The Equivalence of Computation Tree Logic and Failure Trace Testing under Multiple Conversion Algorithms

by

Rui Zuo

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
April 2018

Abstract

The two major systems of formal verifications are model checking and model-based testing. Model checking is based on some temporal logic such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL, the focus of this thesis) for specification and on Kripke structures as models for the systems under test. Model-based testing is an algebraic technique which is based on some operational semantics of processes (such as traces and failures) and its associated pre-orders. One of the most fine-grained pre-order is based on both failures and traces. A previous line of investigation [6, 24, 27] showed that CTL and failure trace testing are equivalent. This equivalence was based in turn on a constructive equivalence between LTS and Kripke Structure; in order for this equivalence to work sets of states rather than individual states need to be considered in model checking. In this paper we will consider another, equally constructive equivalence based on another line of investigation of the matter [13]. We find that this equivalence does not require any modification to the model checking algorithms and CTL and failure trace testing continue to be equivalent under it.

The original conversion algorithm of failure trace tests into CTL formulae [6] produces infinite formulae in certain circumstances. Further work on this matter [24] produced an improved algorithm that under certain circumstances produces finite formulae. We further improve on this algorithm to make it more general.

Contents

Chapter 1

Introduction

We depend on computing systems in every important aspect of our everyday life, to assist us in transportation and communication, but also in more leisurely activities. Almost everything is based on computing systems. Computers are not only not used in our daily life. They also play an important role in mission-critical systems, whose malfunction can lead to loss of property or life. Examples include chemical plants, nuclear power facilities, military equipment, and rockets. Because of widespread using, such an error happened in software or hardware will result in catastrophic consequences. Therefore, ensuring the correct behaviour of computing system is challenging but also increasingly important.

Verification is the process that ensures that a system is correct and follows the desired properties or specifications. Several verification methods exist with the olest being empirical testing [18, 23]. Empirical testing is a non-formal method which consists of providing input data, observing the output, and verifying whether the results conform to expectations. However, this method cannot cover all the possibilities, meaning that the method cannot cover all the situations which might happen in the future that will result in a fault. Thus empirical testing may discover defects but will never be able to guarantee correctness.

Deductive verification [15, 21] is a formal method of verification which is also called program proving. The process is based on a mathematical system of axioms and inference rules and offers a mathematical proof of correctness. Deductive verification can also be used for reasoning about infinite state systems. Unfortunately this method can never be fully automated. The manual process consumes substantial time and requires highly qualified experts.

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 [3, 12, 26] and model checking [8, 9, 20, 1]. In model-based testing the specification of a system is algebraic in nature, using formalisms such as labelled transition systems (LTS) or finite automata. Such a specification is normally an abstract description of the system's desired behaviour. Test cases are algorithmically derived from the specification and then run on the system under test. This way we theoretically ensure that the results are sound and complete. Model checking expresses the specification using some form of temporal logic and tries to determine algorithmically whether the respective temporal logic formula holds for the system under test, modelled in turn by a Kripke structure.

Both these formal methods have advantages and disadvantages. Model-based testing is compositional by definition and so scalable. Model checking is not compositional and so it requires a complete model of the system under test. Another, related problem is that the finite-state nature of Kripke structures can lead to an exponentially increase in the number of distinct states as the complexity of the system increases. Model-based testing on the other hand is not necessary complete, since some of the tests can take infinite time to run. The logical nature of the specification in model checking allows us to only specify the properties of interest. In model-based testing, the finite automata or labelled transition systems representation requires the specification of more or less the whole system.

Various kinds of temporal logic are used to specify the system in model checking, including CTL*, CTL and LTL. In this paper we will focus on CTL. Apart from this, we will also focus on probably the most powerful practically meaningful 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.

The subject of this thesis is the equivalence between model-based testing and model checking. The aim is to open the possibility of mixed, algebraic and logic specification, but also to allow the convenience of using either model checking or model-based testing irrespective of the form taken by the specification. The various advantages and disadvantages of these formal methods illustrate the utility of this effort. In addition, temporal logic may be natural for some system properties, while others can be better specified using LTS or finite automata. Such a mixed specification could be given by somebody else, or it could be simply the case that algebraic specifications are more convenient form some aspects while logic specifications are more suitable for others. No matter the motivation, it is still the case that there is no global algorithm for verifying the whole system, even if some parts can be model checked and some other parts can be verified using model-based testing. We do not even have a global specification for the system. A pursuit toward finding such a global specification spans a number of papers [6, 24, 27] and provides algorithmic conversions between the algebraic specification given by failure trace testing (a flavor of model-based testing) and CTL formulae (used by model checking). This thesis continues this investigation. The previous work just mentioned uses an algorithmic conversion between LTS and Kripke structures that results in very compact Kripke structures but introduces the need to modify the model checking algorithm (by requiring a modified notion of satisfaction for CTL formulae). We now use a different (and still algorithmic) such a conversion based on a different investigation [13] which results in considerably larger Kripke structures but does not require any modification of the model checking algorithm. We find that all the equivalence relations developed earlier [6, 24, 27] continue to hold for this conversion with minimal modifications to the original algorithms. Our thesis is therefore that failure trace testing and CTL are equivalent under any reasonable equivalence relation between LTS and Kripke structures.

In addition we note the algorithm for converting failure trace tests into compact CTL formulae [24] and we improve on it by eliminating some (though not all) of its limitations.

We believe that our effort opens the domain of combined algebraic and logical formal methods. 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: We introduce model checking, model-based testing, temporal logic, and failure trace testing in Chapter 2. We then summarize in Chapter 3 the related previous work which includes two constructive equivalence relations between label transition systems and Kripke structures, the conversion algorithm of CTL formulae into failure trace tests, and also the algorithmic conversion from failure trace test to (compact) CTL formulae. We then proceed with the presentation of our work namely, a new constructive conversion from LTS to Kripke structure in Chapter 4 and the more general algorithmic conversion from failure trace tests to compact CTL formulae in Chapter 5. Our conclusions are provided in Chapter 6. For the remainder of this thesis results proved elsewhere are introduced as propositions, while original results are stated as theorems and lemma.

Chapter 2

Preliminaries

This section covers temporal logic, model checking, labelled transition systems, stable failures, and failure trace testing. In several previous papers the process algebra TLOTOS is used to specify the system under test as well as failure trace tests. Thus, 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 [6, 24, 27]. The content of this section is therefore necessarily similar to the corresponding sections from the previous work.

In what follows we use A^* to denote the set of exactly all the sequences of symbols from A. A binary relation that is reflexive and transitive is called as usual pre-order.

2.1 Temporal Logic and Model Checking

Temporal logic formulae describe a specification suitable for verification using model checking. The system under test is modelled as a Kripke structure. The main goal of model checking is to find the set of all states in a Kripke structure that satisfy the given logic formula. If the states labelled as initial states are in this set then the system is deemed to have satisfied the specification.

A *Kripke structure* 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 [9]. 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 thus feature a "self-loop" transition. A *path* π in a Kripke structure is a non-empty (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 often called the root or the start state of the part. 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 such a family consists of CTL* [14, 9], CTL [9, 7] (computation tree logic) and LTL [19] (linear-time temporal logic). CTL and LTL are defined as restricted subsets of CTL*. CTL is interpreted over computation trees and LTL is interpreted over individual paths (or runs).

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 a computation tree have several other successive states which lead to multiple paths starting from the same state. There are five temporal operators which are used to represent 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 formulae 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 a path quantifier (Aor E). Thus, we have the following syntax for CTL formulae, noting that all the CTL formulae are state formulae:

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

where a is an atomic proposition ranging over AP and f, f_1, f_2 are state formulae. In what follows we let $\mathcal{F}$ denote the set of all CTL formulae.

The CTL semantics is defined with respect to Kripke structures. 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 in state s. Path formulae are also used in the definition of the CTL semantics below. We thus extend this notation to path formulae as follows: 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 $K, \pi \models f$.

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 π^0 being the starting state. 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 formulae f and g iff there exists $j \geq 0$ such that $K, \pi^j \models g$ and $K, \pi^i \models f$ for all $0 \leq i < j$. In other words, g must become true in some state s_j, and f must hold in all the previous states (from s_0 to s_{j-1}).

3. $K, \pi \models f \,\mathsf{R}\, g$ for any state formula f and g iff for all $j \geq 0$ if $K, \pi^i \not\models f$ for all $0 \leq i < j$ then $K, \pi^i \models g$ for all $0 \leq i < j$. In other words, g must remain true until f becomes true and releases g from its obligations.

2.2 Labelled Transition Systems and Stable Failures

The semantics of CTL is defined over Kripke structures, where each state is labelled with atomic propositions. In model-based testing the common models are the labelled transition systems (LTS) and the finite automata, where labels are associated with transitions instead of states.

An LTS [16] 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, and A is a countable set of labels which denotes all the 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 represents a 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". For the reminder of the thesis we use $\mathcal{P}$ to denote the set of all processes.

Generally, we consider a set T of relevant tests and set P of processes. In model-based testing [12, 3, 26] 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 represents 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 it is not preceded by successful state.

Given the non-deterministic nature of some tests and processes, we can have multiple runs for the given test t and process p, and thus a set of outcomes is needed to provide the results of all the possible runs. Let $\mathrm{Obs}(p, t)$ 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 successful run (that is, p may t iff $\top \in \mathrm{Obs}(p, t)$), while p must pass the test t whenever all the runs are successful (that is, p must t iff $\{\top\} = \mathrm{Obs}(p, t)$).

To analyse the behaviour of the processes, we need to consider those sequences of events that can be observed at the interface of the process. There are a number of ways through which this behaviour can be analysed. One aspect of process behaviour is the occurrence of certain events 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 and are defined as follows:

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 $\text{trace}(\pi) = (a_i)_{0<i\leq|\pi|, a_i \neq \tau} \in A^*$. 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 $\text{traces}(p) = \{w : p \overset{w}{\Longrightarrow}\}$. The set of finite traces of process p is defined as $\text{Fin}(p) = \{tr \in \text{traces}(p) : |tr| \in \mathbb{N}\}$ where $|tr|$ refers to the length of trace tr.

A process is said to be *stable* [22] 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 $X \subseteq A$. Whenever p cannot perform any event from X then p will *refuse* the set X. We use the following notation for this purpose: $p \text{ ref } X$ 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 (or trace) 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 [22] whenever $\exists p^w : p \overset{w}{\Longrightarrow} p^w \wedge p^w \downarrow \wedge p^w \text{ ref } X\}$, meaning that p performs the events in w and then reaches a stable state from where it refuses all the events in the set X. The stable failures of p are then described as $\text{SF}(p) = \{(w, X) : \exists p^w : p \overset{w}{\Longrightarrow} p^w \wedge p^w \downarrow \wedge p^w \text{ ref } X\}$.

Depending on the level of interaction with processes, many pre-order relations can be defined (like traces, stable failure, refusal etc.). In general, pre-orders are more convenient and more meaningful than equivalences in comparing specifications and their implementation: if two systems are in a pre-order relation with each other, then one is the implementation of other. Thus such pre-orders can be interpreted as implementation relations in practice. The *stable failure pre-order* $\sqsubseteq_{\mathrm{SF}}$ 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 pre-order $\sqsubseteq_{\mathrm{SF}}$ 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 pre-order $\sqsubseteq_{\mathrm{SF}}$ is considered one of the finest pre-orders [4].

2.3 Failure Trace Testing

In what follows we use the notation $\mathrm{init}(p) = \{a \in A : p \overset{a}{\Longrightarrow}\}$. A failure trace [17] 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 $p \overset{\varepsilon}{\Longrightarrow} p_0 \overset{a_1}{\Longrightarrow} p_1 \overset{a_2}{\Longrightarrow} \cdots \overset{a_n}{\Longrightarrow} p_n$; $f = A_0 a_1 A_1 a_2 A_2 \ldots A_n a_n$ is a failure trace of p if the following two conditions are observed:

- If $p_i \xrightarrow{\tau}$ then $A_i = \emptyset$; when p_i is not stable then it will refuse an empty set of events by definition.
- If $\neg(p_i \xrightarrow{\tau})$, 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 (which could also be the empty set).

In other words, we produce a failure trace of a process p by taking a trace of p and then place refusal sets in between its actions after the stable states.

In this paper, we will use the testing language TLOTOS [17, 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 non-deterministic 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 $\text{trace}(\pi)$ 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 $\{\top\} = \text{Obs}(p, t)$.

In what follows $\mathcal{T}$ will denote the set of all failure trace tests. In addition, 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 $\Sigma\{a; \text{stop} : a \in A'\} \,\Box\, \theta; t \in \mathcal{ST}$ for any $A' \subseteq A$.

A bijection between failure traces and sequential tests exists [17]. For a sequential test t the failure trace $\text{ftr}(t)$ is defined inductively as follows: $\text{ftr}(\textit{pass}) = \emptyset$, $\text{ftr}(a; t') = a\ \text{ftr}(t')$, and $\text{ftr}(\Sigma\{a; \text{stop} : a \in A'\} \,\Box\, \theta; t') = A\ \text{ftr}(t')$. Conversely, let f be a failure trace. Then we inductively define the sequential test $\text{st}(f)$ as follows: $\text{st}(\emptyset) = \textit{pass}$, $\text{st}(af) = a\ \text{st}(f)$, and $\text{st}(A'f) = \Sigma\{a; \text{stop} : a \in A'\} \,\Box\, \theta; \text{st}(f)$ with $A' \subseteq A$. 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 pre-order into a testing based pre-order. 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 pre-order $\sqsubseteq_{\text{FT}}$ as follows: $p \sqsubseteq_{\text{FT}}$ q iff $\text{ftr}(p) \subseteq \text{ftr}(q)$. This pre-order is equivalent to the stable failure pre-order.

Proposition 2.1 **[17]** *Let p be a process, t be a sequential test, and f be a failure trace. Then p may t iff $\text{ftr}(t) \in \text{ftr}(p)$ (note that $\text{ftr}(t)$ is always a single failure trace).*

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 pre-orders, $\sqsubseteq_{\text{SF}}$ can be characterized in terms of may testing only; the must operator does not need to be considered any further.

Chapter 3

Previous Work

This chapter contains a summary of the previous work on combined, logical and algebraic frameworks of formal specification and verification. With the exception of the research continued by this thesis most of this effort was directed toward LTL and its relationship with Büchi automata [25].

Büchi automata were used as a semantical basis for reasoning about combined logical and algebraic specification namely, LTL and the DeNicola and Hennessy testing preorders [10]. A unified semantic theory for heterogeneous system specifications featuring a mixture of LTS and LTL formulae was developed. First the Büchi must-pre-order 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. This effort was further extended to the real-time domain [5, 11].

We are aware of only two investigations relating CTL with algebraic specifications. One is the work continued in this thesis. This work first introduces a constructive conversion of LTS into equivalent Kripke structures [6], and then it constructs the conversion of failure trace tests into CTL formulae and the other way around [6, 24, 27]. The second investigation starts from DeNicola's research [13] which uses a "dummy" elementary proposition to obtain yet another algorithmic conversion between LTS and Kripke structures. Both these investigations will be presented in the remainder of this chapter.

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 [6]. Basically, the actions available in an LTS state are propositions that hold in that state.

Definition 3.1 SATISFACTION FOR PROCESS [6]: *A process p satisfies $a \in A$, written $p \models a$, iff $p \xrightarrow{a}$. That p satisfies some (general) CTL state formula is defined inductively as follows. Let f and g be 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 paths 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 [6]: *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 [6]: *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 [6] *There exist an algorithmic function $\mathbb{K}$ 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 $n \in N$, $s \overset{n}{\Longrightarrow} t$,

   (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).

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

A sample conversion according to $\mathbb{K}$ is shown graphically in Figure 3.1. A state of the Kripke structure $\mathbb{K}(p)$ is defined based on an LTS state and one of its corresponding outgoing actions. Note in particular the split of the LTS state p into two states in the Kripke structure (corresponding to the two actions a and b enabled in p).

3.2 CTL Formulae Are Equivalent with Failure Trace Tests

Recall that $\mathcal{P}$ 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. The equivalence between CTL formulae and failure trace tests is established by Propositions 3.2 and 3.3 below.

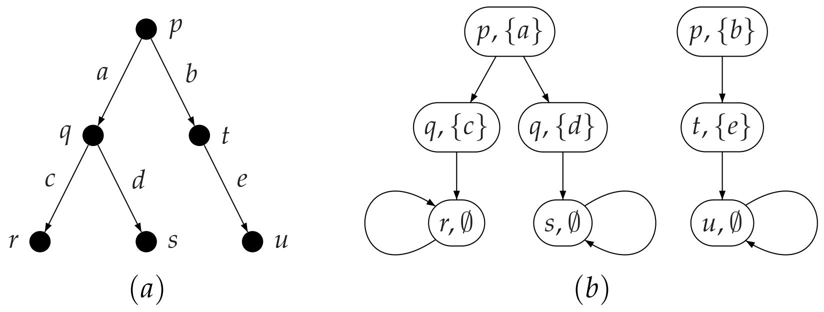

Figure 3.1: Illustration of the conversion from an LTS p (a) to its equivalent Kripke structure $\mathbb{K}(p)$ (b).

Proposition 3.2 [27] *There exist a function* $\mathbb{T}_{\mathbb{K}} : \mathcal{F} \to \mathcal{T}$ *such that* $\mathbb{K}(p) \models f$ *iff* p may $\mathbb{T}_{\mathbb{K}}(f)$ *for any* $p \in \mathcal{P}$.

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

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

The induction for non-temporal operators goes as follows:

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

The definition of complement, conjunction, and disjunction of tests is given elsewhere [27].

The temporal operators are converted as follows:

1. $\mathbb{T}_{\mathbb{K}}(\mathsf{EX}\, f) = \Sigma\{a; \mathbb{T}_{\mathbb{K}}(f) : a \in A\}$
2. $\mathbb{T}_{\mathbb{K}}(\mathsf{EF}\, f) = t'$ such that $t' = \mathbb{T}_{\mathbb{K}}(f) \,\Box\, (\Sigma(a; t' : a \in A))$.
3. $\mathbb{T}_{\mathbb{K}}(\mathsf{EG}\, f) = \mathbb{T}_{\mathbb{K}}(f) \wedge (\mathbb{T}_{\mathbb{K}}(\mathsf{EX}\, f') \,\Box\, \theta; \text{pass})$, with $f' = f \wedge \mathsf{EX}\, f'$.
4. $\mathbb{T}_{\mathbb{K}}(\mathsf{E}\, f_1 \,\mathsf{U}\, f_2) = (\mathbb{T}_{\mathbb{K}}(f_1) \wedge (\mathbb{T}_{\mathbb{K}}(\mathsf{EX}\, f') \,\Box\, \theta; \text{pass})) \,\Box\, \mathbf{i}; (\mathbb{T}_{\mathbb{K}}(f_2) \wedge (\mathbb{T}_{\mathbb{K}}(\mathsf{EX}\, f'') \,\Box\, \theta; \text{pass}))$, 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 [27]. $\Box$

Proposition 3.3 [24] *There exists a function $\mathbb{F}_{\mathbb{K}} : \mathcal{T} \to \mathcal{F}$ such that p* may *t if and only if $\mathbb{K}(p) \models \mathbb{F}_{\mathbb{K}}(t)$ for any $p \in \mathcal{P}$.*

Proof sketch: The proof is done yet again by structural induction over failure trace formulae and the function $\mathbb{F}_{\mathbb{K}}$ is also defined at the same time (inductively).

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

We then have $\mathbb{F}_{\mathbb{K}}(\mathbf{i}; t) = \mathbb{F}_{\mathbb{K}}(t)$: by definition, an internal action is not seen by the external environment of the system under test. We also have $\mathbb{F}_{\mathbb{K}}(a; t) = a \wedge \mathsf{EX}\, \mathbb{F}_{\mathbb{K}}(t)$.

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 $\mathbb{F}_{\mathbb{K}}(\Sigma T) = \bigvee \mathbb{F}_{\mathbb{K}}(t) : t \in T$. p may ΣT iff p may t for at least one $t \in T$ iff $\mathbb{K}(p) \models \mathbb{F}_{\mathbb{K}}(t)$ for at least one $t \in T$(by induction hypothesis) iff $\mathbb{K}(p) \models \bigvee \mathbb{F}_{\mathbb{K}}(t) : t \in T$.

Whenever θ does not participate in a choice then it behaves exactly like an internal action $\mathbf{i}$, so we can assume without loss of generality that θ appears only in choice constructs. We also 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 $\mathbb{F}_{\mathbb{K}}(t_1 \,\Box\, \theta; t) = ((\bigvee \mathit{init}(t_1)) \wedge \mathbb{F}_{\mathbb{K}}(t_1)) \vee (\neg(\bigvee \mathit{init}(t_1)) \wedge \mathbb{F}_{\mathbb{K}}(t))$. $\Box$

3.3 Converting Failure Trace Tests into Compact CTL Formulae

The function $\mathbb{F}_{\mathbb{K}}$ developed in Proposition 3.3 will produce under certain circumstances (namely, when the test features cycles) infinite formulae. This has been remedied to some extent as follows:

Proposition 3.4 [24] *There exists an algorithmic function, denoted by abuse of notation $\mathbb{F}_{\mathbb{K}} : \mathcal{T} \to \mathcal{F}$ such that p* may *t if and only if $\mathbb{K}(p) \models \mathbb{F}_{\mathbb{K}}(t)$ for any $p \in \mathcal{P}$ and $\mathbb{F}_{\mathbb{K}}(t)$ 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 $t_0 \xRightarrow{a_1} t_1 \xRightarrow{a_2} t_2 \xRightarrow{a_3} \cdots \xRightarrow{a_n} t_n = t_0$ we have $a_1 \neq a_2 \neq \cdots \neq a_n$.*

Proof sketch: Consider a loop test of the following form:

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

Then the test can be converted into an equivalent formula using the following transformation:

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

where C_i represents the cycle in its various stages such that

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

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

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

The formula above assumes that neither the actions in the cycle a_i, $0 \leq i < n$ nor the top-level actions of the exit tests init(t_i), $0 \leq i < n$ are θ. The deadlock detection action is then introduced 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 $t_{k-1 \text{ mod } n}$ but not as alternate a_k (Case 2), or both as a_k and on the top level of the alternate $t_{k-1 \text{ mod } n}$ (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 $\theta \notin \text{init}(t_{k-1 \text{ mod } n})$ then all the occurrences of $\mathbb{F}_{\mathbb{K}}(a_k)$ in $\mathbb{F}_{\mathbb{K}}(t)$ are replaced with $\neg(\bigvee_{b \in \text{init}(t_{k-1 \text{ mod } n})} \mathbb{F}_{\mathbb{K}}(b))$ in conjunction with $\bigvee_{b \in \text{init}(t_k) \setminus \{\theta\}} \mathbb{F}_{\mathbb{K}}(b)$ for the "exit" formulae and with $\mathbb{F}_{\mathbb{K}}(a_{k+1 \text{ mod } n})$ for the "cycle" formulae. Therefore $C_i =$ $\mathsf{EG}(\mathbb{F}_{\mathbb{K}}(a_i) \wedge \mathsf{EX}(\cdots \wedge \mathsf{EX}(\neg(\bigvee_{b \in \text{init}(t_{k-1 \text{ mod } n})} \mathbb{F}_{\mathbb{K}}(b)) \wedge \mathbb{F}_{\mathbb{K}}(a_{k+1 \text{ mod } n}) \wedge \mathsf{EX} \cdots \wedge$ $\mathsf{EX}(\mathbb{F}_{\mathbb{K}}(a_{(i+n-1) \text{ mod } n})) \cdots)))$ and $E_k = \neg(\bigvee_{b \in \text{init}(t_{k-1 \text{ mod } n})} \mathbb{F}_{\mathbb{K}}(b)) \wedge \bigvee_{b \in \text{init}(t_k) \setminus \{\theta\}} \mathbb{F}_{\mathbb{K}}(b) \wedge$ $\mathsf{EX}(\mathbb{F}_{\mathbb{K}}(t_k))$.

Case 2: Whenever $\theta \in \text{init}(t_{k-1 \text{ mod } n})$ and $a_k \neq \theta$ the exit formula $E_{k-1 \text{ mod } n}$ is changed to contain two components. If any action in $\text{init}(t_{k-1 \text{ mod } n})$ is available then such an action can be taken, so a first component is $\mathbb{F}_{\mathbb{K}}(a_{k-1 \text{ mod } n}) \wedge \mathsf{EX}\ (\bigvee_{b \in \text{init}(t_{k-1 \text{ mod } n}) \setminus \{\theta\}} \mathbb{F}_{\mathbb{K}}(b)) \wedge$ $\mathbb{F}_{\mathbb{K}}(t_{k-1 \text{ mod } n})$. Note that any θ top-level branch in $t_{k-1 \text{ mod } n}$ is invalidated (since some action $b \in \text{init}(t_{k-1 \text{ mod } n})$ is available). The top-level θ branch of $t_{k-1 \text{ mod } n}$ can be taken only if no action from $\text{init}(t_{k-1 \text{ mod } n}) \cup \{a_k\}$ is available, so the second variant is $\mathbb{F}_{\mathbb{K}}(a_{k-1 \text{ mod } n})$ $\wedge\ \mathsf{EX}\ \neg\mathbb{F}_{\mathbb{K}}(a_{(k)}) \wedge \neg(\bigvee_{b \in \text{init}(t_{k-1 \text{ mod } n}) \setminus \{\theta\}} \mathbb{F}_{\mathbb{K}}(b)) \wedge \mathbb{F}_{\mathbb{K}}(t_{k-1 \text{ mod } n}(\theta))$, where $t_{k-1 \text{ mod } n} =$ $t' \,\square\, \theta; t_{k-1 \text{ mod } n}(\theta)$ for some test t'.

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

Case 3: If $a_k = \theta$ and $\theta \in \text{init}(t_{k-1 \text{ mod } n})$, then both the cycle and the exit test are modified. Let $B = \text{init}(t_{k-1 \text{ mod } n}) \setminus \{\theta\}$.

If an action from B is available the cycle cannot continue, so all the occurrences of a_k in all the formulae C_i are replaced with $\neg(\bigvee_{b \in B} \mathbb{F}_{\mathbb{K}}(b)) \wedge \bigvee_{b \in \{a_{k+1 \text{ mod } n}\} \cup \text{init}(t_k) \cup \setminus \{\theta\}} \mathbb{F}_{\mathbb{K}}(b)$ so that $C_i = \mathsf{EG}(\mathbb{F}_{\mathbb{K}}(a_i) \wedge \mathsf{EX}(\cdots \wedge \mathsf{EX}(\mathbb{F}_{\mathbb{K}}(\neg(\bigvee_{b \in \text{init}(t_{k-1 \text{ mod } n}) \setminus \{\theta\}} \mathbb{F}_{\mathbb{K}}(b))) \wedge$

$\bigvee_{b\in\{a_{k+1 \bmod n}\}\cup\text{init}(t_k)\setminus\{\theta\}} \mathbb{F}_{\mathbb{K}}(b) \wedge \mathsf{EX} \cdots \wedge \mathsf{EX}(\mathbb{F}_{\mathbb{K}}(a_{(i+n-1) \bmod n})) \cdots))).$

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 $E_{k-1 \bmod n} = \mathbb{F}_{\mathbb{K}}(a_{k-1 \bmod n}) \wedge \mathsf{EX}\ \bigvee_{b\in\text{init}(t_{k-1 \bmod n})\setminus\{\theta\}} \mathbb{F}_{\mathbb{K}}(b) \wedge \mathbb{F}_{\mathbb{K}}(t_{k-1 \bmod n}) \vee \neg(\bigvee_{b\in\text{init}(t_{k-1 \bmod n})\setminus\{\theta\}} \mathbb{F}_{\mathbb{K}}(b)) \wedge \mathbb{F}_{\mathbb{K}}(t_{k-1 \bmod n}(\theta))$. As before, $t_{k-1 \bmod n}(\theta)$ is the θ-branch of $t_{k-1 \bmod n}$ that is, $t_{k-1 \bmod n} = t' \,\square\, \theta; t_{k-1 \bmod n}(\theta)$ for some test t'.

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

Chapter 4

Yet Another Constructive Equivalence between LTS and Kripke Structures

The function $\mathbb{K}$ developed earlier [6] produces a very compact Kripke structure. However, a state in the original LTS can result in multiple equivalent state in the resulting Kripke structure, which in turn requires a modified notion of satisfaction (over sets of states, see Definition 3.2). This in turn implies a non-standard model checking algorithm. A different such a conversion algorithm [13] avoids this issue, at the expense of a considerably larger Kripke structure. We now explore a similar equivalence, and then (in Chapter 5) we will study the equivalence between failure trace testing and CTL under it.

The just mentioned conversion algorithm [13] is based on introducing intermediate states in the resulting Kripke structure. These states are labelled with the special proposition Δ which is understood to mark a state that is ignored in the process of determining the truth value of a CTL formula; if Δ labels a state then it is the only label for that state. We therefore base our construction on the following definition of equivalence between processes and Kripke structures:

Definition 4.1 *Given a Kripke structure K and a state s of K, the pair K, s is equivalent to a* *process p, written as $K, s \simeq p$ (or $p \simeq K, s$) iff for any CTL* formula f $K, s \models f$ iff $p \models f$. The operator $\models$ is defined for processes in Definition 3.1 and for Kripke structures as follows:*

1. $p \models \top$ *iff* $K, s \models \top$
2. $p \models a$ *iff* $K, s \models \Delta \,\mathsf{U}\, a$
3. $p \models \neg f$ *iff* $K, s \models \neg f$
4. $p \models f \wedge g$ *iff* $K, s \models f \wedge g$
5. $p \models f \vee g$ *iff* $K, s \models f \vee g$
6. $p \models \mathsf{E}f$ *iff* $K, s \models \mathsf{E}f$
7. $p \models f \,\mathsf{U}\, g$ *iff* $K, s \models (\Delta \vee f) \,\mathsf{U}\, g$
8. $p \models \mathsf{X}\, f$ *iff* $K, s \models \mathsf{X}\, (\Delta \,\mathsf{U}\, f)$
9. $p \models \mathsf{F}f$ *iff* $K, s \models \mathsf{F}\, f$
10. $p \models \mathsf{G}\, f$ *iff* $K, s \models \mathsf{G}\, (\Delta \vee f)$
11. $p \models f \,\mathsf{R}\, g$ *iff* $K, s \models f \,\mathsf{R}\, (\Delta \vee g)$

Note that the definition above is stated in terms of CTL* rather than CTL; however, CTL* is stronger and so equivalence under CTL* implies equivalence under CTL.

Most of the equivalence is immediate. However, some cases need to make sure that the states labelled Δ are ignored. This happens first in $K, s \models \Delta \,\mathsf{U}\, a$, which is equivalent to $p \models a$. Indeed, a needs to hold immediately, except that any preceding states labelled Δ must be ignored, hence a must be eventually true and when it becomes so it releases the chain of Δ labels. The formula for X is constructed using the same idea (except that the formula f releasing the possible chain of Δ happens starting from the next state).

Then expression $(\Delta \vee f) \mathsf{U} g$ means that f must remain true with possible interleaves of Δ until g becomes true. Similarly $f \mathsf{R} (\Delta \vee g)$ requires that g is true (with the usual interleaved Δ) until it is released by f becoming true.

Based on this equivalence we can define a new conversion of LTS into equivalent Kripke structures. This conversion is again based on a similar conversion [13] developed in a different context.

Theorem 4.1 *There exist at least two algorithmic functions for converting LTS into equivalent Kripke structures. The first is the function $\mathbb{K}$ described in Proposition 3.1.*

The new function $\mathbb{X}$ is defined as follows: with Δ a fresh symbol not in A, given an LTS $p = (S, A, \rightarrow, s_0)$, the Kripke structure $\mathbb{X}(p) = (S', Q, R', L)$ is given by:

1. $\mathsf{AP} = A \uplus \Delta$;
2. $S' = S \cup \{(r, a, s) : a \in A \text{ and } r \xrightarrow{a} s\}$;
3. $Q = \{s_0\}$;
4. $R' = \{(r, s) : r \xrightarrow{\tau} s\} \cup \{(r, (r, a, s)) : r \xrightarrow{a} s\} \cup \{((r, a, s), s) : r \xrightarrow{a} s\}$;
5. *For* $r, s \in S$ *and* $a \in A : L(s) = \{\Delta\}$ *and* $L((r, a, s)) = \{a\}$.

Then $p \simeq \mathbb{X}(p)$.

Proof. We prove the stronger equivalence over CTL* rather than CTL by structural induction. Since Δ is effectively handled by the satisfaction operator introduced in Definition 4.1 it will turn out that there is no need to mention it at all.

For the basis of the induction, we note that $\top$ is true for any process and for any state in any Kripke structure. $p \models \top$ iff $\mathbb{X}(p) \models \top$ is therefore immediate. The same goes for $\perp$ (no process and no state in any Kripke structure satisfy $\perp$). $p \models a$ iff $\mathbb{X}(p) \models a$; Indeed, $a \in A$(so that $p \models a$) iff $a \in L((r, a, s))$.

That $p \models \neg f$ iff $\mathbb{X}(p) \models \neg f$ is immediately given by the induction hypothesis that $p \models f$ iff $\mathbb{X}(p) \models f$.

Suppose that $p \models f$ and [or] $p \models g$ (so that $p \models f \wedge g$ [$p \models f \vee g$]). This is equivalent by induction hypothesis to $\mathbb{X}(p) \models f$ and [or] $\mathbb{X}(p) \models g$, that is, $\mathbb{X}(p) \models f \wedge g$ [$\mathbb{X}(p) \models f \vee g$], as desired.

Let now π' be a path $\pi' = p \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ starting from a process p. According to the definition of $\mathbb{X}$, all the equivalent paths in the Kripke structure $\mathbb{X}(p)$ have the form $\pi' = \Delta \rightarrow A_0 \rightarrow \Delta \rightarrow A_1 \rightarrow \Delta \rightarrow A_2 \rightarrow \cdots \Delta \rightarrow A_n$, such that $a_i \in A_i$ for all $0 \leq i < n$. Clearly, such a path π' exists. According to the function $\mathbb{X}$, we know that Δ is a symbol that stands for states in the LTS and has no meaning in the Kripke structure. The satisfaction operator for Kripke structures (Definition 4.1) is specifically designed to ignore the Δ label and this insures that the part π' is equivalent to the path $\pi = A_0 \rightarrow A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n$ with $a_i \in A_i$ for all $0 \leq i < n$ and so we will use this form for the reminder of the proof.

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

The proof for F, G, U, and R operators proceed similarly. Whenever $\pi \models \mathsf{F}\,f$, there is a state π^i such that $\pi^i \models f$. By induction hypothesis then $\mathbb{X}(\pi)^i \models f$ and so $\mathbb{X}(\pi) \models \mathsf{F}\,f$. The other way (from $\mathbb{X}(\pi)$ to π) is similar. The G operator requires that all the states along π satisfy f, which implies that all the states in any $\mathbb{X}(\pi)$ satisfy f, and thus $\mathbb{X}(\pi) \models \mathsf{G}\,f$ (and again things proceed similarly in the other direction). In all, the induction hypothesis established a bijection between the states in π and the states in (any) $\mathbb{X}(\pi)$. This bijection is used in the proof for U and R just as it was used in the above proof for F and G. Indeed, the states along the path π will satisfy f or g as appropriate for the respective operator, but

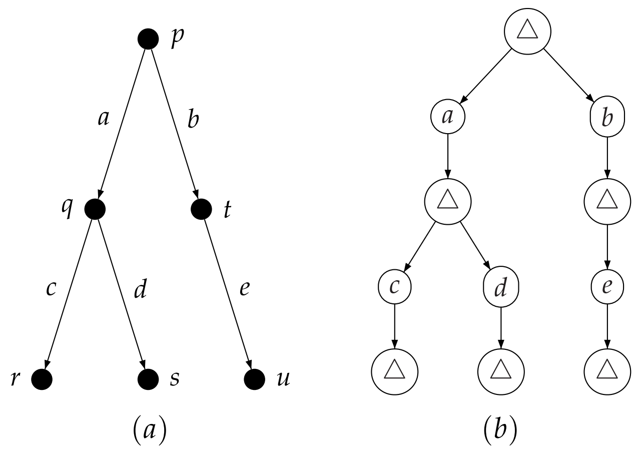

Figure 4.1: Illustration of the conversion from LTS (a) to its equivalent Kripke structure (b).

this translates in the same set of states satisfying f and g in $\mathbb{X}(\pi)$, so the whole formula (using U or R holds in π iff it holds in $\mathbb{X}(\pi)$).

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

The process of the new version described in Theorem 4.1 is most easily described graphically; refer for this purpose to Figure 4.1. Specifically, the function $\mathbb{X}$ converts the LTS given in Figure 4.1(a) into the equivalent Kripke structure shown in Figure 4.1(b). In this new structure, instead of combining each state with its corresponding actions in the LTS (and thus possibly splitting the LTS state into multiple Kripke structure states), we use the new symbol Δ to stand for the original LTS states. Every Δ state of the Kripke structure is the LTS state, and all the other states in the Kripke structure are the actions in the LTS. This ensures that all states in the Kripke structure corresponding to actions that are outgoing from a single LTS state have all the same parent. This in turn eliminates the need for the weaker satisfaction operator over sets of states (Definition 3.2).

Chapter 5

CTL Is Equivalent to Failure Trace Testing

We now find that the equivalence between CTL and failure trace testing as described in Section 3.2 also holds under the new equivalence between LTS and Kripke structures. Recall that $\mathcal{F}$ stands for the set of CTL formulae and $\mathcal{T}$ for the set of failure trace tests. We then have:

Theorem 5.1 *There exist a function* $\mathbb{T}_{\mathbb{X}} : \mathcal{F} \to \mathcal{T}$ *such that* $\mathbb{X}(p) \models f$ *iff* p may $\mathbb{T}_{\mathbb{X}}(f)$ *for any* $p \in \mathcal{P}$*. There exists a function* $\mathbb{F}_{\mathbb{X}} : \mathcal{T} \to \mathcal{F}$ *such that* p may t *if and only if* $\mathbb{X}(p) \models \mathbb{F}_{\mathbb{X}}(t)$ *for any* $p \in \mathcal{P}$*.*

Proof. The proof established earlier for $\mathbb{T}_{\mathbb{K}}$ [27] (see Proposition 3.2) will also work for $\mathbb{T}_{\mathbb{X}}$. Indeed, the way the operator $\models$ is defined (Definition 4.1) ensures that all occurrences of Δ are "skipped over" as if they were not there in the first place. However, the paths without the Δ labels are identical to the paths examined in Proposition 3.2.

The proof for $\mathbb{F}_{\mathbb{K}}$ [24] also holds in almost all the cases for the same reason (see Proposition 3.3). However the way LTS states are split by $\mathbb{K}$ facilitates the proof or Proposition 3.3, yet such a split no longer happens in $\mathbb{X}$. The original proof had $\mathbb{F}_{\mathbb{K}}(a; t) = a \wedge \mathsf{EX}\ \mathbb{F}_{\mathbb{K}}(t)$ but under $\mathbb{X}$ this construction will fail to work correctly on LTS such as

$p = a \Box b; p'$ such that p' may t. Indeed, $\mathbb{X}(p)$ features a node labeled Δ with two children; the first child is labeled a while the second child is not but has $\mathbb{X}(p')$ as an eventual descendant (through a possible chain of nodes labeled Δ). Clearly it is not the case that p may $a; t$, yet $\mathbb{X}(p) \models a \wedge \mathsf{EX}\ \mathbb{F}_{\mathbb{X}}(t)$, which shows that such a simple construction is not sufficient for $\mathbb{X}$.

To remedy this we set $\mathbb{F}_{\mathbb{X}}(a; t) = \mathsf{E}\ (a\ \wedge\ \mathsf{AX}\ \neg a)\ \mathsf{U}\ \mathbb{F}_{\mathbb{X}}(t) \wedge \mathsf{EX}\ \mathbb{F}_{\mathbb{X}}(t)$. The second term in the conjunction ensures that $\mathbb{F}_{\mathbb{X}}(t)$ will hold in some next state, while the first term specifies that a run of a will be followed by $\mathbb{F}_{\mathbb{X}}(t)$ (the $a\ \mathsf{U}\ \mathbb{F}_{\mathbb{X}}(t)$ component) and also that the run of a is exactly one state long (the $\mathsf{AX}\ \neg a$ part). Note in passing that the U operator is necessary in order to make sure that a and $\mathbb{F}_{\mathbb{X}}(t)$ are on the same path, for otherwise the example used above to show that the original $\mathbb{K}(p)$ does not work here will continue to be in effect.

The rest of the proof remains unchanged from the proof of Proposition 3.3. $\Box$

To illustrate the new equivalence relation we will re-use the exampled provided earlier [6, 24, 27]. The first is relatively uninteresting because the conversion algorithms is unchanged.

Example 1 COFFEE MACHINES AND THEIR TESTS [27]:

We consider the following coffee machines:

$$\begin{aligned} b_1 &= \text{coin}; (\text{tea} \Box \text{bang}; \text{coffee}) \quad \Box \quad \text{coin}; (\text{coffee} \Box \text{bang}; \text{tea}) \\ b_2 &= \text{coin}; (\text{tea} \Box \text{bang}; \text{tea}) \quad \Box \quad \text{coin}; (\text{coffee} \Box \text{bang}; \text{coffee}) \end{aligned}$$

The first machine accepts a coin and then dispenses either tea or coffee, at its discretion. Still, if one wants the other beverage, one just hits the machine. The second machine is rather stubborn, giving either tea or coffee at its discretion. By contrast with the first machines, the beverage offered will not be changed by hits.

The following CTL formula was found to differentiate between the two machines:

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

Following the same conversion process as used earlier [27] we obtain the following equivalent test:

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

Note however that the test that was originally proposed to differentiate between the two machines [17] was slightly different, namely:

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

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 $\mathbb{T}_{\mathbb{X}}(\phi)$ does not have a deadlock detection branch in the choice following coin; however, the only alternative to coffee in $\mathbb{T}_{\mathbb{X}}(\phi)$ is bang, which is precisely the same alternative as for t (as shown above). We thus conclude that t and $\mathbb{T}_{\mathbb{X}}(\phi)$ are equivalent. □

The conversion the other way around is a bit more complex and illustrated the modified X construct, as follows:

Example 2 COFFEE MACHINES AND THEIR LOGICAL PROPERTIES [6]:
Once more we have the same coffee machines:

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

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

As discussed earlier, one failure trace test that differentiate these machines is

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

The conversion of the failure trace test t in to a CTL formula (after elimination) will be:

$$\begin{aligned}\mathbb{F}_{\mathbb{X}}(t) \quad = \quad & \mathsf{E}(\text{coin} \wedge \mathsf{AX}\neg\text{coin}\ \mathsf{U}\ \text{coffee}) \wedge \mathsf{EX}(\text{coffee}) \vee \\ & \mathsf{E}(\text{coin} \wedge \mathsf{AX}\neg\text{coin}\ \mathsf{U}\ \text{bang}) \wedge \\ & \mathsf{EX}((\text{bang} \wedge \mathsf{AX}\neg\text{bang}\ \mathsf{U}\ \text{coffee}) \wedge \mathsf{EX}(\text{coffee}))\end{aligned}$$

This formula specifies that after a coin we can get a coffee immediately or after a bang we get a coffee immediately, The meaning of this formula is clearly equivalent to the meaning of t. As expected the formula $\mathbb{F}_{\mathbb{X}}(t)$ holds for b_1 but not for b_2. □

5.1 Converting Failure Trace Tests into Compact CTL Formulae, Revisited

Whenever all the runs of a test are finite then the conversion shown in Proposition 3.3 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. An algorithmic method of obtaining more compact (and more importantly, finite) formulae from tests with cycles was developed earlier [24] under the assumption that all the actions in a cycle are different from each other. We are able to improve this result by largely eliminating the restriction of distinct cycle actions as follows: Like before [24], we extend the function $\mathbb{F}$ to produce finite formulae out of tests with cycles, but this time we only require that a "first" action is marked in each cycle. The extension will work for both $\mathbb{F}_{\mathbb{K}}$ and $\mathbb{F}_{\mathbb{X}}$.

Theorem 5.2 *Let $\mathbb{F} \in \{\mathbb{F}_{\mathbb{K}}, \mathbb{F}_{\mathbb{X}}\}$. Then there exists an extension of $\mathbb{F}$ (denoted by abuse of notation $\mathbb{F}$ as well) such that $\mathbb{F} : \mathcal{T} \to \mathcal{F}$, p* may *t if and only if $\mathbb{K}(p) \models \mathbb{F}(t)$ for any $p \in \mathcal{P}$,* *and $\mathbb{F}(t)$ is finite for any test t provided that we are allowed to mark some entry action a so that we can refer to it as either a or* $\mathsf{start}(a)$ *in each loop of t. An entry action for a loop is defined as an action labeling an outgoing edge from a state that has an incoming edge from outside the loop.*

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 Proposition 3.3 or Theorem 5.1 (depending on whether $\mathbb{F}$ extends $\mathbb{F}_{\mathbb{K}}$ or $\mathbb{F}_{\mathbb{X}}$, respectively) for the rest of the test. Given the reliance on Proposition 3.3 or Theorem 5.1 we obtain overall an inductive construction. Nested loops in particular will be converted inductively (that is, from the innermost 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_i, $0 \leq i < n$, with the action a_0 marked as $\mathsf{start}(a_0)$. Each such an action a_i has the "exit" test t_i as an alternative. We make no assumption about the particular form of t_i, $0 \leq i < n$.

Given the intended use of our function, this proof will be done within the inductive assumptions of the proof of Proposition 3.3 or Theorem 5.1. 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:

$$\mathbb{F}(t) = \left(\mathsf{E}\left(\bigvee_{i=0}^{n-1} C_i\right) \mathsf{U} \left(\bigvee_{i=0}^{n-1} E_i\right)\right) \vee \mathbb{F}(t_0)$$

where C_i represents the cycle in its various stages and so

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

For convenience we also refer in what follows to $\bigvee_{i=0}^{n-1} C_i$ as C. Each E_i represents one possible exit from the cycle and so

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

with

$$\mathrm{count}(a_i) = \mathsf{EG}\, \mathrm{start}(a_{j_0}) \wedge \mathsf{EX}\ (a_{j_1} \wedge \cdots \mathsf{EX}\ a_{j_{i-1}})$$

where the sequence $(j_0, j_1, \ldots, j_{i-1})$ is the subsequence of $(0, 1, \ldots, i-1)$ that contains exactly all the indices p such that $a_p \neq \tau$. Note in passing that the formula above is applicable only for tests for which $a_i \neq \theta$ for all $0 \leq i < n$; however, θ in the cycle will be introduced below and so this general definition for count will prove useful later.

It is worth noting that the second term in the disjunction ($\mathbb{F}(t_0)$) accounts for the possibility that while running t we exit immediately upon entering the cycle through the exit test t_0, in effect without traversing any portion of the loop. Such a scenario is also pertinent to the proof for $\mathbb{F}_{\mathbb{K}}$ (Proposition 5.2), but was erroneously absent from the original proof [24] and so from the sketch proof of Proposition 5.2 shown erlier in this paper.

Like the name implies, the function of $\mathrm{count}(a_i)$ is like a counter for how many actions separate the test t_i from the marked action a_0. By counting actions we know what test is available to exit from the loop (depending on how many actions away we are from the start of the loop).

Intuitively, C specify the sequence of all the actions following each other in the loop, and always in the loop. It remains true for as long as the test goes around the loop, no matter where in the loop the test is. Next, E_i combines two parts. The first part counts how many actions we are away from the start of the loop. The second part $\mathsf{EX}\ \mathbb{F}(t_i)$ is the exit condition. The first part enables the right exit test t_i. Thus E_i will become true whenever the test t_i succeeds after i actions have been performed starting from $\mathrm{start}(a_0)$.

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 $t_{k-1 \bmod n}$ (Item 1 below), on the top level of the test $t_{k-1 \bmod n}$ but not as alternate a_k (Item 2), or both as a_k and on the top level of the alternate $t_{k-1 \bmod n}$ (Item 3). Given that θ only affects the top level of the choice in which it participates, these cases are exhaustive. Just like in the previous proof [24], we have:

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

2. If $\theta \in \mathrm{init}(t_{k-1 \bmod n})$ and $a_k \neq \theta$ then we change the exit formula $E_{k-1 \bmod n}$ so that it contains 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 $\mathrm{count}(a_{k-1 \bmod n}) \wedge \mathsf{EX}\ (\bigvee_{b\in\mathrm{init}(t_{k-1 \bmod n})\setminus\{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_{k-1 \bmod n})$. 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 $t_{k-1 \bmod n}$).

   We take the disjunction of the above variants and so $E_{k-1 \bmod n} = \mathrm{count}(a_{k-1 \bmod n}) \wedge$

$\mathsf{EX}(\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_{k-1 \bmod n}) \vee \neg\mathbb{F}(a_{(k)}) \wedge \neg(\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b))$ $\wedge\, \mathbb{F}(t_{k-1 \bmod n}(\theta))$.

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

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

   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 $E_{k-1 \bmod n} = \text{count}(a_{k-1 \bmod n}) \wedge \mathsf{EX}\ \bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b) \wedge \mathbb{F}(t_{k-1 \bmod n}) \vee \neg(\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_{k-1 \bmod n}(\theta))$. 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 $E_k = \text{count}(a_k) \wedge \mathsf{EX}\,\mathbb{F}(t_k)$. Now however $a_k = \theta$ and so we must apply to E_k the same process that we repeatedly performed earlier namely, adding $\neg(\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b))$. In addition, θ does not consume any input by definition, so the EX construction disappears. In all we have $E_k = \text{count}(a_{k-1}) \wedge \neg(\bigvee_{b \in \text{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_k)$ (when $a_k = \theta$, we do not need to count a_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 $p \overset{a_i}{\Longrightarrow} p_1 \wedge p_1 \overset{a_{i+1}}{\Longrightarrow} \wedge \cdots \wedge p_{n-1} \overset{a_{i+n-1}}{\Longrightarrow} p$, which shows that process p performs some actions in the cycle. We further notice that these are equivalent to $\mathbb{K}(p) \models a_i$ and $\mathbb{K}(p_1) \models a_{i+1} \wedge \cdots \wedge \mathbb{K}(p_{n-1}) \models a_{i+n-1}$, respectively. Therefore $p \overset{a_i}{\Longrightarrow} p_1 \wedge p_1 \overset{a_{i+1}}{\Longrightarrow} \wedge \cdots \wedge p_{n-1} \overset{a_{i+n-1}}{\Longrightarrow} p$ iff $\mathbb{K}(p) \models \mathsf{EG}(\mathbb{F}(a_i) \wedge$

$\mathsf{EX}(\mathbb{F}(a_{(i+1) \bmod n}) \wedge \mathsf{EX} \cdots \wedge \mathsf{EX}(\mathbb{F}(a_{(i+n-1) \bmod n})) \cdots))$. That is,

$$p \overset{a_i}{\Longrightarrow} p_1 \wedge p_1 \overset{a_{i+1}}{\Longrightarrow} \wedge \cdots \wedge p_{n-1} \overset{a_{i+n-1}}{\Longrightarrow} p \text{ iff } \mathbb{K}(p) \models C_i \tag{5.1}$$

We exit from the cycle as follows: When $p \overset{a_i}{\Longrightarrow} p' \overset{a_{i+1}}{\not\Longrightarrow}$ 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.

We now note that $p \overset{a_i}{\Longrightarrow} p' \wedge p'$may t_i is equivalent to $\mathbb{K}(p) \models \text{count}(a_i) \wedge \mathsf{EX}(\mathbb{F}(t_i))$. Indeed, when p and t perform a_i the test t is i actions away from start(a_0) and so count(a_i) is true. Therefore given the inductive hypothesis (that p' may t_i iff $\mathbb{K}(p') \models \mathbb{F}(t_i)$ for any process p') we concluded that:

$$(p \overset{a_i}{\Longrightarrow} p' \wedge p' \text{ may } t_i) \text{ iff } \mathbb{K}(p) \models (\text{count}(a_i) \wedge \mathsf{EX}(\mathbb{F}(t_i))) = E_i \tag{5.2}$$

Taking the disjunction of Relations (5.2) over all $0 \leq i < n$ we have

$$(p \overset{a_i}{\Longrightarrow} p' \wedge p' \text{ may } \Sigma_{i=0}^{n-1} t_i) \text{ iff } \mathbb{K}(p) \models \bigvee_{i=0}^{n-1} E_i \tag{5.3}$$

The correctness of $\mathbb{F}(t)$ is then the direct consequences of Relations (5.1) and (5.3): We can stay in the cycle as long as one C_i remains true (Relation (5.1)), and we can exit at any time using the appropriate exit test (Relation (5.3)).

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

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 $p'\|_\theta t' \overset{a_{k-1 \bmod n}}{\longrightarrow} p''\|_\theta t''$. Let $b \in \text{init}(t_{k-1 \bmod n})$. If $p''\|_\theta t'' \xrightarrow{b}$ then

the run must exit the cycle according to the definition of $\|_\theta$. At the same time C_k is false because the disjunction $\bigvee_{b \in \mathrm{init}(t_{k-1 \bmod n})} \mathbb{F}(b)$ 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 $b \in \mathrm{init}(t_{k-1 \bmod n})$, 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.

2. Now $\theta \in \mathrm{init}(t_{k-1 \bmod n})$ 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 $t_{k-1 \bmod n}$. First, $p \stackrel{a_{k-1 \bmod n}}{\Longrightarrow} p' \wedge \exists b \in \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\} : p' \stackrel{b}{\Longrightarrow} \wedge\ p'$ may $t_{k-1 \bmod n}$. Second, $p \stackrel{a_{k-1 \bmod n}}{\Longrightarrow} p' \wedge \neg(\exists b \in \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\} : p' \stackrel{b}{\Longrightarrow}) \wedge p' \stackrel{a_k}{\not\Longrightarrow} \wedge\ p'$ 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 $t_{k-1 \bmod n}$). 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 $\mathbb{K}(p') \models \mathbb{F}(t_i)$ for any process p') we have

$$\begin{aligned} & p \stackrel{a_{k-1 \bmod n}}{\Longrightarrow} p' \wedge \exists b \in \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\} : p' \stackrel{b}{\Longrightarrow} \wedge p' \text{ may } t_{k-1 \bmod n} \text{ iff} \\ & \mathbb{K}(p) \models \mathrm{count}(a_{k-1 \bmod n}) \wedge \mathsf{EX} \bigvee\nolimits_{b \in \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b) \wedge \mathbb{F}(t_{k-1 \bmod n}) \end{aligned} \tag{5.4}$$

$$\begin{aligned} p \overset{a_{k-1 \bmod n}}{\Longrightarrow} p' \wedge \neg(\exists b \in \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\} : p' \overset{b}{\Longrightarrow}) \wedge p' \overset{a_k}{\not\Longrightarrow} \wedge \\ p' \text{ may } t_{k-1 \bmod n}(\theta) \text{ iff } \mathbb{K}(p) \models \mathrm{count}(a_{k-1 \bmod n}) \wedge \mathsf{EX}\, \neg\mathbb{F}(a_{(k)}) \wedge \\ \neg(\textstyle\bigvee_{b \in \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_{k-1 \bmod n}(\theta)) \end{aligned} \tag{5.5}$$

The conjunction of Relations (5.4) and (5.5) 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).

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

   Suppose first that $p' \overset{b}{\Longrightarrow}$ for some $b \in \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\}$. 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 $\mathbb{K}(p') \models \bigvee_{b \in \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b) \wedge \mathbb{F}(t_{k-1 \bmod n})$. That C_k is false happens because $\neg(\bigvee_{b \in \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b))$ 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 $\mathbb{K}(p') \models \mathbb{F}(t_{k-1 \bmod n})$ by inductive hypothesis).

   Suppose now that $p' \overset{b}{\not\Longrightarrow}$ for any $b \in \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\}$. Then the only possible continuations are (a) p' remaining in the cycle which is equivalent to C_k being true (ensured by $\bigvee_{b \in \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)$ being false), or (b) p' taking the θ branch of $t_{k-1 \bmod n}$, which is equivalent to $\neg(\bigvee_{b \in \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)) \wedge \mathbb{F}(t_k(\theta))$ by the fact that $\bigvee_{b \in \mathrm{init}(t_{k-1 \bmod n}) \setminus \{\theta\}} \mathbb{F}(b)$ 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.

Again taking the disjunction of the two alternatives above establishes this case. □

5.2 Example Generation of Compact CTL Formulae

In this section we illustrate the conversion of failure trace test into CTL formulae. For this purpose consider the following simple vending machines, also shown graphically in Figure 5.1(a, b):

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

These two machines dispense a coffee after accepting a coin; the first machine dispenses a tea after second coin, whereas the second still dispenses coffee. After two coins and a hit by customer, the two machines will dispense tea.

The Kripke structures equivalent to the two machines and constructed according to Theorem 4.1 are shown in Figure 5.1(c, d), respectively.

Consider now the following test:

$$t = \text{coin}; (\text{coffee}; \text{pass} \;\square\; \text{coin}; (\text{tea}; \text{pass} \;\square\; \text{bang}; (\text{tea}; \text{pass} \;\square\; t)))$$

Using Theorem 5.2 (and thus implicitly Theorem 5.1) we can convert this test into the following CTL formula (after eliminating all the obviously true sub-formulae):

$$\begin{aligned} \mathbb{F}_{\mathbb{X}}(t) \;=\; & (\mathsf{EG}\ (\text{coin}) \land \mathsf{EX}(\text{coin} \land \mathsf{EX}((\text{bang}) \land \mathsf{EX}(\text{coin})))\lor \\ & \mathsf{EG}\ (\text{coin}) \land \mathsf{EX}(\text{bang} \land \mathsf{EX}((\text{coin}) \land \mathsf{EX}(\text{coin})))\lor \\ & \mathsf{EG}\ (\text{bang}) \land \mathsf{EX}(\text{coin} \land \mathsf{EX}((\text{coin}) \land \mathsf{EX}(\text{bang}))) \\ & \mathsf{U}\ \mathsf{EG}(\text{init coin}) \land \mathsf{EX}\ (\text{coffee}) \lor \mathsf{EG}(\text{init coin}) \\ & \land \mathsf{EX}(\text{coin} \land \mathsf{EX}(\text{tea})) \lor \mathsf{EG}(\text{init coin}) \land \mathsf{EX}(\text{coin} \land \mathsf{EX}(\text{bang} \land \mathsf{EX}(\text{tea})))) \\ & \lor \mathsf{EG}(\text{tea}) \end{aligned}$$

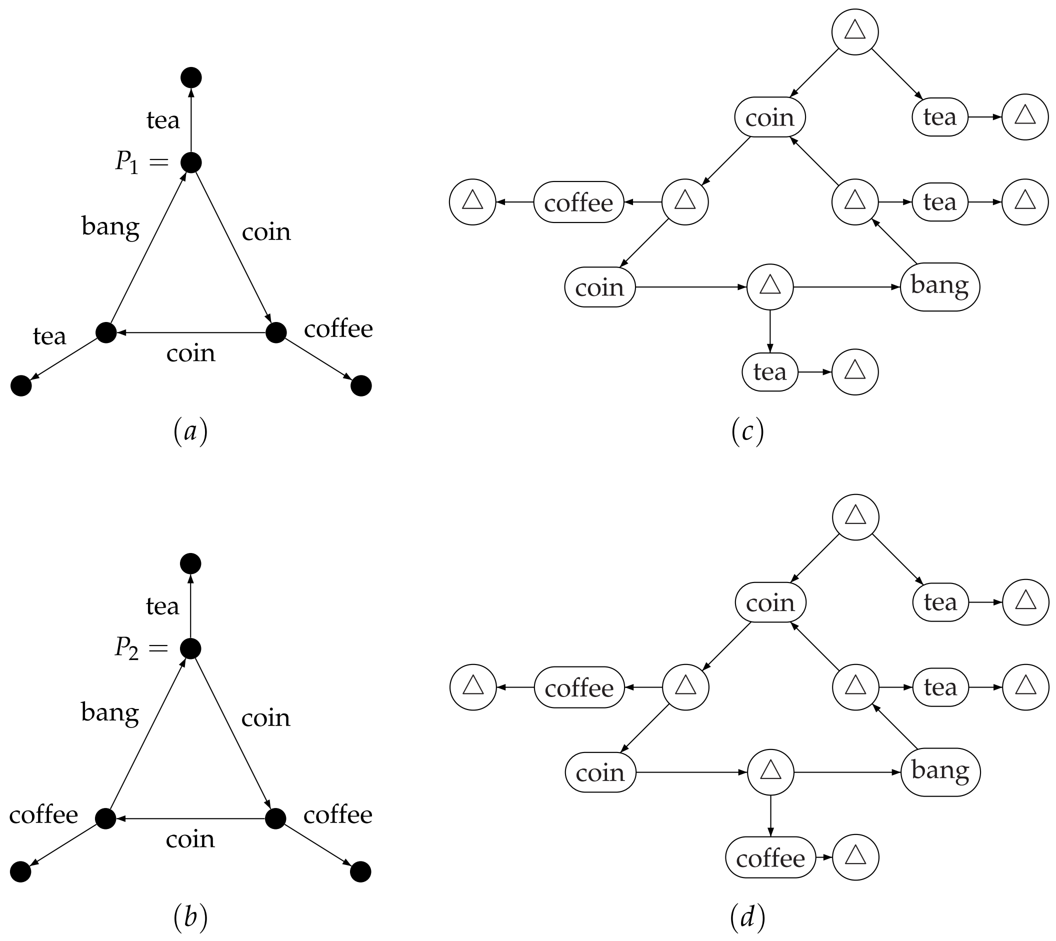

Figure 5.1: Two vending machines P_1 and P_2 (a, b) and their equivalent Kripke structures $\mathbb{X}(P_1)$ and $\mathbb{X}(P_2)$ (c, d).

It is not difficult to see that the meaning of this formula is equivalent to the meaning of t. Indeed, the following is true for both the test t as well as the formula $\mathbb{F}_{\mathbb{X}}(t)$: tea is offered in the first step without any coin, or after a coin coffee is offered, or after two coins tea is offered, or after two coins and customer hits tea is offered; On the other hand after two coins and a bang, and if no coin is available next, then tea will be offered. Finally, after two coins and a bang if both coin and tea are available, then two options comes out: we can either keep in the cycle, or offer a tea. In all the process can remain in the cycle indefinitely or can exit from the cycle and pass the test. We can get coffee or tea at the first cycle or after few repetitions of the cycle.

The formula $\mathbb{F}_{\mathbb{X}}(t)$ holds for $\mathbb{X}(P_1)$, but it does not hold for $\mathbb{X}(P_2)$.

Chapter 6

Conclusions

The contribution of this thesis can be summarized as follows: First we offered a new algorithmic function $\mathbb{X}$ that converts an LTS into its equivalent Kripke structure (Theorem 4.1) and then we showed that CTL and failure trace tests continue to be equivalent under this new conversion (Theorem 5.1).

A previous paper [24] also noted that the function for converting failure trace tests into CTL formulae produces infinite formulae whenever the test being converted features potentially infinite runs (or loops). In that paper, the loops have been converted into finite CTL formulae under the assumption that all the actions in the loop are different from each other. We now eliminate the need for this assumption and we show that compact formulae can be obtained from loops under the weaker assumption that we can mark one action in the loop but otherwise not imposing any restrictions on the loop actions (Theorem 5.2). This conversion works under both the old conversion function from LTS to Kripke structures (Section 3.1 [6]) and out new conversion (Chapter 4). Between other things, we thus solve one of the open problem noted in the previous paper [24].

Another open problem mentioned earlier [6] is that $\mathbb{K}(p)$ is a Kripke structure that may have multiple initial states, which in turn requires the use of a weaker satisfaction operator (Definition 3.2). In order to solve this problem we adapted a construction used earlier in a different context [13] to the task of converting LTS into equivalent Kripke structures and so develop a new conversion function $\mathbb{X}$ which avoids the issue of multiple initial states. The drawback of this conversion however is that the resulting Kripke structure is considerably larger.

As we mentioned above, the generation of compact CTL formulae out of loop tests relies on marking an initial action in each loop. Whether it is possible to effect such a conversion without supplementary markers at all remains an open problem.

In all, this thesis contributes to framework for combined, algebraic and logic formal specifications, which allows the development of mixed specifications for reasons of convenience or even personal taste. Most importantly, no matter which form the specification takes (logic, algebraic, or mixed), it can now be used in both model checking and model-based testing (or even both!).

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 C. Dai, *Timed test generation based on timed temporal logic*, in Proceedings of the 11th International Conference on Automation and Information (ICAI 10), Iasi, Romania, June 2010, pp. 15–20.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] R. DE NICOLA AND F. VAANDRAGER, *Action versus state based logics for transition systems*, Lecture Notes in Computer Science, 469 (1990), pp. 407–419.

[14] ——, *Three logics for branching bisimulation*, Journal of the ACM, 42 (1995), pp. 438–487.

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

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

[17] 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.

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

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

[20] 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.

[21] 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.

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

[23] 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.

[24] S. SINGH, *On the equivalence between computation tree logic and failure trace testing*, Master's thesis, Bishop's University, Aug. 2017.

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

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

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