Formal Verification of abstract digital controllers for hybrid systems

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

by

Santhosh Prabhu M

10CS60R31

under the supervision of

Prof. Pallab Dasgupta

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

India

April 2012

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
WB 721302, India.

Certificate

This is to certify that the thesis entitled **Formal Verification of abstract digital controllers for hybrid systems**, submitted by **Santhosh Prabhu M**, Roll no: **10CS60R31**, in the *Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, India*, for the award of the degree of **Master of Technology**, is a record of original research work carried out by him under my supervision and guidance. The thesis fulfills all the requirements as per the regulations of this institute. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Prof. Pallab Dasgupta

Acknowledgements

I am thankful to the almighty for showering his blessings upon me, without which this project would have never reached completion.

I am also deeply indebted to my guide, Prof. Pallab Dasgupta for his support and constant motivation. His vast knowledge and able guidance helped me throughout this project, taking me past each and every obstacle. His guidance was certainly one of the most important reasons behind the successful completion of this project.

I am immensely thankful to Debjit Pal and other members of the Formal Verification group, IIT Kharagpur, for their help in completing this project.

I also wish to thank all faculty and staff of the Department of Computer Science, IIT Kharagpur for their support.

My gratitude to my parents and my friends - Sai Sambhu J and S Harikrishnan in particular - is beyond words. They have been unwavering in their support to me, and have helped me dedicate my best efforts towards the success of this project. I thank them profusely for their love and support.

Santhosh Prabhu M

Abstract

This project proposes extensions of formal verification techniques, so as to bring finite state controllers of hybrid systems within the ambit of the verification framework. These controllers are represented using Kripke Structures whose states are labelled with Predicates Over Real Variables(PORVs). An extension to LTL is proposed, for representing the properties, and two model checking approaches - one automata theoretic and the other symbolic - are proposed for verifying such properties over PORV labelled Kripke Structures. The automata theoretic approach is based on the existing on-the-fly verification technique. Using SMT solvers, the symbolic approach reduces the verification problem into a usual model checking problem with propositions, which can be solved using industrial model checking tools. This project also addresses the problem of model checking under assume properties which may themselves contain PORVs. Correctness of the proposed model checking approaches are proven, and case studies are presented for illustrating the toolflow.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Formal verification refers to the process of verifying and proving by formal techniques that a given system satisfies a certain property desired of it. Over the years, formal verification has proven to be immensely helpful in unearthing design and implementation flaws that are very difficult to be located by traditional testing techniques. It has grown to be an indispensable tool in the hardware industry, for ensuring correct designs of ever increasing complexity, and its obvious benefits have prompted its acceptance in the software industry also. Today, it is being used for verifying a wide variety of systems, ranging from priority arbiters to operating system kernels.

The most attractive feature of formal verification is exhaustivity. Formal verification tools analyze the system under verification and come up with either a proof of correctness, or give a counterexample, which is essentially a possible scenario where the system violates the property being verified. The counterexamples are particularly useful to design engineers, since they help them know where exactly the system fails the requirement and thus facilitate easy modification. Due to this reason, significant efforts are being made by formal verification researchers to develop techniques that provide more and more realistic counterexamples.

Various techniques have been developed for carrying out the verification process quickly and correctly, overcoming the often huge complexities of the system under verification. Probably the most popular approach to verification is *model checking*[4], where the process of verification usually involves representing the system as a transition system, and algorithmically checking whether the transition system is a model for the property being verified. Different formalisms, like *Linear Temporal Logic*[11] (LTL), *Computation Tree Logic* [6](CTL) etc. have been developed for expressing the properties to be verified. Verification techniques, and industrial tools based on them have also been developed for verification of various types of systems against such properties[8, 3, 10].

In the industry, digital circuits domain is where formal verification has found widest acceptance. Possibly due to this reason, highly scalable techniques have been developed for verification of digital circuits. Verification of such circuits involves representing the system as a finite state machine(FSM), and model checking the FSM against the property to be verified. The current methodologies for model checking fall into two broad categories[5]:

- *Automata theoretic approaches*

  Automata theoretic model checking [17, 7] proceeds by performing a depth first traversal of the state space of the system, looking for runs that refute the property. This approach models the negation of the property to be verified in the form of a special kind of automaton known as *Linear Weak Alternating Automaton*(LWAA), and checks for common runs between the system and the LWAA. The biggest advantage of this approach is that it doesn't require the entire state space to be present in memory at the same time.

- *Tableau based approaches*

  In tableau based techniques[4], the negation of the property is modelled as a closed automaton, known as the tableau. The tableau contains all those paths that satisfy the negation of the property. Then, the product of the model of the system with the tableau is computed, and the product is checked for emptiness. In this approach, the state space is explored in a non-explicit manner, and hence, this approach is also known as the *symbolic model checking* approach.

The success of formal verification in the digital circuits domain has encouraged researchers to come up with verification techniques for other domains as well, one of the prominent ones being the hybrid systems domain. A hybrid system differs from a purely digital system in that, in addition to discrete behaviours, the systems has certain continuous dynamics as well. The verification of such systems is typically carried out by modelling them in the form of hybrid automata[16, 1, 2]. Hybrid automata have a set of discrete locations, and a set of real variables. The variables evolve in a continuous fashion in each discrete location, with the manner of evolution defined by a location-specific activation function. Significant amount of work has been carried out in the field of hybrid system verification, and there are quite a few powerful tools that can carry out the verification in an automated fashion [13].

1.1 Motivation and Problem Description

When digital controllers are developed for controlling environments(plants) with real valued quantities, the development proceeds through two distinct phases: The *modelling phase* during which the control strategy and the specification of the controller is finalized, and the *synthesis phase* when the controller is implemented. The specification, and later the implementation, operate by identifying certain threshold crossings over the various real valued quantities and performing certain actions whenever the thresholds are crossed. For example, consider a controller that controls the water pump in a steam boiler.

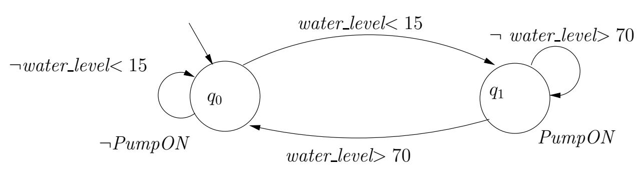


Figure 1.1: Controller for water pump in a steam boiler

The specification for the controller could be as follows:

1. P1: If the water level is below 10, then the water pump must be ON in the next time instance.

2. P2: If the water level is above 75, the water pump must be OFF in the next time instance.

Given the specification, the implementation is built, using the specification as a reference. However, controller implementations base their actions on more conservative thresholds than the ones in the specifications, so as to robustly satisfy the specifications. For instance, a possible implementation of the water pump controller could be as shown in Figure 1.1.

The thresholds that the specification and the implementation look for can be expressed in the form of *Predicates Over Real Variables*(PORVs). Since the properties that one wishes to verify are obtained from the specification, formalisms are needed, by which properties over PORVs can be expressed. As shall be discussed later in greater detail, an extension of LTL is used, that can express properties(often referred to as *assertions*) over PORVs in addition to propositions. Using this extension, it is possible, for instance, write the properties for the pump controller as follows:

$$P1: \qquad G((water\_level < 10) \Rightarrow XPumpON)$$

$$P2: \qquad G((water\_level > 75) \Rightarrow X\neg PumpON)$$

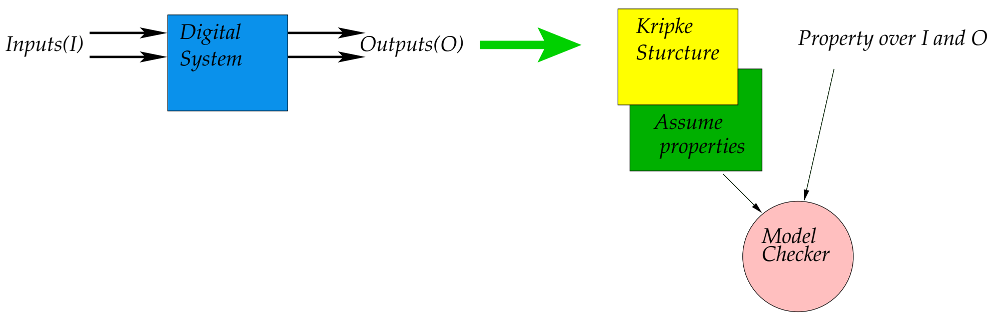
It may be noted that the assertions, being derived out of the specification, use PORVs that are different from the ones that appear in the implementation. For formally verifying properties over implementations of such controllers, one may choose between the following options:

1. Perform a closed loop simulation with the controller and the plant together, using hybrid system verification techniques, or,

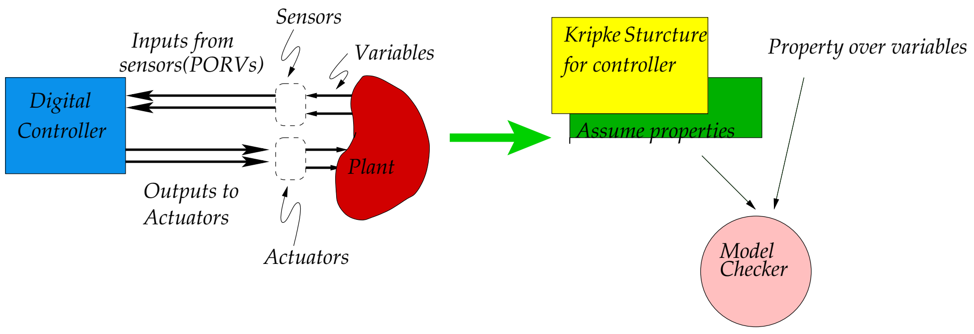
2. Come up with techniques that will enable open loop verification of the controller alone.

This work describes techniques for open loop verification of such controllers, thereby escaping the large computational complexities of hybrid system verification. The controller under verification is modelled as a Kripke Structure, whose states will be labelled with truth assignments to both PORVs and propositions. Properties are to be written in the extended version of LTL, which allows the use of PORVs(possibly different from the PORVs that appear in the Kripke Structure) just like propositions in LTL. By developing techniques for model checking such properties over PORV labelled Kripke Structures, the intention is to put in place mechanisms for verifying controllers like the one in Figure 1.1.
In verification parlance, assume properties are properties that one can expect the system under verification and its environment to satisfy. Assume properties are of interest due to 2 main reasons:

1. In real world systems, it is only necessary that the property under verification be satisfied under realistically possible conditions. Many of the counterexamples that verification process may throw up are often unrealistic, and hence not worthy of attention. Verification under assume properties attempts to find counterexamples that may actually occur in the real world operation of the system.



(a) Traditional verification



(b) New approach

Figure 1.2: Comparison of traditional and the new verification processes

2. Often, the property to be verified is not on the controller alone, but on the controller and the plant together. For example, in the steam boiler example, one could wish to verify the property that the water level always stays within certain levels. These properties cannot be verified through open loop verification of the controller alone, since it is necessary to know how the plant will behave in response to various controller actions. Assume properties can be used as a model of the plant, to verify such properties.

1.2 Summary of Contributions

Figure 1.2 shows a comparison between the traditional verification of digital circuits and the process presented in this thesis. The contribution of this project is the development of methods for carrying out the entire verification paradigm as shown in Figure 1.2(b). For this, the following issues have been addressed:

- Extension of LTL for defining properties over PORVs. A new extended version of LTL has been proposed, the formulae in which are interpreted over real valued traces(explained later in greater detail).
- Development of automata theoretic model checking techniques for verification of properties in the extended logic. An approach based on the existing *on-the-fly* approach[7] has been proposed, and proven to be correct.
- Development of tableau based model checking techniques for verification. A technique is proposed, which uses SMT solvers to reduce the verification problem with PORVs into the standard model checking problem, that can be handled by industrial model checkers. The correctness of the proposed approach has been proven, and a tool kit has been implemented, which can be used for the verification of industrial controllers that control environments with continuously evolving variables. The tool kit has been used successfully for verifying some real-world digital controllers, and the results are presented in this thesis.
- Development of techniques to handle assume properties. The tableau based approach has been implemented in such a way that it allows the use of assume properties in the verification process. Like assertions, the assume properties are also written in the extended version of LTL, and can have PORVs different from the ones used by the controller.

1.3 Organization of this report

The rest of this document is organized as follows:

- Chapter 2 presents some background material that is required for better understanding of the rest of thesis.
- Chapter 3 lays down formalisms used for constructing the various model checking algorithms.
- Chapter 4 presents the automata theoretic approach is presented, including the proof of correctness.
- Chapter 5 describes the tableau based approach, and gives the proof of correctness. The technique for verification with assume properties is also presented in this chapter.
- Chapter 6 describes a few case studies, and experimental results.
- Chapter 7 presents the conclusion and the future course of work.

Chapter 2

Background

This project deals with development of techniques for specifying properties and verifying them over digital controllers. For building these techniques, some established notions and results from the verification domain are leveraged. These are presented now.

2.1 Linear Temporal Logic

Linear Temporal Logic is a modal temporal logic, where modalities refer to time. LTL formulae make it possible to specify properties like *"The grant signal is eventually made high"*, *"The request signal is held high until the grant signal is made high"* etc. and hence are used for describing the future states of a system. LTL formulae include propositional formulae which describe the static state of a system, and also temporal formulae constructed using temporal operators X (next)and U (until), along with the propositional operators $\neg$ and $\wedge$. These temporal formulae describe the behaviour of the system over all possible sequences of states starting with the current state. Syntax of LTL formulae is defined below.

Definition 1 (LTL Syntax)**.** *Let $\mathcal{P}$ be a non-empty set of atomic propositions. Then, the set of well-formed LTL formulae over $\mathcal{P}$ is inductively defined as* *follows:*

1. *If $p \in \mathcal{P}$, then p is a well formed formula.*

2. *If ϕ and ψ are well formed formulae, then $\neg\phi, \phi \wedge \psi, \phi U \psi$ and $X\phi$ are well formed formulae.* $\square$

LTL formulae are interpreted over an infinite temporal structure $\sigma = s_0 s_1 ... \in (2^{\mathcal{P}})^{\omega}$. The structure $\sigma = s_0 s_1 ...$ is said to satisfy the formula ϕ, represented by $\sigma \models \phi$ under the following conditions:

- $\sigma \models p$ for $p \in \mathcal{P}$ iff $p \in s_0$.
- $\sigma \models \phi \wedge \psi$ iff $\sigma \models \phi$ and $\sigma \models \psi$.
- $\sigma \models \neg\phi$ iff $\sigma \not\models \phi$.
- $\sigma \models X\,\phi$ iff $\sigma|_1 \models \phi$.
- $\sigma \models \phi\ U\,\psi$ iff for some $i \in \mathbb{N}, \sigma|_i \models \psi$ and for all $j{<}i$, $\sigma|_j \models \phi$.

Here, $\sigma|_i$ denotes the suffix $s_i s_{i+1} ...$ of σ from state s_i. In addition to the operators discussed above, for concisely writing LTL formulae, $\top$, $\bot$, Boolean connectives $\vee$, $\Rightarrow$, $\Leftrightarrow$ and additional temporal operators F (future), G (globally), and V (release) are also used. The temporal operators are defined as follows:

1. $F\,\phi \equiv \top U \phi$
2. $G\,\phi \equiv \neg F \neg \phi$
3. $\phi\ V\,\psi \equiv \neg(\neg\phi U \neg\psi)$

By using temporal operators X, U and V and Boolean operators $\neg, \wedge$ and $\vee$, any given LTL formula can be converted into negation normal form, where negation is applied to only propositions. So, for this discussion, it is safe to assume that the LTL formulae are given in negation normal form.

2.2 Some notions in automata theory

This section describes certain concepts of automata theory, which find application in the verification of LTL and LTL-like properties. Special emphasis is laid on the notion of alternation and *Linear Weak Alternating Automata*(LWAA) because, as shall be discussed shortly, given a formula to be verified, there exist procedures for representing it(or its negation) in the form of an LWAA, which can then be used for easily model checking the property over the system under verification.

2.2.1 Acceptance of infinite words

Given a finite non-empty alphabet Σ, a finite word can be defined as a finite sequence $a_0a_1...a_n$ of symbols, where $a_i \in \Sigma$ for $0 \leq i \leq n$. The set of such strings is represented by Σ^*. An infinite word on the other hand is an infinite sequence of symbols $a_0a_1...$, where $a_i \in \Sigma$. The set of such strings is represented Σ^ω. Acceptance of finite words by a finite automaton is fairly straightforward and well understood. However, in the case of infinite words, different conventions exist on when a word can be considered to be accepted by a finite automaton. In this discussion, the *Co-Büchi* acceptance condition shall be used. The definition of a non-deterministic co-Büchi automaton is as follows:

Definition 2 (Non-deterministic co-Büchi automaton). *A Non-deterministic co-Büchi automaton is a tuple (Σ, S, s^0, ρ, F), where Σ is a finite non-empty alphabet, S is a finite non-empty set of states, $s^0 \in S$ is an initial state, $F \subseteq S$ is the set of final states, and $\rho : S \times \Sigma \rightarrow 2^S$ is a transition function.* □

A run r of a Non-deterministic co-Büchi automaton A on an infinite word $w = a_0a_1...$ over Σ is a sequence $s_0, s_1...$, where $s_0 = s^0$ and $s_{i+1} \in \rho(s_i, a_i)$ for all $i \geq 0$. Let $\text{Inf}(r)$ represent the set of states that occur in r infinitely often (at least one state will certainly occur infinitely often, since the run is infinite, and there are only finite number of states). r is said to be accepting if there is no final state that occurs infinitely often in r, i.e., $\mathrm{Inf}(r) \cap F = \emptyset$. An infinite word w is said to be accepted by A if there is an accepting run of A on w. $\mathcal{L}(A)$ denotes the set of infinite words accepted by A.

2.2.2 Alternation and LWAA

Non-determinism in a model of computation represents the power of existential branching. That is, given an input string, it can be said to be accepted by a non-deterministic automaton if there exists at least one accepting run in the automaton for that string. In contrast, alternation represents the power of the dual of existential branching that is, universal branching. Presented here is a brief discussion on the concept of alternating automata. Later, the use of alternating automata in the verification framework shall be discussed in detail. For a given set X, let $\mathcal{B}^+(X)$ represent the set of positive Boolean formulae over X(i.e., formulae constructed using elements of X, $\wedge$, $\vee$ and formulae **true** and **false**). $Y \subseteq X$ is said to satisfy a formula $\theta \in \mathcal{B}^+$ if a truth assignment that assigns *true* to the elements of Y and *false* to the elements of $X\backslash\ Y$ makes θ evaluate to *true*. For instance, $\{s_1, s_2\}$ satisfies $s_1 \vee s_3$ while $\{s_2\}$ doesn't.
In a non deterministic automaton $A = (\Sigma,\ S,\ s^0,\ \rho,\ F)$, the transition function ρ maps a state $s \in S$ and an input symbol $a \in \Sigma$ to a set of states (a subset of S). This transition function can be represented using $\mathcal{B}^+(S)$. For instance, $\rho(s,a)$=$\{s_1, s_2, s_3\}$ can be written as $\rho(s,a)$=$s_1 \vee s_2 \vee s_3$. It is obvious that, for non-deterministic automata, the formulae that represent the transition function cannot use Boolean $\wedge$, since from any state, the automata can transit only to one of the possibly many next states. In an alternating automata, $\rho(s,a)$ can be an arbitrary formula from $\mathcal{B}^+(S)$. For instance,

$$\rho(s, a) = (s_1 \wedge s_2) \vee (s_2 \wedge s_3)$$

is valid, meaning that the automaton accepts aw where $a \in \Sigma$ and w is a word, from state s if it accepts w from both s_1 and s_2 or from both s_2 and s_3. Thus, disjunctions represent existential choice whereas conjunctions represent universal choice. Accepting runs of alternating automata are represented by trees (unlike sequences of states in the case of non-deterministic automata without alternation), due to universal branching.

Bringing in co-Büchi acceptance in into alternating automata makes it possible to design alternating automata that accept infinite words over the alphabet. Naturally, in these automata, accepting runs are represented by infinite trees in which along no path does a final state occur infinitely often. Within this class of alternating automata with co-Büchi acceptance, a specific subclass is of particular interest to the verification community. Known as (Co-Büchi) Linear Weak Alternating Automata, these automata are alternating automata that accept infinite words according to the co-Büchi acceptance condition, and also satisfy an additional interesting property, which is as follows. Let $\preceq$ denote the reachability relation on the set of states of the automata, i.e., for $s, s' \in S$, $s' \preceq s$ iff s' can be activated by taking zero or more transitions from s. For an automaton $\mathcal{A}$ to be called a (Co-Büchi)LWAA, it is required that, in addition to $\mathcal{A}$ being an alternating automaton that satisfies the co-Büchi acceptance condition, the relation $\preceq$ is a partial order on the set of states of $\mathcal{A}$.

Intuitively, this property means that the hypergraph of transitions of the automata does not contain any cycles other than self loops, and hence, every infinite path in the accepting run should eventually remain stable at some location s, and the co-Büchi acceptance requires that $s \notin F$.

Chapter 3

Problem Formulation

Before applying any kind of formal verification techniques, one must have in place formalisms using which the property, and the system under verification can be described in a non-ambiguous fashion. This chapter describes the various notations used in this project, their syntax, semantics etc.

3.1 Controllers as Kripke Structures

Definition 3 (Predicate Over Real Variables)**.** *A Predicate Over a set* var$=\{x_1, x_2, \ldots x_n\}$ *of Real Variables(PORV) is defined by a tuple* $\langle \alpha_1, \alpha_2 \ldots \alpha_{n+1} \rangle \in \mathbb{R}^{n+1}$ *and a relational operator* $\star \in \{\leq, <\}$. *The PORV represents the inequality* $\alpha_1 x_1 + \alpha_2 x_2 + \ldots \alpha_n x_n + \alpha_{n+1} \star 0$. □

A controller for a hybrid system can be defined as a tuple:

$$\mathcal{G} = \langle Q, I, \delta, Q_0, \mathcal{AP}, var, \mathcal{L} \rangle$$

where:

- Q is the set of states of the controller,
- $Q_0 \subseteq Q$ is the set of initial states,

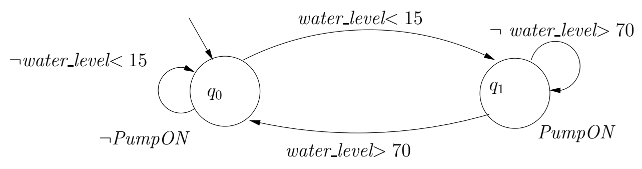
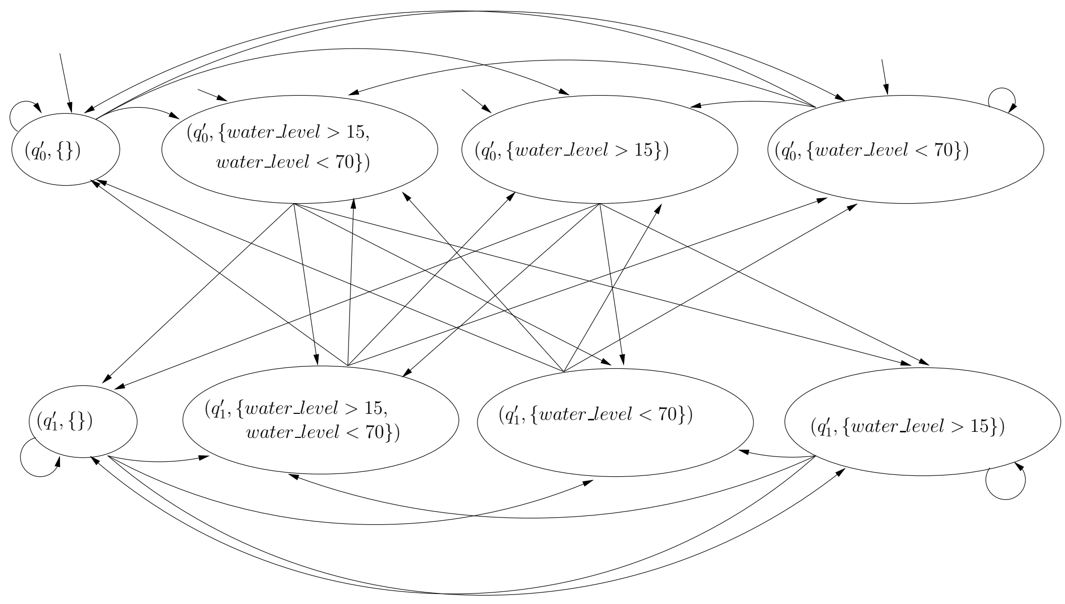


Figure 3.1: Water pump controller for a steam boiler (Reproduction of Figure 1.1)

- $\mathcal{AP}$ is a set of atomic propositions (labels),
- var is a set of real valued variables,
- $I = I_B \cup I_{PORV}$ is the set of inputs of the controller, where I_B is a set of Boolean signals and I_{PORV} is a set of PORVs over var,
- $\delta \subseteq Q \times 2^I \times Q$ is the transition relation,
- $\mathcal{L} : Q \to 2^{\mathcal{AP}}$ is a function for labelling the states in Q with propositions in $\mathcal{AP}$

For example, the constituents of the tuple corresponding to the controller for controlling the water pump in a steam boiler, shown in Figure 3.1 can be defined as follows:

- $Q = \{S_0, S_1\}$
- $Q_0 = \{S_0\}$
- $\mathcal{AP} = \{PumpON\}$
- $var = \{water\_level\}$
- $I_B = \emptyset$ and I_{PORV}=$\{water\_level < 15$, $water\_level > 70\}$
- δ is as illustrated in Figure 1.1,
- $\mathcal{L}(S_0) = \emptyset$ and $\mathcal{L}(S_1) = \{PumpON\}$



$(q_0', \{\})$
$(q_0', \{water\_level > 15, water\_level < 70\})$
$(q_0', \{water\_level > 15\})$
$(q_0', \{water\_level < 70\})$
$(q_1', \{\})$
$(q_1', \{water\_level > 15, water\_level < 70\})$
$(q_1', \{water\_level < 70\})$
$(q_1', \{water\_level > 15\})$

Figure 3.2: Kripke Structure for the water pump controller

Kripke Structure $\mathcal{M}$ equivalent to the controller $\mathcal{G} = \langle Q, I, \delta, Q_0, \mathcal{AP}, var, \mathcal{L}\rangle$ can be defined as follows :

$$\mathcal{M} = \langle Q', \delta', Q_0', \mathcal{AP}', \mathcal{L}'\rangle$$

where:

- $Q' = Q \times 2^I$. A state $q_i' \in Q'$ is a pair $\langle q_i, a_i\rangle$, where $q_i \in Q$ and $a_i \in 2^I$,
- $Q_0' = Q_0 \times 2^I$,
- $\delta' \subseteq Q' \times Q'$ is the transition relation, such that $(q_i', q_j') \in \delta'$ iff $(q_i, a_i, q_j)) \in \delta$.
- $\mathcal{AP}' = \mathcal{AP} \cup I$.
- $\mathcal{L}' : Q' \to 2^{\mathcal{AP}}$ is a labelling function, such that $\mathcal{L}'(q_i') = \mathcal{L}(q_i) \cup A_i$, where $q_i' = (q_i, A_i)$

For example, the Kripke Structure equivalent of the controller of Figure 1.1, shown in Figure 3.2, can be expressed as the tuple $\langle Q', \delta', Q'_0, \mathcal{AP}, \mathcal{L}' \rangle$ where,

- $Q' = \{S_0, S_1\} \times 2^{\{water\_level<15, water\_level>70\}}$
- $Q'_0 = \{S_0\} \times 2^{\{water\_level<15, water\_level>70\}}$
- δ' is as illustrated in Figure 3.2
- $\mathcal{AP}' = \{PumpON, water\_level < 15, water\_level > 70\}$
- $\mathcal{L}'(q'_i) = \begin{cases} A_i, & s = S_0 \\ \{PumpON\} \cup A_i, & s = S_1 \end{cases}$ where $q'_i = (q_i, A_i)$

A *path* $\pi = q'_0, q'_1, \ldots$ in the Kripke Structure is defined as an infinite sequence of states, where $\forall i, q'_i \in Q'$, $q'_0 \in Q'_0$, and $\forall i, (q'_i, q'_{i+1}) \in \delta'$.

3.2 Signal Trace

Let $\Sigma = I_B \cup var \cup \mathcal{AP}$ denote the set of variables of the controller. $I_B \cup var$ contains the input variables and $\mathcal{AP}$ represents the set of outputs asserted by the controller.A *signal trace* (or simply *trace*) is defined as follows:

Definition 4 (Signal Trace)**.** *A signal trace σ, is defined as an infinite sequence $A_0, A_1, \ldots$, where each $A_i \in 2^{I_B} \times \mathbb{R}^{|var|} \times 2^{\mathcal{AP}}$.* □

In other words, a trace is an infinite sequence of valuations of the variables in Σ. It may be noted that the valuations of the variables in *var* are real valued, while the rest are Boolean. A trace $\sigma = A_0, A_1, \ldots$, *simulates* a path $\pi = q'_0, q'_1, \ldots$ of the Kripke Structure iff for every k:

1. Set of propositions in $\mathcal{L}'(q'_k)$ is exactly equal to the subset of $\mathcal{AP}$ that is true in A_k, and
2. the valuation of the variables, *var*, in A_k satisfies each PORV that labels q'_k.

3.3 Logic for properties over PORVs

For expressing properties in terms of PORVs, an extended form of LTL is proposed here, which defines formulae over atomic propositions as well as PORVs. Given $\Sigma = I_B \cup var \cup \mathcal{AP}$, the set of variables, well-formed formulae in extended LTL may be:

- $p \in I_B \cup \mathcal{AP}, \top, \bot$
- q, a PORV over var
- $\varphi \wedge \psi$, where φ and ψ are well formed formulae
- $\neg\varphi$, where φ is a well formed formula
- $X\varphi$, where φ is a well formed formula
- $\varphi U \psi$, where φ and ψ are well formed formulae

Truth of formulae written in extended LTL are interpreted over signal traces as defined earlier. The signal trace $\sigma = A_0, A_1, \ldots$ is said to satisfy the formula φ, written as $\sigma \models \varphi$ under the following conditions:

- $\sigma \models \top$
- $\sigma \not\models \bot$
- $\sigma \models p$, where $p \in I$ iff p is made true by A_0
- $\sigma \models \phi \wedge \psi$ iff $\sigma \models \phi$ and $\sigma \models \psi$.
- $\sigma \models \neg\phi$ iff $\sigma \not\models \phi$.
- $\sigma \models X\,\phi$ iff $\sigma \models \phi$.
- $\sigma \models \phi\ U\ \psi$ iff for some i $\in \mathbb{N}_0, \sigma_i \models \psi$ and for all $j<i$, $\sigma_j \models \phi$.

Here, σ_i is defined as the suffix $A_i, A_{i+1}, \ldots$ of σ.

In addition to the operators discussed above, the Boolean connectives $\vee$, $\Rightarrow$, $\Leftrightarrow$ and additional temporal operators F, G and V are also used. The temporal operators are defined as follows:

- $F\,\phi \equiv \top\ U\,\phi$
- $G\,\phi \equiv \neg F \neg \phi$
- $\phi\ V\,\psi \equiv \neg(\neg\phi U \neg\psi)$

The following are examples of well formed formulae in extended LTL:

- $GF((x \geq 10) \vee ((y < 5)U(x + y > 15)))$
- $(\mathit{speed} \leq 15) \Leftrightarrow FG(\mathit{throttle} \leq 10)$

Since formulae in extended LTL are similar to LTL formulae, their truth can also be interpreted over infinite sequences of truth assignments to the PORVs and propositions that appear in them. The interpretation is exactly same as that of traditional LTL formulae, and given a trace, PORVs are treated just like atomic propositions. If a trace $\sigma \models \varphi$, and S be the sequence of truth assignments that σ gives to the PORVs and propositions in φ, then S$\models \varphi$. Also, if S is a sequence of truth assignments (to the PORVs and propositions) that satisfies φ, and σ is any trace such that it defines the same truth assignments to those PORVs and propositions as S, then $\sigma \models \varphi$.

A Kripke Structure $\mathcal{M}$ is said to satisfy a property φ, written as $\mathcal{M} \models \varphi$, iff there exists no trace σ such that σ simulates a path in $\mathcal{M}$ and $\sigma \not\models \varphi$. For example, the Kripke Structure shown in Figure 3.2 satisfies the property $F(\mathit{water\_level} \geq 65) \vee G(\mathit{water\_level} < 100)$ (A close look will reveal that this property, in fact, is a tautology), but not $G(\mathit{water\_level} \geq 30)$.

Chapter 4

Automata Theoretic Approach

It has been shown previously [12, 15] that LWAA characterize precisely the class of star-free ω-regular languages, which correspond to first order definable ω-languages and therefore also the languages definable using propositional LTL formulae. Given the nature of extension that has been made to LTL, this equivalence shall hold for extended LTL also. This equivalence is key to the correctness of automata-theoretic verification because, given a formula to be verified, automata theoretic verification begins by constructing the LWAA corresponding to the negation of the property.

4.1 Translation from formulae to LWAA

The construction of the Linear Weak Alternating Automaton $\mathcal{A}_\phi$ equivalent to a given formula ϕ shall now be described. As already discussed,it can be assumed that the formula is in negation normal form, without any loss of generality. The automaton is defined as $\mathcal{A}_\phi = (Q, q_\phi, \delta, F)$, where Q is the set of states, containing a location q_ψ corresponding to every subformula ψ of ϕ, and q_ϕ is the initial state. The input alphabet is not explicitly defined, but the existence of $\mathcal{P}$, the set of PORVs and propositions in ϕ, is assumed.
Transition function δ can be defined in various ways. One straightforward way

is to define it as a function from $Q \times 2^{\mathcal{P}}$ to $\mathcal{B}^+(Q)$. However, a somewhat more compact representation commonly used in literature is to define δ as a function from Q to the set of propositional formulae over $\mathcal{P} \cup Q$, in which elements of Q may not occur in negated form. For example, if $\mathcal{P} = \{p_1, p_2\}$, one could write

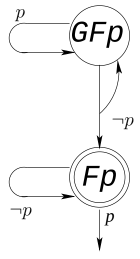
$$\delta(q) = (p_1 \wedge \neg p_2 \wedge q_1) \vee (\neg p_1 \wedge p_2 \wedge (q_1 \vee (q_2 \wedge q_3)) \vee (p_1 \wedge p_2)$$

to represent $\delta(q, \{p_1\}) = q_1$, $\delta(q, \{p_2\}) = (q_1 \vee (q_2 \wedge q_3))$, $\delta(q, \{p_1, p_2\}) = \mathbf{true}$. One of the major differences between model checking with propositions and model checking with PORVs is that all truth assignments of PORVs may not be consistent. This knowledge is used to keep down the number of outgoing transitions from each state in the LWAA. Once the Boolean formula representing the transitions is obtained from a state q in the form of a disjunctive formula, each of the terms that occur in the formula are checked for satisfiability. Any unsatisfiable term is summarily dropped. For example, if $\delta(q) = ((y < 5) \wedge q_1) \vee ((x < 7) \wedge (x > 23) \wedge q_2))$, it is reduced to $\delta(q) = (y < 5) \wedge q_1$.
Figure 4.1 gives the rules for defining the transition function, and also an example automaton, for the property GFp. That an automata with these transition rules is an LWAA can be proven easily. According to the rules, there can exist a transition from state q_ψ to state q_χ only if χ is a subformula of ψ. Since the relation of a formula being the subformula of another obviously defines a partial order, the requirement of reachability relation defining a partial order on the set of states is satisfied by this automaton.

Final states of the LWAA are the states corresponding to subformulae of the form $\phi U \psi$. This choice is fairly intuitive. The requirement is that the automata doesn't get trapped in $q_{\phi U \psi}$. Rather, ψ has to be satisfied at some future time instant, when the automata can come out of $q_{\phi U \psi}$. So, the accepting runs of the automata are the ones which do not have $q_{\phi U \psi}$ occurring

location(q)	$\delta(q)$
q_ψ(ψ is a literal)	ψ
$q_{X\psi}$	q_ψ
$q_{\psi\wedge\chi}$	$\delta(q_\psi) \wedge \delta(q_\chi)$
$q_{\psi\vee\chi}$	$\delta(q_\psi) \vee \delta(q_\chi)$
$q_{\psi U\chi}$	$\delta(q_\chi) \vee (\delta(q_\psi) \wedge q_{\psi U\chi}$
$q_{\psi V\chi}$	$\delta(q_\chi) \wedge (\delta(q_\psi) \vee q_{\psi V\chi}$

(a) Transition Table



(b) $\mathcal{A}_{GFp}$

Figure 4.1: Translation from LTL formulae to LWAA[7]

infinitely often. One can see that, given a property φ and a trace σ that satisfies it, the sequence ν of truth assignments that σ gives to the elements of $\mathcal{P}$ shall constitute an accepting run of $\mathcal{A}_\varphi$. Likewise, given a sequence ν that constitutes an accepting run of $\mathcal{A}_\varphi$, any trace that defines the sequence of truth assignments ν to the elements of $\mathcal{P}$ will satisfy φ.

4.2 Model checking using LWAA

The model checking task is to check whether there exists a path of the Kripke Structure $\mathcal{M}$ corresponding to the controller, that is simulated by a trace that produces an accepting run of $\mathcal{A}_{\neg\phi}$ (If such a path exists, it represents a counterexample to the property being verified). For this, the product of $\mathcal{A}_{\neg\phi}$ with $\mathcal{M}$ is computed, and checked for emptiness. A popular optimization on this, that is used during traditional automata theoretic verification, is to perform the emptiness check *on-the-fly*-that is, as the product is still being computed[7]. This approach is followed here.

As has been seen already, the LWAA that is constructed from a given formula goes from one configuration to possibly different configurations for the different truth assignments of the propositions and PORVs over which the formula is defined. It can thought of as running over $\nu = \nu_0\nu_1....$, where each ν_i correspond to different truth values to each of the propositions and PORVs. Runs of an LWAA(or any alternating automaton) over such sequences give rise to trees, due to universal branching. A more economical way of representing these runs is in the form of a directed acyclic graph, where states active along multiple paths is represented only once. Figure 4.2 illustrates a possible dag for $\mathcal{A}_{GFp}$, shown in Figure 4.1(b). It can be seen that each vertical “slice” of the dag represents a set of states that are active simultaneously. Each such set is referred to as a configuration. For formally defining a run dag and configurations of an automaton, a set of PORVs and propositions and the Boolean valuation that makes true precisely the elements of the set is identified. Depending on whether a formula evaluates to true or not under that valuation, one says that the set does or does not satisfy a formula. For instance, $\{q_1, q_2\}$ can be said to satisfy the formula $(q_1 \wedge q_2) \vee q_3$. For a relation $r \subseteq S \times T$, let its domain be denoted by $\text{dom}(r)$, and the image of a set $A \subseteq S$ under r by $r(A)$.

Definition 5 (Run dag). *[7]Let $\mathcal{A} = (Q, q_0, \delta, F)$ be a Co-Büchi alternating automaton and $\nu = \nu_0\nu_1....$ where $\nu_i \subseteq \mathcal{P}$ be a temporal structure. A* run dag *of $\mathcal{A}$ over ν is represented by the infinite sequence $\Delta = e_0e_1...$ of its edges, where $e_i \subseteq Q \times Q$. The configurations $c_0c_1...$ of Δ, where $c_i \subseteq Q$ are inductively defined by $c_0 = \{q_0\}$ and $c_{i+1} = e_i(c_i)$. For all $i \in \mathbb{N}$, it is required that* $\text{dom}(e_i) \subseteq c_i$ *and that for all $q \in c_i$, the valuation $\nu_i \cup e_i(\{q\})$ satisfies $\delta(q)$. A* finite run dag *is a finite prefix of a run dag.*
A path *in a run dag Δ is a (finite or infinite) sequence $\pi = p_0p_1...$ of locations $p_i \in Q$ such that $p_0 = q_0$ and $(p_i, p_{i+1} \in e_i$ fro all i. A run dag Δ is accepting iff for all infinite paths π in Δ, no final state occurs infinitely often in π. The language accepted by $\mathcal{A}$, represented by $\mathcal{L}(\mathcal{A})$ is the set of words that admit some accepting run dag.* □

There exist necessary and sufficient conditions that can be checked to decide the emptiness of any LWAA, but checking them happens to be somewhat

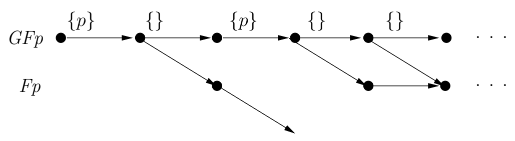


Figure 4.2: Use of dag to represent runs

cumbersome. So, a class of LWAA known as simple LWAA is identified, that gives a much simpler emptiness criterion.

Definition 6 (Simple LWAA). *[7] An LWAA $\mathcal{A} = (Q, q_0, \delta, F)$ is simple if for all $q \in F$, all $q' \in Q$, all $s \subseteq \mathcal{P}$, and all $X, Y \subseteq Q$ not containing q, if $s \cup X \cup \{q\} \models \delta(q')$ and $s \cup Y \models \delta(q)$ then $s \cup X \cup Y \models \delta(q')$.* $\square$

This means that, in a simple LWAA, if a final state q can be activated from some state q' for some truth combination s while it can be exited during the same transition, then it is possible to activate from q' all those states except q that can be activated from q and $q\prime$ together, and avoid activating q at all. The emptiness condition for simple LWAA is as follows:

Theorem 1. *[7] Assume that $\mathcal{A} = (Q, q_0, \delta, F)$ is a simple LWAA. Then $\mathcal{L}(\mathcal{A}) \neq \emptyset$ if and only if there exists a finite run dag $\Delta = e_0e_1...e_n$ with configurations $c_0c_1...c_{n+1}$ over a finite sequence $\nu_0\nu_1....$ of truth assignments and some $k \leq n$ such that*

1. *$c_k = c_{n+1}$ and*

2. *for every $q \in F$, one has $q \notin c_j$ for some j where $k \leq j \leq n$.*

Proof of this theorem has been reproduced from [7] in Appendix A.
It has been shown that, for any formula ϕ that does not contain any sub formula of the type $X(\varphi_1 U \varphi_2)$, the automata A_ϕ is a simple LWAA[7]. But subformulae of such form can be easily removed from any given formula, because X distributes over U. So, one can say that for any given formula ϕ, it is possible to construct a simple LWAA that accepts $\mathcal{L}(\phi)$, the set of infinite strings that satisfy ϕ.

4.3 On-the-fly Verification Algorithm

The model checking algorithm using LWAAs directly shall now be presented. The task of the algorithm is to check for common accepting runs of a Kripke Structure $\mathcal{M}$ and an LWAA $\mathcal{A}$. The pseudo code for the algorithm is given in Figure 4.3. The algorithm is almost exactly same as the one presented in [7], for on-the-fly model checking for LTL properties with propositions. It is based on Tarjan's algorithm for detecting strongly connected components in a graph [14]. The algorithm explores the product automaton of $\mathcal{M}$ with $\mathcal{A}$, starting from $(s_0, \{q_0\})$, where s_0 is the initial state of $\mathcal{M}$, and $\{q_0\}$ is the initial configuration of $\mathcal{A}$. The algorithm proceeds essentially like a Depth First Search on pairs (s, C), where s is a state of $\mathcal{M}$, and C is a configuration of $\mathcal{A}$. The pair (s, C) means that the current configuration of the LWAA is C, and it is about to consume input s, which is the state of the Kripke Structure. From any pair $c = (s, C)$, the set $succ_{\mathcal{M}}(s) \times succ_{\mathcal{A}}(s, C)$ is computed, that contains all (s', C') pairs to which the product automaton may transit, from (s, C). Here, $succ_{\mathcal{M}}(s)$ represents the set of successor states of s in $\mathcal{M}$, and $succ_{\mathcal{A}}(s, C)$ represents the set of possible successor configurations of $\mathcal{A}$ from configuration C, for all input truth combinations of PORVs and propositions that are consistent with the labels of s.An SMT solver is used to perform this consistency check.

Tarjan's algorithm assigns to each node of the graph a *root*, which represents the oldest node on the DFS stack that is known to belong to the same SCC. The requirement in model checking is to verify that the product automaton satisfies Theorem 1. For this purpose, with the root candidate of each SCC, a list is maintained, of final states that have been found absent in the configuration C of some pair (s, C) belonging to that SCC. When it is found that a particular root node has all the final states in its list, the product can be declared to be non empty. This is because, in the SCC for which that node is the root, one

```
procedure Visit(s,C):
    let c=(s,C) in
        inComp[c]:=false; root[c]=c; labels[c]:=$\emptyset$;
        cnt[c]:=cnt; cnt:=cnt+1; seen:=seen $\cup$ {c};
        push(c,stack);
        forall c$'$ =(s$'$,C$'$) in Succ(c) do
            if c$'$ $\notin$ seen then Visit(s$'$,C$'$) end if;
            if $\neg$ inComp[c$'$] then
                if cnt[root[c$'$]] < cnt[root[c]] then
                    labels[root[c$'$]]:=labels[root[c$'$]] $\cup$
                            labels[root[c]];
                    root[c]:=root[c$'$];
                end if;
                labels[root[c]]:=labels[root[c]] $\cup$ (f_lwaa$\setminus$ C);
                if labels[root[c]]=f_lwaa then raise Good_Cycle
                end if;
            end if;
        end forall;
        if root[c]=c then
            repeat
                d:=pop(stack);
                inComp[d]:=true;
            until d=c;
        end if;
    end let;
end Visit;
procedure Check:
    stack:=empty;seen:=$\emptyset$;cnt:=0;
    Visit(init_ts,{init_lwaa});
end Check;
```

Figure 4.3: LWAA based model checking algorithm

can have a sequence of pairs, with the first pair same as the last one, such that for each $q \in F$, there is a pair in the sequence whose configuration does not contain q. If one looks at the configurations in each of the pairs in this sequence, it can be seen that they constitute the sequence $c_k ... c_{n+1}$ mentioned in Theorem 1. The sequence $c_0 c_1 ... c_k$ certainly exists, since the search arrived at a pair with configuration c_k, starting from pair (s_0, c_0). Though the sequence of states in the Kripke Structure that lead to a violation of the property has been obtained, due to the presence of predicates, this sequence corresponds to many different traces, and it is possible that there be traces that proceed through these states and yet not refute the property. To find exactly which traces do refute the property, for each (s, C), it is necessary to keep track of the intersection regions between the labels of the system at state s and the input combination by which the transition from C to the next configuration C'(assuming that (s', C') is the next the pair in common run) is marked. Once a run is found to be accepting, any trace where the variable valuations lie within the intersection regions for that run is a valid counterexample.
Figure 4.3 shows the pseudocode for the algorithm. It is in fact the same as in [7]. The difference between the 2 algorithms is in the Succ() procedure.

4.4 Proof of Correctness

By the correctness of Tarjan's algorithm, if there is an MSCC in the product, it will be found. And that the product is non-empty if there exists an MSCC has already been proven. So, the correctness of the technique is captured by the following theorem:

Theorem 2. *The product computed by the proposed technique is non empty iff the property is violated.*

Proof

Part 1

Suppose that there exists a counterexample trace σ that is simulated by $\mathcal{M}$. Let σ_i represent the real value and Boolean assignments that σ makes to the real variables and propositions in the ith step. Since σ is a counterexample, there must exist a sequence of configurations $C_0C_1\ldots$ of the LWAA that constitutes an accepting path such that the transitions form C_i to C_{i+1} is enabled by σ_i. Let s_i be the state of the Kripke Structure corresponding to σ_i. Existence of σ_i means that s_i and the labels from C_i to C_{i+1} has an intersection, and hence, $Succ(s_i, C_i)$ shall include C_{i+1}. Since between any to configurations C_i and C_{i+1}, the transitions are enabled by s_i, this sequence will exist in the computed product also.

Part 2

Suppose that the search gives a counterexample

$$(s_0, C_0), (s_1, C_1)\ldots(s_k, C_k), (s_{k+1}, C_{k+1})\ldots(s_n, C_n), (s_{n+1}, C_{n+1})$$

where $C_k = C_{n+1}$. The sequence of configurations $C_0C_1\ldots$ is indeed a valid accepting sequence for the negation of the property. Also, the sequence of states $s_0, s_1, \ldots$ is a valid path in the Kripke Structure. What remains to be shown is that there exists a trace, that takes $\mathcal{M}$ through $s_0, s_1, \ldots$, and takes the LWAA through $C_0C_1\ldots$.

Since there is an intersection between s_i and the labels of the transitions from C_i to C_{i+1}, there is at least one possible assignment of real and Boolean values to the variables and propositions for which the transition is enabled, and the labels of s_i are made true. Let the assignment be denoted by η_i. Let the trace σ be defined as $\eta_0\eta_1\ldots$. It is easy to see that σ is simulated by $\mathcal{M}$, and also defines an accepting path in the LWAA. $\square$

4.5 Chapter Summary

The equivalence between extended LTL formulae and Linear Weak Alternating Automata serves as the foundation for Automata Theoretic Verification. Given a formula in extended LTL, The LWAA corresponding to its negation is constructed. Transitions take the LWAA from one configuration to the next, and these transitions are labelled with truth assignments to the propositions and PORVs. During model checking, the product of the LWAA and the Kripke Structure representing the controller is taken, and checked for emptiness. A transition that exists from the current configuration in the LWAA can be taken in the product iff the labels of that transition are consistent with the labels of the current state of the Kripke Structure. Tarjan's MSCC finding algorithm is used for checking the product for emptiness even as it is being computed.

Chapter 5

Tableau based verification

Years of research in the field of tableau based verification (*symbolic model checking*) has given rise to amazingly efficient model checkers, that can traverse huge state spaces within reasonable amounts of time. So, rather than going for an altogether new model checking algorithm, a reduction from verification with PORVs to the traditional verification problem, that can be solved by symbolic model checking, is presented here.

5.1 The Verification Methodology

Given a PORV-labelled Kripke Structure $\mathcal{M}$ and a property φ, the model checking task is a search for a trace σ such that σ simulates a path in $\mathcal{M}$ and $\sigma \not\models \varphi$. Since the PORVs are defined over dense real valued variables, each path of $\mathcal{M}$ can have infinite number of traces that simulate it. Also, due to non-determinism, a trace may simulate multiple paths in $\mathcal{M}$.

Let Traces(φ) denote the set of traces satisfying φ, and let Traces($\mathcal{M}$) denote the set of traces that simulate some path in $\mathcal{M}$. The objective is to determine whether Traces($\mathcal{M}$) $\bigcap$ Traces($\neg\varphi$) $= \emptyset$.

Let $True(\varphi, \sigma, i)$ denote the set of propositions and PORVs of φ that are made true by A_i on the trace $\sigma = A_0, A_1, \ldots$. Let $True(\pi, i)$ denote the set of propositions and PORVs labelling state q'_i in the path $\pi = q'_0, q'_1, \ldots$ of the Kripke Structure. $True(\varphi, \sigma, i)$ and $True(\pi, i)$ are said to be *consistent* iff:

1. the set of atomic propositions of φ in $True(\varphi, \sigma, i)$ and the set of atomic propositions of φ in $True(\pi, i)$ are exactly the same, and
2. the conjunction of PORVs in $True(\varphi, \sigma, i)$ is satisfiable with the conjunction of PORVs in $True(\pi, i)$, that is, there is some valuation of *var* that satisfies all PORVs in $True(\varphi, \sigma, i)$ and all PORVs in $True(\pi, i)$.

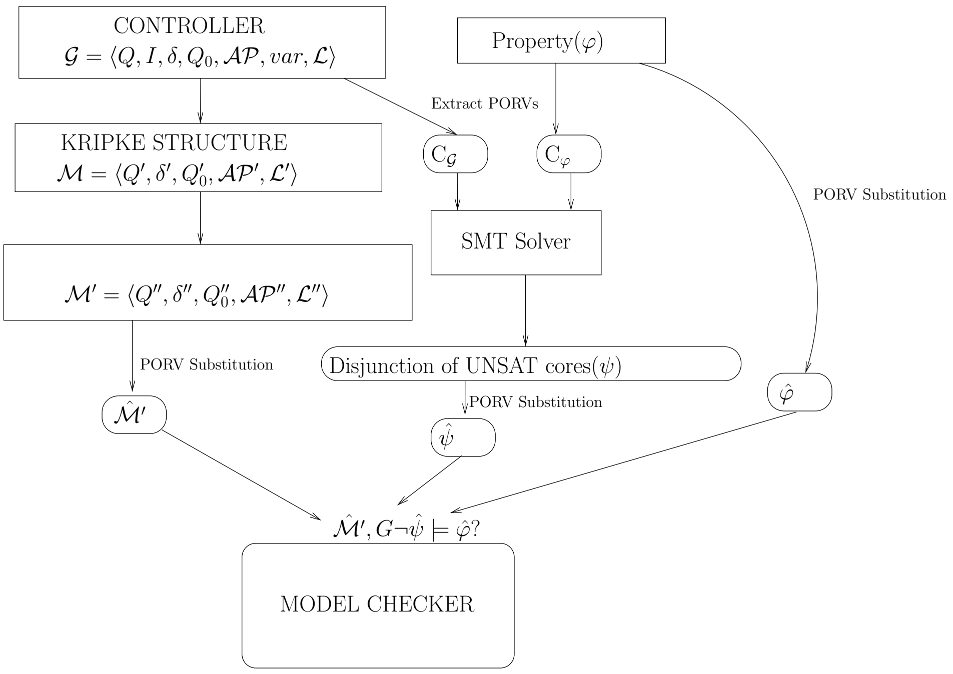
A sequence, $\nu = \nu_0, \nu_1, \ldots..$, is said to be *consistent* with a path, $\pi = q'_0, q'_1, \ldots$, of a Kripke Structure, $\mathcal{M}$, iff ν_i is consistent with $True(\pi, i)$, for all $i \geq 0$.

Lemma 1. *For a given PORV-labelled Kripke Structure $\mathcal{M}$ and a property φ, $\mathcal{M} \not\models \varphi$ if and only if there exists a sequence, $\nu = \nu_0, \nu_1, \ldots..$ over $\mathcal{P}$ such that $\nu \models \neg\varphi$ and ν is consistent with some path π in $\mathcal{M}$.*

Proof:

Part 1

Assume that there indeed exists a sequence ν, such that $\nu \models \neg\varphi$ and ν is consistent with some path π in $\mathcal{M}$. Since $\nu \models \neg\varphi$, if σ be any trace such that $True(\neg\varphi, \sigma, i) = \nu_i$ for all $i \geq 0$, then $\sigma \models \neg\varphi$. Since ν_i is consistent with $True(\pi, i)$ for all $i \geq 0$, for each i, there exists $\eta(i)$, a Boolean and real value assignment to the propositions and real values in Σ respectively, such that it makes true exactly those propositions that are present in $True(\pi, i)$ (since the set of propositions of φ present in in $True(\pi, i)$ and ν_i are the same, among propositions of φ, it naturally makes true only those that are present in ν_i), and those PORVs that are present in ν_i or in $True(\pi, i)$. Trace $A = \eta(1), \eta(2), \ldots$ is constructed. Since $\eta(i)$ makes true exactly the propositions in $True(\pi, i)$, and also all the PORVs in $True(\pi, i)$, A simulates π. Also, since $True(\neg\varphi, A, i) = \nu_i$, $A \models \neg\varphi$. So, $M \not\models \varphi$.



CONTROLLER
$\mathcal{G} = \langle Q, I, \delta, Q_0, \mathcal{AP}, var, \mathcal{L}\rangle$
Property(φ)
Extract PORVs
KRIPKE STRUCTURE
$\mathcal{M} = \langle Q', \delta', Q_0', \mathcal{AP}', \mathcal{L}'\rangle$
$C_{\mathcal{G}}$
C_{φ}
PORV Substitution
SMT Solver
$\mathcal{M}' = \langle Q'', \delta'', Q_0'', \mathcal{AP}'', \mathcal{L}''\rangle$
PORV Substitution
Disjunction of UNSAT cores(ψ)
PORV Substitution
$\hat{\varphi}$
$\hat{\mathcal{M}'}$
$\hat{\psi}$
$\hat{\mathcal{M}'}, G\neg\hat{\psi} \models \hat{\varphi}$?
MODEL CHECKER

Figure 5.1: Verification Toolflow

Part 2

Assume that $\mathcal{M} \not\models \varphi$. So, there exists a path π in $\mathcal{M}$, which is simulated by some trace σ, and $\sigma \models \neg\varphi$. From the observations made earlier, it can be seen that $\nu \models \neg\varphi$, where ν is the sequence $\nu_1, \nu_2, \ldots$ such that $\nu_i = True(\neg\varphi, \sigma, i)$. Since σ simulates π, it must make true only those propositions, and at least those PORVs, that occur in $True(\pi, i)$, for all $i \geq 0$.The existence of σ means that for all $i \geq 0$, $True(\pi, i)$ is consistent with $True(\neg\varphi, \sigma, i)$, since both requirements for consistency are satisfied by σ_i. So, ν is consistent with π. $\square$

Lemma 1 forms the basis of the proposed verification method. Each path π in a PORV-labeled Kripke Structure, $\mathcal{M} = \langle Q', \delta', Q_0', \mathcal{AP}', \mathcal{L}'\rangle$, represents a sequence over $\mathcal{AP}'$, but since $\mathcal{AP}' \neq \mathcal{P}$, truth of φ cannot be interpreted over this sequence directly. However, one can consider all possible sequences over $\mathcal{P}$ that are consistent with π and determine (using standard interpretation of LTL) whether any of them refutes φ. If so, then by Lemma 1, $\mathcal{M} \not\models \varphi$, and the path π is simulated by some trace that refutes φ.

Therefore, for a given PORV-labeled Kripke Structure, $\mathcal{M} = \langle Q', \delta', Q'_0, \mathcal{AP}', \mathcal{L}' \rangle$, and a property φ, Kripke Structure $\mathcal{M}' = \langle Q'', \delta'', Q''_0, \mathcal{AP}'', \mathcal{L}'' \rangle$ is developed, as follows:

- $Q'' = Q' \times 2^{\mathcal{C}_\varphi \setminus \mathcal{C}_\mathcal{M}}$
- $\delta'' = \{((s,a),(s',b)) | (s,s') \in \delta',\ a,b \in 2^{\mathcal{C}_\varphi \setminus \mathcal{C}_\mathcal{M}}\}$
- $Q''_0 = Q'_0 \times 2^{\mathcal{C}_\varphi \setminus \mathcal{C}_\mathcal{M}}$
- $\mathcal{L}'' : Q'' \rightarrow 2^{\mathcal{C}_\varphi \cup \mathcal{C}_\mathcal{M}}$, is defined as $\mathcal{L}''((s,a)) = \mathcal{L}'(s) \cup a$.

where $\mathcal{C}_\mathcal{M}$ denotes the PORVs in $\mathcal{M}$. The states of the Kripke Structure $\mathcal{M}'$ are labeled with the PORVs of $\mathcal{M}$ as well as φ, the truth of φ can be interpreted over the paths of $\mathcal{M}'$ using the standard semantics of LTL. This enables the use of a standard LTL model checker to check φ on $\mathcal{M}'$. However this check is not equivalent to checking whether $\mathcal{M} \models \varphi$ for the reason explained below. A state, s, of $\mathcal{M}'$ may have inconsistent labels, that is, the PORVs labeling s (consisting of the PORVs from $\mathcal{M}$ and φ) may not be satisfiable together. The model checker sees these PORVs as just propositions, and is hence incapable of detecting the occurrence of these states. If the model checker finds a counter-example path π, that contains such a state, then the counter-example is fictitious. This is because the sequence over $\mathcal{P}$ defined by π is not consistent with the underlying path of $\mathcal{M}$ that π represents.

In order to prevent the model checker from coming up with such counter-examples, there are two broad options, namely:

1. Eliminate the inconsistently labeled states from $\mathcal{M}'$, or
2. Add fairness constraints to prevent counter-examples containing inconsistently labeled states

The second option is considerably easy to implement and hence it has been chosen here, and the following steps have been developed to prevent spurious counter-examples.

1. Take the union, $\mathcal{C} = \mathcal{C}_\varphi \bigcup \mathcal{C}_\mathcal{M}$ and use a SMT solver to compute the set of minimal unsatisfiable cores of $\mathcal{C}$.
2. For each unsatisfiable core add a constraint to express that no state in the path should contain the elements of this core in its labels. This can be expressed in LTL as the property:

   $$G(\neg\hat{\psi})$$

   where $\hat{\psi}$ is the disjunction of the UNSAT cores, with the PORVs replaced by propositions.

The steps of the proposed model checking methodology is shown in Figure 5.1. The steps of this method is demonstrated using the controller $\mathcal{G}$ shown in Figure 1.1. Consider verification of the property $G((water\_level < 10) \Rightarrow XPumpON)$ over $\mathcal{G}$. First, the corresponding Kripke Structure $\mathcal{M}$ is constructed, that is shown in Figure 3.2. Now, $\mathcal{M}'$ is constructed, which has of two copies of each state in $\mathcal{M}$, one which is labelled with $(water\_level < 10)$ and one which is not. For every edge from state q to state q' in $\mathcal{M}$, in $\mathcal{M}'$, there are edges from both copies of q to both copies of q'. Using an SMT solver, compute the set of UNSAT cores of the set $\{(water\_level < 10), (water\_level < 15), (water\_level > 70)\}$ is computed. Let ψ denote the disjunction of these cores.

$$\psi = ((water\_level < 10) \wedge \neg(water\_level < 15)) \vee$$

$$((water\_level < 15) \wedge \neg(water\_level < 70)) \vee$$

$$((water\_level < 10) \wedge \neg(water\_level < 70))$$

Once ψ is computed, the following replacement is performed on $\mathcal{M}'$, φ and ψ:

- $(water\_level < 10) \equiv w\_lt\_10$
- $(water\_level > 15) \equiv w\_gt\_15$
- $(water\_level < 70) \equiv w\_lt\_70$

to obtain $\hat{\mathcal{M}}'$, $\hat{\varphi}$ and $\hat{\psi}$ respectively. Now, using an industrial model checker, $\hat{\varphi}$ is verified over $\hat{\mathcal{M}}'$, under the fairness constraint $G\neg\hat{\psi}$.

Lemma 2. *A trace σ simulates a path in $\mathcal{M}'$ if and only if it simulates a path in $\mathcal{M}$(where $\mathcal{M}'$ is defined using $\mathcal{M}$, as shown in Figure 5.1).*

Proof:

Part 1

Consider a trace $\sigma = \sigma_1, \sigma_2, \ldots$ that simulates some path π_1 in $\mathcal{M}$. Let s_i and s_{i+1} be any two successive states in π_1. σ_i and σ_{i+1} make true only those propositions, and at least those PORVs, that are in $\mathcal{L}'(s_i)$ and $\mathcal{L}'(s_{i+1})$ respectively. Let X be the set of PORVs present in φ but not in $\mathcal{M}$, that are made true by σ_i and Y be the same for σ_{i+1}. By the definition of Q'', s'_i= (s_i,X) and s'_{i+1}=(s_{i+1},Y) are present in Q''. Also, by definition of δ'', since (s_i,s_{i+1}) $\in \delta'$, (s'_i, s'_{i+1}) $\in \delta''$, and by definition of Q''_0, $s'_0 \in Q''_0$. So, sequence of states s'_i constitute a path in $\mathcal{M}'$. Since the elements of $\mathcal{L}'(s_i)$ and X are exactly the propositions and PORVs made true by σ_i, σ simulates that path.

Part 2

Let σ be a trace that simulates a path π by $\mathcal{M}'$. Consider any two states (p,a) and (q,b) that occur in π, at positions i and i+1 respectively. According the definition of Q'', there are states p and q in Q', whose labels define the same truth assignment as (p,a) and (q,b) respectively, for the propositions and the PORVs present in $\mathcal{M}$. So, σ_i and σ_{i+1} shall make true the labels of p and q in $\mathcal{M}$. Also, since ((p,a),(q,b)) $\in \delta''$, (p,q) $\in \delta'$. Since $(q, a) \in Q'' \Rightarrow q \in Q'$ and$(q, a) \in Q_0'' \Rightarrow q \in Q_0'$ for all a, there is a path in $\mathcal{M}$, simulated by σ. $\square$

5.2 Proof of Correctness

While verifying $\hat{\varphi}$ over $\hat{\mathcal{M}}$, a model checker reports as a counter-example any path in $\hat{\mathcal{M}}$ that refutes $\hat{\varphi}$ according to LTL semantics. A counter-example so obtained is actually a path in $\mathcal{M}$, that refutes φ. The following theorem, based on this observation, establishes the correctness of the proposed technique:

Theorem 3. *Model checking $\hat{\varphi}$ over $\hat{\mathcal{M}}'$ under the constraint $G\neg\hat{\psi}$ will produce a counter-example if and only if $\mathcal{M}$ refutes φ, and any trace that takes $\mathcal{M}'$ through the path corresponding to the counter-example will be simulated by $\mathcal{M}$, and will violate φ, and hence will be a valid counter-example for φ in $\mathcal{M}$.*

Proof:

Part 1

Assume that the model checker reports as a counter-example a path in $\hat{\mathcal{M}}'$. The labels of this path would define a sequence ν over $\mathcal{A}_\varphi$, such that $\nu \not\models \varphi$. Due to the constraint $G\neg\hat{\psi}$, for any path that the model checker reports, the corresponding path π in $\mathcal{M}'$ cannot contain a state whose labels are inconsistent. For any state s_i in π, let η_i denote an assignment to the Boolean and real variables in *var* such that, among the elements of $\mathcal{AP}''$, only the labels of s_i are made true. Define a trace $\sigma = \eta_1\eta_2\ldots$. Since σ simulates π in $\mathcal{M}'$, it simulates a path in $\mathcal{M}$ too(by Lemma 2). Since ν is the sequence defined by σ over $\mathcal{A}_\varphi$, and σ simulates π, π and ν are consistent. And since $\nu \not\models \varphi$, by Lemma 1, $\mathcal{M} \not\models \varphi$, and σ is the counter-example trace that simulates a path in $\mathcal{M}$ and refutes φ.

Part 2

Assume that $\mathcal{M}$ refutes φ, and σ is the counter-example trace. By Lemma 2,

σ simulates a path in $\mathcal{M}'$. Let it be π. Since π is simulated by a trace, it cannot contain any inconsistent states. So, the constraint $G\neg\hat{\psi}$ is satisfied by the path π' in $\hat{\mathcal{M}}'$ corresponding to π. Also, since $\sigma \not\models \varphi$, the sequence ν that σ defines over $\mathcal{A}_\varphi$ shall also refute φ. Thus, π shall refute φ, and π' shall refute $\hat{\varphi}$, according to LTL semantics. Hence, the model checker will report as counter-example the path π', which corresponds to π, the path simulated by σ in $\mathcal{M}'$. $\square$

5.3 Verification with assume properties

Verification, as seen so far makes no assumptions about the system being verified. While this offers better coverage, it may give unrealistic counterexamples (e.g.: Ambient temperature switches between 0 and 1000 in alternate seconds). There is a very real possibility that a real counterexample goes unnoticed among the thousands of false ones. Making realistic assumptions about the environment, that represent the expected behaviour of the environment in the real world can help in obtaining better counterexamples.

Yet another reason for bringing in assume properties is the need to verify properties over the controller and the plant together, and not just on the controller. In such situations, verification is not possible without using a suitable model of the plant. Assume properties over the plant can serve as plant models, enabling the verification in such cases.

By building on the symbolic approach, a method has been developed for incorporating assume properties in the verification framework. The assume properties are to be written in extended LTL. Like extended LTL assertions, the assume properties can also have PORVs that are different from the ones that appear in the Kripke Structure, but they should be over the same set of real variables. In addition, the PORVs that appear in the assume properties, and the assertions to be verified may aslo be different.

Given a Kripke Structure $\mathcal{M}$, a property φ and an assume property λ, the Kripke Structure $\mathcal{M}'$ is defined as before, but instead of C_φ, $C_\varphi \cup C_\lambda$ is considered, where C_λ is the set of PORVs in λ. That is, the PORVs that appear in λ are treated in a similar manner to the PORVs that appear in φ. Then, the PORV substitution is carried out on λ too, in addition to $\mathcal{M}'$ and φ.
As an example, consider the controller shown in Figure 1.1. Suppose that the property $\varphi = G(water\_level > 5)$ is to be verified, subject to the assume property $\lambda = G(water\_level > 7 \wedge p_1 on \Rightarrow X(water\_level > 10))$. After constructing the Kripke Structure $\mathcal{M}$, $\mathcal{M}'$ is obtained by splitting each state into 8 different states, each labelled with a different truth assignment to the PORVs in $\{(water\_level > 10), (water\_level > 7), (water\_level > 5)\}$. ψ in this case will be $((water\_level > 10) \wedge \neg(water\_level > 7)) \vee ((water\_level > 10) \wedge \neg(water\_level > 5)) \vee ((water\_level > 7) \wedge \neg(water\_level > 5))$. The PORVs in $\mathcal{M}'$, λ, φ and ψ are substituted with propositions to get $\hat{\mathcal{M}}'$, $\hat{\lambda}$, $\hat{\varphi}$ and $\hat{\psi}$ respectively. Now, $\hat{\varphi}$ is checked over $\hat{\mathcal{M}}'$ subject to assume properties $\hat{\lambda}$ and $\hat{\psi}$.

5.4 Chapter Summary

Given a PORV labelled Kripke Structure and a property in extended LTL, one can use industrial model checkers to solve the problem of checking the property over the Kripke Structure, by reducing it into a standard model checking problem with only propositions. The major hurdle in doing so is to ensure that the truth assignments to the various PORVs are consistent. This is done by adding an assume property, that enforces the requirement that no conflicting PORVs may be true together. To find such combinations(UNSAT cores) among the PORVs labelling the Kripke Structure and the PORVs appearing in the property, one can make use of SMT solvers. While model checking with assume properties, in addition to the PORVs from the Kripke Structure and the assertion, UNSAT cores are computed over the PORVs from the assume property as well.

Chapter 6

Case Studies and Experimental Results

As a part of this work, a tool kit has been developed implementing the methodology illustrated in Figure 5.1. It uses MALL[18], a tool that uses lp_solve package in the back end, for computing the unsatisfiable cores and the Synopsys Magellan tool [10] as the model checker. The tool kit has been used for verifying two digital controllers, both of which control hybrid plants. The results of the experiments are presented in this chapter. These results were taken on a 2.0GHz 8-core Intel Xeon with 64GB RAM.

6.1 Steam boiler controller

The first case study presented here is that of a controller that controls the pumps in a steam boiler. This controller is based on the one described in [9]. The task of the controller is to appropriately turn ON/OFF two pumps that feed water into the boiler, and also open/close a valve that releases water from the boiler. the controller has 7 states, as shown in Figure 6.1. Among these, the states BOTH_ON, ONE_ON and BOTH_OFF are the normal operating modes. The transitions from one state to another are labelled with PORVs.

An advantage with this example is that the PORVs used by the controller are visible, and hence, it is easy to formulate the modified Kripke Structure and property (but this may not always be the case). Using the proposed approach, this controller has been verified against properties written using PORVs that are different from the ones labelling the transitions, but are defined over the same real valued variable w. To demonstrate that real counter-examples are not missed, such properties have also been verified, that the controller is not expected to satisfy, and also those that are otherwise satisfied by the controller, after introducing faults in the controller implementation, that would create counter-examples for those properties.
The properties that hold are:

- If a pump is on currently, and water level remains above 27 for 3 consecutive time instants, in the state after that, both pumps will be off.

  $$\varphi_1 = G(p_1on \wedge G_3(w > 27)) \Rightarrow X_3\neg p_1on$$

  ($G_i\psi$ denotes that ψ holds for i consecutive time instants, including the current one, and $X_i\psi$ to specify that ψ holds after i steps).

- If the steam boiler is in operation but not in a normal operating mode, and is waiting, the steam rate is zero and the water level is between 16 and 19 for 3 time instants, normal operation is entered.

  $$\varphi_2 = G(\neg n\_op \wedge \neg stopped \wedge G_3(waiting\wedge$$

  $$\neg steam\_rate\_neq\_0 \wedge w \leq 19 \wedge w \geq 16) \Rightarrow X_3(n\_op))$$

- If the controller is in one of the normal operating modes and the water level falls below 7, within 2 transitions, the controller goes to STOP, or turns both pumps on.

$$\varphi_3 = G((n\_op) \wedge w \leq 7 \Rightarrow (XF_2(p_2on \vee stopped)))$$

($F_i\psi$ denotes that ψ holds within i time steps, including the current one.)

Counter-examples are obtained in the following cases:

- If the boiler is not in STOP, and the water level falls below 5, the boiler stops within 2 steps.

$$\varphi_4 = G((\neg stopped) \wedge (w < 5) \Rightarrow XF_2(stopped))$$

- If both pumps are on, and the water level stays above 25 for 3 time instants, at least one pump is turned off. (Counter-example obtained if pumps are not turned off in STOP mode).

$$\varphi_5 = G((p_1on \wedge p_2on \wedge G_3(w \geq 25)) \Rightarrow X_3\neg p_2on)$$

6.1.1 Assume properties

Properties have been checked over boiler controller example under assume properties too. As already mentioned, the assume properties are formulae in extended LTL, with PORVs over the same real valued variables as the Kripke Structure.
Assume Properties:

$$w \geq 15 \wedge X\neg(w \geq 15) \Rightarrow (X(10 \leq w) \wedge XX(10 \leq w))$$

$$w \geq 20 \wedge X\neg(w \geq 20) \Rightarrow (X(15 \leq w) \wedge XX(15 \leq w))$$

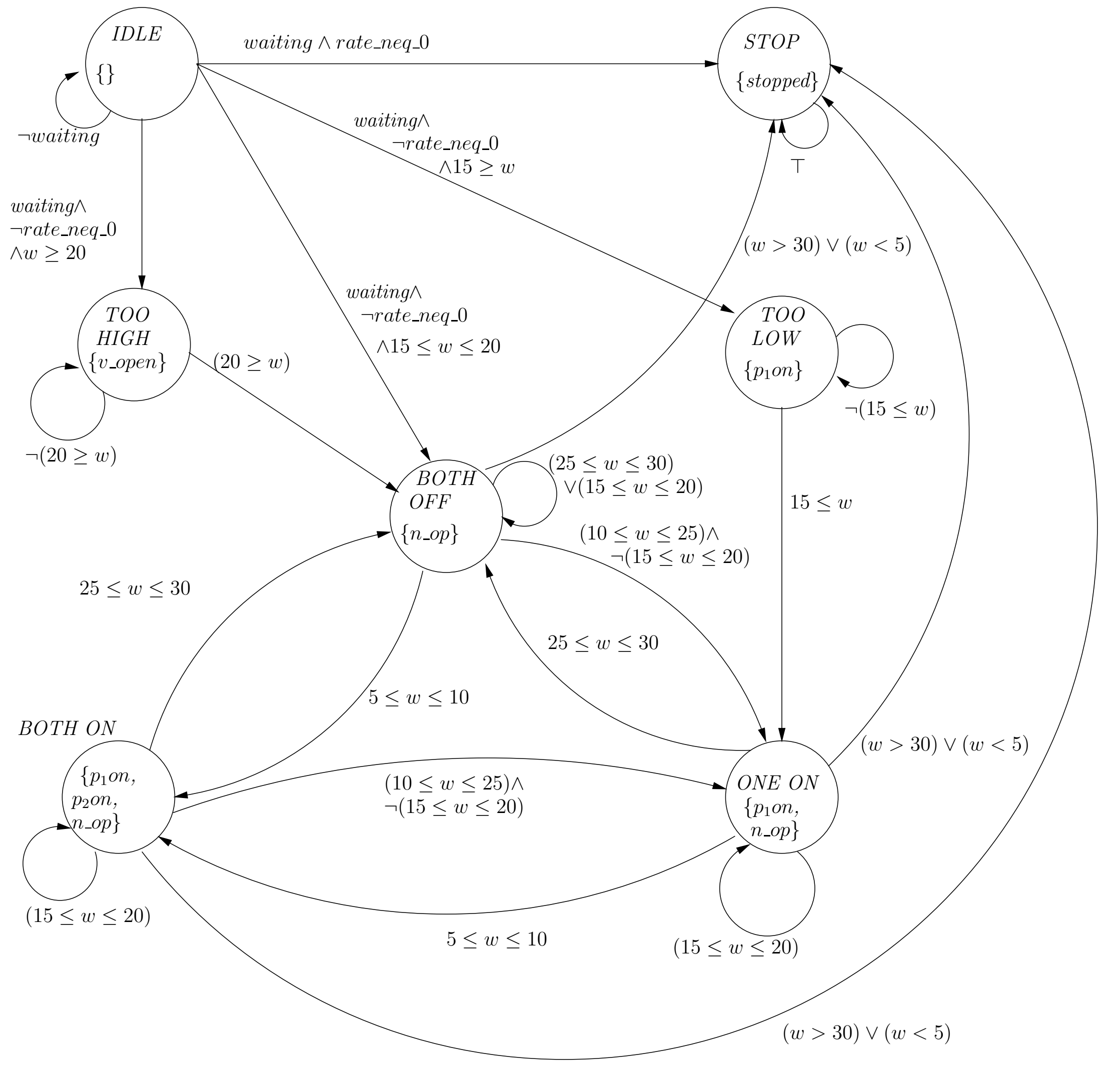


Figure 6.1: Controller for steam boiler pumps

$$w \geq 10 \wedge p_1 on \Rightarrow (X(w \geq 15))$$

Properties found to be satisfied:

$$n\_op \Rightarrow (w \geq 5)$$

Assume Properties:

$$\varphi_6 = \neg(w \geq 25) \wedge X(w \geq 25) \Rightarrow (X(w \leq 30) \wedge XX(w \leq 30) \wedge XXX(w \leq 30))$$

$$\varphi_7 = w \leq 30 \wedge \neg p_1 on \Rightarrow (X(w \leq 23))$$

Properties found to be satisfied:

$$n\_op \Rightarrow (w \leq 30)$$

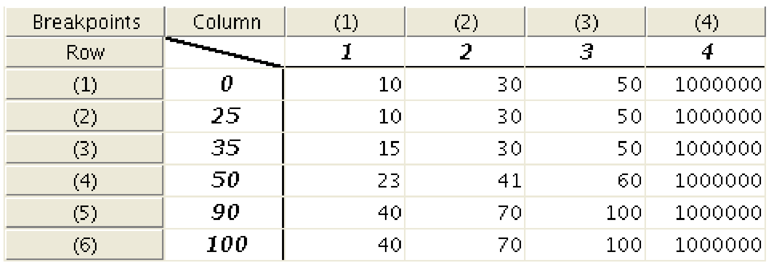
The experimental results are shown in Table 6.1.

Controller	Property	No. of States	No. of UNSAT Cores	Time to compute UNSAT cores	Time for Model Checking	Proven?(Yes/No)
Boiler Controller	φ_1	$7 * 2^{12}$	67	0.04s	28s	Yes
	φ_2	$7 * 2^{13}$	79	0.06s	35s	Yes
	φ_3	$7 * 2^{12}$	67	0.04s	35s	Yes
	φ_4	$7 * 2^{11}$	56	0.03s	25s	No
Boiler Controller (assume properties)	φ_6	$7 * 2^{11}$	56	0.03s	35s	Yes
	φ_7	$7 * 2^{12}$	67	0.04s	36s	Yes
Boiler Controller (modified)	φ_5	$7 * 2^{11}$	56	0.03s	28s	No

Table 6.1: Results for Steam Boiler Controller

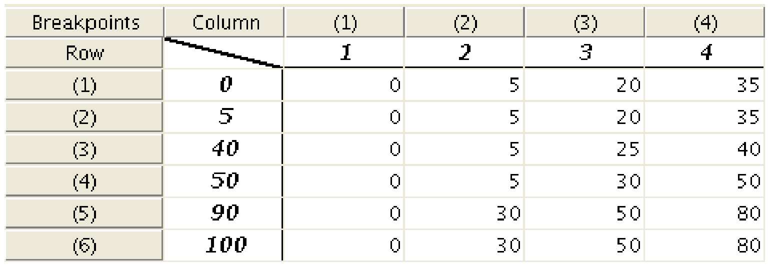
6.2 Automatic Transmission Controller

The proposed approach has also been used to verify a controller designed for controlling the gear positions of a vehicle. This controller is based on an embedded controller implemented in Simulink/Stateflow. As can be seen in Figure 6.3, the controller has four different gear positions to choose from, and the transition from one gear to the next is decided based on the



Breakpoints	Column	(1)	(2)	(3)	(4)
Row		1	2	3	4
(1)	0	10	30	50	1000000
(2)	25	10	30	50	1000000
(3)	35	15	30	50	1000000
(4)	50	23	41	60	1000000
(5)	90	40	70	100	1000000
(6)	100	40	70	100	1000000

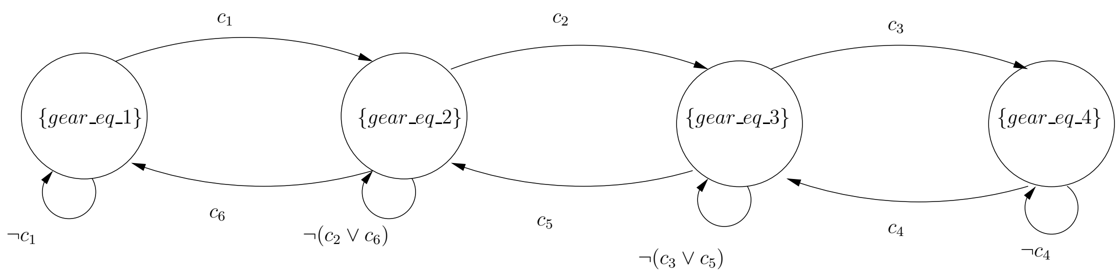
(a) Upper Threshold



Breakpoints	Column	(1)	(2)	(3)	(4)
Row		1	2	3	4
(1)	0	0	5	20	35
(2)	5	0	5	20	35
(3)	40	0	5	25	40
(4)	50	0	5	30	50
(5)	90	0	30	50	80
(6)	100	0	30	50	80

(b) Lower Threshold

Figure 6.2: Lookup tables for computing upper and lower threshold speeds for given gear and throttle (from Stateflow)



$$c_1 = (t \leq 25 \wedge (s - 10 > 0)) \vee (\neg(t \leq 50) \wedge (t \leq 90) \wedge (15s - 8t + 55 > 0)) \vee (\neg(t \leq 25) \wedge (t \leq 35) \wedge (2s - t + 5 > 0)) \\ \vee(\neg(t \leq 35) \wedge (t \leq 50) \wedge (40s - 17t - 90 > 0)) \vee (t > 90 \wedge (s - 40 > 0))$$

$$c_2 = (t \leq 35 \wedge (s - 30 > 0)) \vee (\neg(t \leq 35) \wedge (t \leq 50) \wedge (15s - 11t - 65 > 0)) \\ \vee(\neg(t \leq 50) \wedge (t \leq 90) \wedge (40s - 29t - 190 > 0)) \vee (t > 90 \wedge (s - 70 > 0)) \wedge (s - t < 0))$$

$$c_3 = (t \leq 35 \wedge (s - 50 > 0)) \vee (\neg(t \leq 35) \wedge (t \leq 50) \wedge (3s - 2t - 80 > 0)) \\ \vee(\neg(t \leq 50) \wedge (t \leq 90) \wedge (s - t - 80 > 0)) \vee (t > 90 \wedge (s - 100 > 0))$$

$$c_4 = (t \leq 5 \wedge (s - 35 < 0)) \vee (\neg(t \leq 5) \wedge (t \leq 40) \wedge (7s - t - 240 < 0)) \vee (\neg(t \leq 40) \wedge (t \leq 50) \\ \vee(\neg(t \leq 50) \wedge (t \leq 90) \wedge (4s - 3t - 50 < 0)) \vee (t > 90 \wedge (s - 80 < 0))$$

$$c_5 = (t \leq 5 \wedge (s - 20 < 0)) \vee (\neg(t \leq 5) \wedge (t \leq 40) \wedge (7s - t - 135 < 0)) \\ \vee(\neg(t \leq 40) \wedge (t \leq 90) \wedge (2s - t - 10 < 0)) \vee (t > 90 \wedge (s - 50 < 0))$$

$$c_6 = (t \leq 50 \wedge (s - 5 < 0)) \vee (\neg(t \leq 50) \wedge (t \leq 90) \wedge (8s - 5t + 210 < 0)) \vee (t > 90 \wedge (s - 30 < 0))$$

Figure 6.3: Automatic Transmission Controller

truth of certain PORVs. However, unlike the previous example, in this case, the PORVs are not directly visible in the implementation. The controller computes in software upper and lower thresholds of speed for the current gear position and throttle value, compares them with the current speed, and the gear is shifted up or down as necessary. Figure 6.2 shows the lookup tables used for determining the upper and lower thresholds. The thresholds for intermediate values of throttle are calculated by linear interpolation. Since the functions used for computing the thresholds are linear, the result of the comparisons can be represented as linear predicates over the variables speed and throttle, as shown in Figure 6.3.

The properties that hold include:

- If for consecutive the speed remains above 110 for 3 time instants, and also 3s-2t>80, the gear will be in position 4(s denotes speed, and t throttle).
$$\varphi_8 = G(G_3(s > 110 \wedge 3s - 2t > 80) \Rightarrow X_3(gear\_eq\_4))$$

- Always, if the speed falls below 5, the gear goes down by one position, if possible.

$$\varphi_9 = G(((gear\_eq\_2 \wedge s < 5) \Rightarrow X(gear\_eq\_1))$$
$$\wedge((gear\_eq\_3 \wedge s < 5) \Rightarrow X(gear\_eq\_2))$$
$$\wedge((gear\_eq\_4 \wedge s < 5) \Rightarrow X(gear\_eq\_3)))$$

The following properties are falsified:

- If the speed is in [35,50) and throttle is in (40,50], the gear doesn't change.

$$\varphi_{10} = G((35 < s \leq 50) \wedge (40 < t \leq 50) \Rightarrow$$

$$((gear\_eq\_1 \Rightarrow X(gear\_eq\_1)) \wedge$$

$$(gear\_eq\_2 \Rightarrow X(gear\_eq\_2)) \wedge$$

$$(gear\_eq\_3 \Rightarrow X(gear\_eq\_3)) \wedge$$

$$(gear\_eq\_4 \Rightarrow X(gear\_eq\_4)))$$

- If the speed is greater than or equal 40 and also twice the throttle value for 5 consecutive time instants, then the gear will be at position 4.

$$\varphi_{11} = G(G_5(s \geq 40 \wedge s \geq 2t) \Rightarrow X_5(gear\_eq\_4))$$

The experimental results are shown in Table 6.2.

Controller	Property	No. of States	No. of UNSAT Cores	Time to compute UNSAT cores	Time for Model Checking	Proven?(Yes/No)
Transmission Controller	φ_8	2^{35}	3075	95.70s	175s	Yes
	φ_9	2^{35}	3077	88.48s	134s	Yes
	φ_{10}	2^{34}	2896	81.18s	122s	No
	φ_{11}	2^{36}	3511	127.87s	167s	No

Table 6.2: Results for Automatic Transmission Controller

Chapter 7

Conclusion and Future Work

This project addressed the problem of verifying digital controllers for hybrid plants, by developing model checking techniques for PORV labelled Kripke Structures. Formalisms have been developed for representing LTL-like properties with PORVs, where the PORVs could possibly be different from the ones labelling the states of the Kripke Structures. For verification, a symbolic model checking technique and an automata theoretic model checking technique have been proposed. The automata theoretic approach builds on the existing on-the-fly verification techniques, while the symbolic approach involves reducing the new model checking problem into a standard model checking problem with propositions, and solving it using industrial model checking tools. SMT solvers are used for making this reduction. Both the proposed approaches have been proven to be correct. A tool kit has been built around the symbolic model checking approach, and has been found to give good results with fairly large state spaces. The ability to verify properties under assumptions have been built into the tool, and the technique has been proven to be effective in verifying properties over controller-plant combinations.

7.1 Future Work

Following are some of the avenues that remain to be explored:

- **Automatic extraction of PORVs from software**: As seen in Chapter 6, some controllers, like the Automatic Transmission Controller may be implemented in software, and the PORVs might not be readily observable. Currently, the extraction of PORVs from such models is done manually. It would be an interesting exercise to try and extract the PORVs from these models automatically, by static analysis of software.

- **Learning PORVs from simulation traces**: Another alternative, when the PORVs are not known beforehand, is to use machine learning techniques to learn the PORVs from simulation traces of the controller. This approach requires that the truth values of the unknown PORVs be available in the traces. Work has already begun in this direction, and it has been found that Support Vector Machines are a good choice for learning the PORVs.

- **Mining assume properties**: As seen in Chapter 5, assume properties are helpful in coming up with better counterexamples during verification. However, often, the assume properties may not be explicitly specified by the designer. If so, the assume properties can be identified from simulation traces of environment models. SVMs can be used to learn the PORVs over which the assume properties may be written. A technique for mining the assume properties themselves, given the PORVs, is still under development.

Appendix A

Proof of Theorem 1[7]

The following proof has been reproduced from [7].

Part 1("If")

Consider the infinite dag $\o = e_0 \ldots e_{k-1}(e_k \ldots e_n)^\omega$. Because $c_k = c_{k+1}$, it is obvious that $\o$ is a run dag over $\sigma = s_0 \ldots s_{k-1}(s_k \ldots s_n)^\omega$; we now show that $\o$ is accepting. Assume, to the contrary, that $\pi = p_0 p_1 \ldots$ is some infinte path in $\o$ such that $p_i \in F$ holds for infinitely many $i \in \mathbb{N}$. Because $\mathcal{A}$ is an LWAA, there exists some $m \in \mathbb{N}$ and some $q \in Q$ such that $p_i = q$ for all $i \geq m$. It follows that $(q, q) \in e_i$ hlods for all $i \geq m$, which is impossible by assumption (2) and the construction of $\o$. Therefore, $\o$ must be accepting and $\mathcal{L}(\mathcal{A}) \neq \emptyset$.

Part2("Only if")

Assume that $\sigma = s_0 s_1 \ldots \in \mathcal{L}(\mathcal{A})$, and let $\o = e_0 e_1 \ldots$ be some accepting run dag of $\mathcal{A}$ over σ. Since Q is finite, $\o$ can contain only finitely many different configurations $c_0, c_1 \ldots$, and there is some configuration $c \subseteq Q$ such that $c_i = c$ for infinitely many $i \in \mathbb{N}$. Denote by $i_0 < i_1 < \ldots$ the ω-sequence of indexes such that $c_{i_j} = c$. If there were some $q \in F$ such that $q \in e_j(q)$ for all $j \geq i_0$(implying in particular that $q \in c_j$ for all $j \geq i_0$) then $\o$ would contain an infinite path ending in a self loop at q, contradicting the assumption that $\emptyset$ is accepting. Therefore, for every $q \in F$ there must be some $j_q \geq i_0$ such that $(q,q) \notin e_{j_q}$. Choosing $k = i_0$ and $n = i_m - 1$ for some m such that $i_m > j_q$ for all(finitely many) $q \in F$, we obtain a finite run dag $\emptyset$ as required. $\square$

Bibliography

[1] Rajeev Alur, Costas Courcoubetis, Thomas Henzinger, and Pei Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Robert Grossman, Anil Nerode, Anders Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer Berlin / Heidelberg, 1993.

[2] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Trans. Softw. Eng.*, 22(3):181–201, March 1996.

[3] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:410–425, 2000. 10.1007/s100090050046.

[4] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[5] Pallab Dasgupta. How does the property checker work? In *A Roadmap for Formal Property Verification*, pages 59–100. Springer Netherlands, 2006.

[6] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the*

fourteenth annual ACM symposium on Theory of computing, STOC '82, pages 169–180, New York, NY, USA, 1982. ACM.

[7] Moritz Hammer, Alexander Knapp, and Stephan Merz. Truly on-the-fly LTL model checking. In *Proceedings of 11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, Lecture Notes in Computer Science, pages 191–205. Springer-Verlag, 2005.

[8] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[9] Thomas Henzinger Howard and Howard Wong-toi. Using hytech to synthesize control parameters for a steam boiler. In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control, LNCS 1165*, pages 265–282. Springer-Verlag, 1996.

[10] Magellan. http://www.synopsys.com/tools/verification/ functionalverification/pages/magellan.aspx.

[11] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[12] Gareth Scott Rohde. *Alternating automata and the temporal logic of ordinals*. PhD thesis, University of Illinois at Urbana-Champaign, 1997. AAI9812757.

[13] B.I. Silva, O. Stursberg, B.H. Krogh, and S. Engell. An assessment of the current status of algorithmic approaches to the verification of hybrid systems. In *Proceedings of the 40th IEEE Conference on Decision and Control, 2001.*, volume 3, pages 2867 –2874 vol.3, 2001.

[14] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[15] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of Formal Languages*, pages 389–455. Springer, 1996.

[16] Claire J. Tomlin, Ian Mitchell, Alexandre M. Bayen, and Meeko Oishi. Computational techniques for the verification of hybrid systems. In *Proceedings of the IEEE*, pages 986–1001, 2003.

[17] Moshe Y. Vardi. Alternating automata and program verification. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 1995.

[18] Jun Yan, Jian Zhang, and Zhongxing Xu. Finding relations among linear constraints. In Jacques Calmet, Tetsuo Ida, and Dongming Wang, editors, *AISC*, volume 4120 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 2006.