework for formal verification. The use of our conversion algorithms allows the mixed specification of a system, where some parts are specified logically and some others algebraically. Which part is specified in which way becomes now a matter of convenience or even taste; no matter how mixed the specification is, we can always obtain a unified specification by applying one of our conversion functions (which one being again a matter of convenience or even taste).

Taste notwithstanding, the convenience of mixed specifications is nicely illustrated by the task of specifying a communication protocol, where the two end points are algorithmic and so it is likely that they are more conveniently specified algebraically, while the communication medium is unknown except for some common properties and so a logical specification is probably more appropriate. Another, more general example is a system with different components at different levels of maturity: the mature parts can be specified algebraically, while the more "loose" logical specification can be used for the less mature ones.

In addition to all of the above, it is also worth noting that once we have a (any!) specification, we can effectively use it to verify a system using either model checking or a model-based testing tool (or both!).

In all, we believe that we have provided a very useful tool for formal verification.

Bibliography

[1] C. BAIER AND J.-P. KATOEN, *Principles of Model Checking*, MIT Press, 2008.

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

[3] 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.

[4] S. D. BRUDA, *Preorder relations*, in Broy et al. [3], pp. 117–149.

[5] 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.

[6] 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.

[7] E. M. CLARKE, E. A. EMERSON, AND A. P. SISTLA, *Automatic verification of finite state concurrent systems using temporal logic specification*, ACM Transactions on Programming Languages and Systems, 8 (1986), pp. 244–263.

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

[9] 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.

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

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

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

[13] J.-P. KATOEN, *Labelled transition systems*, in Broy et al. [3], pp. 615–616.

[14] 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.

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

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

[17] 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.

[18] 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.

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

[20] 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.

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

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

[23] A. F. M. N. UDDIN, *Computation tree logic is equivalent to failure trace testing*, Master's thesis, Bishop's University, July 2015.