ASIAN'03, Lecture Notes in Computer Science (LNCS) 2896, pp 260-275. 2003

Generating Optimal Linear Temporal Logic Monitors by Coinduction

Koushik Sen, Grigore Roşu, Gul Agha
Department of Computer Science,
University of Illinois at Urbana-Champaign.
{ksen,grosu,agha}@cs.uiuc.edu

Abstract. A coinduction-based technique to generate an optimal monitor from a Linear Temporal Logic (LTL) formula is presented in this paper. Such a monitor receives a sequence of states (one at a time) from a running process, checks them against a requirements specification expressed as an LTL formula, and determines whether the formula has been violated or validated. It can also say whether the LTL formula is not monitorable any longer, i.e., that the formula can in the future neither be violated nor be validated. A Web interface for the presented algorithm adapted to extended regular expressions is available.

1 Introduction

Linear Temporal Logic (LTL) [19] is a widely used logic for specifying properties of reactive and concurrent systems. The models of LTL are infinite execution traces, reflecting the behavior of such systems as ideally always being ready to respond to requests, operating system being typical example. LTL has been mainly used to specify properties of finite-state reactive and concurrent systems, so that the full correctness of the system can be verified automatically, using model checking or theorem proving. Model checking of programs has received an increased attention from the formal methods community within the last couple of years, and several tools have emerged that directly model check source code written in Java or C [7, 26, 27]. Unfortunately, such formal verification techniques are not scalable to real-sized systems without exerting a substantial effort to abstract the system more or less manually to a model that can be analyzed.

Testing scales well, and in practice it is by far the technique most used to validate software systems. Our approach follows research which merges testing and temporal logic specification in order to achieve some of the benefits of both approaches; we avoid some of the pitfalls of ad hoc testing as well as the complexity of full-blown theorem proving and model checking. While this merger provides a scalable technique, it does result in a loss of coverage: the technique may be used to examine a single execution trace at a time, and may not be used to *prove* a system correct. Our work is based on the observation that software engineers are willing to trade coverage for scalability, so our goals is relatively conservative: we provide tools that use formal methods in a lightweight manner, use traditional programming languages or underlying executional engines (such as JVMs), are completely automatic, implement very efficient algorithms, and can help find *many* errors in programs.

Recent trends suggest that the software analysis community is interested in scalable techniques for software verification. Earlier work by Havelund and

Roşu [10] proposed a method based on merging temporal logics and testing. The Temporal Rover tool (TR) and its successor DB Rover by Drusinsky [2] have been commercialized. These tools instrument the Java code so that it can check the satisfaction of temporal logic properties at runtime. The MaC tool by Lee et al. [14, 17] has been developed to monitor safety properties in interval past time temporal logic. In works by O'Malley et al. and Richardson et al. [20, 21], various algorithms to generate testing automata from temporal logic formulae, are described. Java PathExplorer [8] is a runtime verification environment currently under development at NASA Ames. It can analyze a single execution trace. The Java MultiPathExplorer tool [25] proposes a technique to monitor all equivalent traces that can be extracted from a given execution, thus increasing the coverage of monitoring. Giannakopoulou et al. and Havelund et al. in [4, 9] propose efficient algorithms for monitoring future time temporal logic formulae, while Havelund et al. in [11] gives a technique to synthesize efficient monitors from past time temporal formulae. Roşu et al. in [23] shows use of rewriting to perform runtime monitoring of extended regular expressions. An approach similar to this paper is used to generate optimal monitors for extended regular expressions in work by Sen et al. [24].

In this paper, we present a new technique based on the modern coalgebraic method to generate optimal monitors for LTL formulae. In fact, such monitors are the minimal deterministic finite automata required to do the monitoring. Our current work makes two major contributions. First, we give a coalgebraic formalization of LTL and show that coinduction is a viable and reasonably practical method to prove monitoring-equivalences of LTL formulae. Second, building on the coinductive technique, we present an algorithm to directly generate minimal deterministic automata from an LTL formula. Such an automaton may be used to monitor good or bad prefixes of an execution trace (this notion will be rigorously formalized in subsequent sections).

We describe the monitoring as synchronous and deterministic to obtain *minimal* good or bad prefixes. However, if the cost of such monitoring is deemed too high in some application, and one is willing to tolerate some delay in discovering violations, the same technique could be applied on the traces intermittently – in which case one would not get minimal good or bad prefixes but could either bound the delay in discovering violations, or guarantee eventual discovery. We also give lower and upper bounds on the size of such automata.

The closely related work by Geilen [3] builds monitors to detect a subclass of bad and good prefixes, which are called *informative bad and good prefixes*. Using a tableau-based technique, [3] can generate monitors of exponential size for informative prefixes. In our approach, we generate minimal monitors for detecting all kinds of bad and good prefixes. This generality comes at a price: the size of our monitors can be doubly exponential in the worst case, and this complexity cannot be avoided.

One standard way to generate an optimal monitor is to use the Büchi automata construction [16] for LTL to generate a non-deterministic finite automaton, determinize it and then to minimize it. In this method, one checks only the syntactic equivalence of LTL formulae. In the coalgebraic technique that we propose as an alternative method, we make use of the *monitoring equivalence* (defined in subsequent sections) of LTL formulae. We thus obtain the minimal automaton *in a single go* and minimize the usage of computational space. Moreover, our technique is completely based on deductive methods and can be applied to any logic or algebra for which there is a suitable behavioral specification. A related application can be found in [24] in which the minimal deterministic finite automata for extended regular expressions is generated.

2 Linear Temporal Logic and Derivatives

In order to make the paper self-contained, we briefly describe classical Linear Temporal Logic over infinite traces. We use the classical definition of Linear Temporal Logic and assume a finite set AP of atomic propositions. The syntax of LTL is as follows:

$$\begin{array}{lll}\phi ::= & \text{true} \mid \text{false} \mid a \in AP \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \phi \equiv \phi \mid \phi \oplus \phi & \text{propositional} \\ & \phi\, \mathcal{U}\, \phi \mid \circ\phi \mid \Box\phi \mid \Diamond\phi & \text{temporal}\end{array}$$

The semantics of LTL is given for infinite traces. An infinite trace is an infinite sequence of program states, each state denoting the set of atomic propositions that hold at that state. The atomic propositions that hold in a given state s is given by $AP(s)$. We denote an infinite trace by ρ; $\rho(i)$ denotes the i-th state in the trace and ρ^i denotes the suffix of the trace ρ starting from the i-th state. The notion that an infinite trace ρ *satisfies* a formula ϕ is denoted by $\rho \models \phi$, and is defined inductively as follows:

$$\begin{array}{ll}\rho \models \text{ true for all } \rho & \rho \not\models \text{ false for all } \rho \\ \rho \models a \text{ iff } a \in AP(\rho(1)) & \rho \models \neg\phi \text{ iff } \rho \not\models \phi \\ \rho \models \phi_1 \vee \phi_2 \text{ iff } \rho \models \phi_1 \text{ or } \rho \models \phi_2 & \rho \models \phi_1 \wedge \phi_2 \text{ iff } \rho \models \phi_1 \text{ and } \rho \models \phi_2 \\ \rho \models \phi_1 \oplus \phi_2 \text{ iff } \rho \models \phi_1 \text{ exclusive or } \rho \models \phi_2 & \rho \models \phi_1 \rightarrow \phi_2 \text{ iff } \rho \models \phi_1 \text{ implies } \rho \models \phi_2 \\ \rho \models \phi_1 \equiv \phi_2 \text{ iff } \rho \models \phi_1 \text{ iff } \rho \models \phi_2 & \rho \models \circ\phi \text{ iff } \rho^2 \models \phi \\ \rho \models \Box\phi \text{ iff } \forall\, j \geq 1\ \rho^j \models \phi & \rho \models \Diamond\phi \text{ iff } \exists\, j \geq 1 \text{ such that } \rho^j \models \phi \\ \rho \models \phi_1\, \mathcal{U}\, \phi_2 \text{ iff there exists a } j \geq 1 \text{ such that } & \rho^j \models \phi_2 \text{ and } \forall\, 1 \leq i < j :\ \rho^i \models \phi_1\end{array}$$

The set of all infinite traces that satisfy the formula ϕ is called the language expressed by the formula ϕ and is denoted by L_ϕ. Thus, $\rho \in L_\phi$ if and only if $\rho \models \phi$. The language L_ϕ is also called the *property* expressed by the formula ϕ. We informally say that an infinite trace ρ satisfies a property ϕ iff $\rho \models \phi$.

A property in LTL can be seen as the intersection of a *safety* property and a *liveness* property [1]. A property is a liveness property if for every finite trace α there exists an infinite trace ρ such that $\alpha.\rho$ satisfies the property. A property is a safety property if for every infinite trace ρ not satisfying the property, there exists a finite prefix α such that for all infinite traces ρ', $\alpha.\rho'$ does not satisfy the property. The prefix α is called a *bad prefix* [3]. Thus, we say that a finite prefix α is a bad prefix for a property if for all infinite traces ρ, $\alpha.\rho$ does not satisfy the property. On the other hand, a *good prefix* for a property is a prefix α such that for all infinite traces ρ, $\alpha.\rho$ satisfies the property. A bad or a good prefix can also be *minimal.* We say that a bad (or a good prefix) α is minimal if α is a bad (or good) prefix and no finite prefix α' of α is bad (or good) prefix.

We use a novel coinduction-based technique to generate an optimal monitor that can detect good and bad prefixes incrementally for a given trace. The essential idea is to process, one by one, the states of a trace as these states are generated; at each step the process checks if the finite trace that we have already generated is a minimal good prefix or a minimal bad prefix. At any point, if we find that the finite trace is a minimal bad prefix, we say that the property is violated. If the finite trace is a minimal good prefix then we stop monitoring for that particular trace and say that the property holds for that trace.

At any step, we will also detect if it is not possible to monitor a formula any longer. We may stop monitoring at that point and say the trace is no longer *monitorable* and save the monitoring overhead. Otherwise, we continue by processing one more state and appending that state to the finite trace. We will see in the subsequent sections that these monitors can report a message as soon as a good or a bad prefix is encountered; therefore, the monitors are synchronous. Two more variants of the optimal monitor are also proposed; these variants can be used to efficiently monitor either bad prefixes or good prefixes (rather than both). Except in degenerate cases, such monitors have smaller sizes than the monitors that can detect both bad and good prefixes.

In order to generate the minimal monitor for an LTL formula, we will use several notions of equivalence for LTL:

Definition 1 **($\equiv$).** *We say that two LTL formulae ϕ_1 and ϕ_2 are* equivalent *i.e. $\phi_1 \equiv \phi_2$ if and only if $L_{\phi_1} = L_{\phi_2}$.*

Definition 2 **($\equiv_B$).** *For a finite trace α we say that $\alpha \not\models \phi$ iff α is bad prefix for ϕ i.e. for every infinite trace ρ it is the case that $\alpha.\rho \notin L_\phi$. Given two LTL formulae ϕ_1 and ϕ_2, ϕ_1 and ϕ_2 are said to be* bad prefix equivalent *i.e. $\phi_1 \equiv_B \phi_2$ if and only if for every finite trace α, $\alpha \not\models \phi_1$ iff $\alpha \not\models \phi_2$.*

Definition 3 **($\equiv_G$).** *For a finite trace α we say that $\alpha \models \phi$ iff α is good prefix for ϕ i.e. for every infinite trace ρ it is that case that $\alpha.\rho \in L_\phi$. Given two LTL formulae ϕ_1 and ϕ_2, ϕ_1 and ϕ_2 are said to be* good prefix equivalent *i.e. $\phi_1 \equiv_G \phi_2$ if and only if for every finite trace α, $\alpha \models \phi_1$ iff $\alpha \models \phi_2$.*

Definition 4 **($\equiv_{GB}$).** *We say that ϕ_1 and ϕ_2 are* good-bad prefix equivalent *i.e. $\phi_1 \equiv_{GB} \phi_2$ if and only if $\phi_1 \equiv_B \phi_2$ and $\phi_1 \equiv_G \phi_2$.*

Thus, for our purpose, the two non equivalent formulae $\Box\Diamond\phi$ and $\Diamond\Box\phi$ are good-bad prefix equivalent since they do not have any good or bad prefixes. Such formula are not monitorable. Note that the equivalence relation $\equiv$ is included in the equivalence relation $\equiv_{GB}$, which is in turn included in both $\equiv_G$ and $\equiv_B$. We will use the equivalences $\equiv_G$,$\equiv_B$, and $\equiv_{GB}$ to generate optimal monitors that detect good prefixes only, bad prefixes only and both bad and good prefixes respectively. We call these three equivalences *monitoring equivalences.*

2.1 Derivatives

We describe the notion of derivatives for LTL [9, 10] based on the idea of *state consumption*: an LTL formula ϕ and a state s generate another LTL formula, denoted by $\phi\{s\}$, with the property that for any finite trace α, $s\alpha \not\models \phi$ if and only if $\alpha \not\models \phi\{s\}$ and $s\alpha \models \phi$ if and only if $\alpha \models \phi\{s\}$. We define the operator $\_\{\_\}$ recursively through the following equations:

$\text{false}\ \{s\} = \text{false}$	$\text{true}\ \{s\} = \text{true}$
$p\{s\} = \text{ if } p \in AP(s) \text{ then true else false}$	$(\neg\phi)\{s\} = \neg(\phi\{s\})$
$(\phi_1 \vee \phi_2)\{s\} = \phi_1\{s\} \vee \phi_2\{s\}$	$(\phi_1 \wedge \phi_2)\{s\} = \phi_1\{s\} \wedge \phi_2\{s\}$
$(\phi_1 \rightarrow \phi_2)\{s\} = \phi_1\{s\} \rightarrow \phi_2\{s\}$	$(\phi_1 \oplus \phi_2)\{s\} = \phi_1\{s\} \oplus \phi_2\{s\}$
$(\Diamond\phi)\{s\} = \phi\{s\} \vee \Diamond\phi$	$(\Box\phi)\{s\} = \phi\{s\} \wedge \Box\phi$
$(\phi_1\ \mathcal{U}\ \phi_2)\{s\} = \phi_2\{s\} \vee (\phi_1\{s\} \wedge \phi_1\ \mathcal{U}\ \phi_2)$	

We use the decision procedure for propositional calculus by Hsiang [13] to get a canonical form for a propositional formula. The procedure reduces a tautological formula to the constant true, a false formula to the constant false, and all other formulae to canonical forms modulo associativity and commutativity. An unusual aspect of this procedure is that the canonical forms consist of exclusive or ($\oplus$) of conjunctions. The procedure is given below using equations that are shown to be Church-Rosser and terminating modulo associativity and commutativity.

$\text{true}\ \wedge \phi = \phi$	$\text{false}\ \wedge \phi = \text{false}$
$\phi \wedge \phi = \phi$	$\text{false}\ \oplus \phi = \phi$
$\phi \oplus \phi = \text{false}$	$\neg\phi = \text{true}\ \oplus \phi$
$\phi_1 \wedge (\phi_2 \oplus \phi_3) = (\phi_1 \wedge \phi_2) \oplus (\phi_1 \wedge \phi_3)$	$\phi_1 \vee \phi_2 = (\phi_1 \wedge \phi_2) \oplus \phi_1 \oplus \phi_2$
$\phi_1 \rightarrow \phi_2 = \text{true} \oplus \phi_1 \oplus (\phi_1 \wedge \phi_2)$	$\phi_1 \equiv \phi_2 = \text{true} \oplus \phi_1 \oplus \phi_2$

The exclusive or operator $\oplus$ and the $\wedge$ operator are defined as commutative and associative. The equations DERIVATIVE and PROPOSITIONAL CALCULUS when regarded as rewriting rules are terminating and Church-Rosser (modulo associativity and commutativity of $\wedge$ and $\oplus$), so they can be used as a functional procedure to calculate derivatives.

In the rest of the paper, at several places we need to check if an LTL formula is equivalent to true or false. This can be done using the tableau-based proof method for LTL; the STeP tool at Stanford [18] has such an implementation.

The following result gives a way to determine if a prefix is good or bad for a formula through derivations.

Theorem 1. *a) For any LTL formula ϕ and for any finite trace $\alpha = s_1 s_2 \ldots s_n$, α is a bad prefix for ϕ if and only if $\phi\{s_1\}\{s_2\}\ldots\{s_n\} \equiv \text{false}$. Similarly, α is a good prefix for ϕ if and only if $\phi\{s_1\}\{s_2\}\ldots\{s_n\} \equiv \text{true}$. b) The formula $\phi\{s_1\}\{s_2\}\ldots\{s_n\}$ needs $O(2^{size(\phi)})$ space to be stored.*

Proof. b): Due to the Boolean ring equations above regarded as simplification rules, any LTL formula is kept in a canonical form, which is an exclusive disjunction of conjunctions, where conjuncts have temporal operators at top. Moreover, after a series of applications of derivatives $s_1, s_2, ..., s_n$, the conjuncts in the normal form $\phi\{s_1\}\{s_2\}...\{s_n\}$ are subterms of the initial formula ϕ, each having a temporal operator at its top. Since there are at most $size(\phi)$ such subformulae, it follows that there are at most $2^{size(\phi)}$ possibilities to combine them in a conjunction. Therefore, one needs space $O(2^{size(\phi)})$ to store any exclusive disjunction of such conjunctions. This reasoning only applies on "idealistic" rewriting engines, which carefully optimize space needs during rewriting.□

In order to effectively generate optimal monitors, it is crucial to detect efficiently and as early as possible when two derivatives are equivalent. In the rest of the paper we use coinductive techniques to solve this problem. We define the operators $G: LTL \to \{\text{true}, \text{false}\}$ and $B: LTL \to \{\text{true}, \text{false}\}$ that return true if an LTL formula is equivalent ($\equiv$) to true or false respectively, and return false otherwise. We define an operator $GB: LTL \to \{0, 1, ?\}$ that checks if an LTL formula ϕ is equivalent to false or true and returns 0 or 1, respectively, and returns ? if the formula is not equivalent to either true or false.

3 Hidden Logic and Coinduction

We use circular coinduction, defined rigorously in the context of hidden logics and implemented in the BOBJ system [22, 5, 6], to test whether two LTL formulae are good-bad prefix equivalent. A particularly appealing aspect of circular coinduction in the framework of LTL formula is that it not only shows that two LTL formulae are good-bad prefix equivalent, but also generates a larger set of good-bad prefix equivalent LTL formulae which will all be used in order to generate the target monitor. Readers familiar with circular coinduction may assume the result in Theorem 4 and read Section 4 concurrently.

Hidden logic is a natural extension of algebraic specification which benefits of a series of generalizations in order to capture various natural notions of behavioral equivalence found in the literature. It distinguishes *visible* sorts for data from *hidden* sorts for states, with states *behaviorally equivalent* if and only if they are indistinguishable under a formally given set of experiments. In order to keep the presentation simple and self-contained, we define a simplified version of hidden logic together with its associated circular coinduction proof rule which is nevertheless general enough to support the definition of LTL formulae and prove that they are behaviorally good and/or bad prefix equivalent.

3.1 Algebraic Preliminaries

We assume that the reader is familiar with basic equational logic and algebra but recall a few notions in order to just make our notational conventions precise. An S-sorted signature Σ is a set of sorts/types S together with operational symbols on those, and a Σ-algebra A is a collection of sets $\{A_s \mid s \in S\}$ and a collection of functions appropriately defined on those sets, one for each operational symbol. Given an S-sorted signature Σ and an S-indexed set of variables Z, let $T_\Sigma(Z)$ denote the Σ-term algebra over variables in Z. If $V \subseteq S$ then $\Sigma\restriction_V$ is a V-sorted signature consisting of all those operations in Σ with sorts entirely in V. We may let $\sigma(X)$ denote the term $\sigma(x_1, ..., x_n)$ when the number of arguments of σ and their order and sorts are not important. If only one argument is important, then to simplify writing we place it at the beginning; for example, $\sigma(t, X)$ is a term having σ as root with no important variables as arguments except one, in this case t. If t is a Σ-term of sort s' over a special variable $*$ of sort s and A is a Σ-algebra, then $A_t: A_s \to A_{s'}$ is the usual interpretation of t in A.

3.2 Behavioral Equivalence, Satisfaction and Specification

Given disjoint sets V, H called *visible* and *hidden sorts*, a *hidden* (V, H)*-signature*, say Σ, is a many sorted $(V \cup H)$-signature. A *hidden subsignature of* Σ is a hidden (V, H)-signature Γ with $\Gamma \subseteq \Sigma$ and $\Gamma\restriction_V = \Sigma\restriction_V$. The *data signature* is $\Sigma\restriction_V$. An operation of visible result not in $\Sigma{\restriction}_V$ is called an *attribute*, and a hidden sorted operation is called a *method*.

Unless otherwise stated, the rest of this section assumes fixed a hidden signature Σ with a fixed subsignature Γ. Informally, Σ-algebras are universes of possible states of a system, i.e., "black boxes," for which one is only concerned with behavior under experiments with operations in Γ, where an experiment is an observation of a system attribute after perturbation.

A *Γ-context for sort $s \in V \cup H$* is a term in $T_\Gamma(\{* : s\})$ with one occurrence of $*$. A Γ-context of visible result sort is called a *Γ-experiment*. If c is a context for sort h and $t \in T_{\Sigma,h}$ then $c[t]$ denotes the term obtained from c by substituting t for $*$; we may also write $c[*]$ for the context itself.

Given a hidden Σ-algebra A with a hidden subsignature Γ, for sorts $s \in (V \cup H)$, we define *Γ-behavioral equivalence of* $a, a' \in A_s$ by $a \equiv^\Gamma_\Sigma a'$ iff $A_c(a) = A_c(a')$ for all Γ-experiments c; we may write $\equiv$ instead of $\equiv^\Gamma_\Sigma$ when Σ and Γ can be inferred from context. We require that all operations in Σ are compatible with $\equiv^\Gamma_\Sigma$. Note that behavioral equivalence is the identity on visible sorts, since the trivial contexts $* : v$ are experiments for all $v \in V$. A major result in hidden logics, underlying the foundations of coinduction, is that Γ-behavioral equivalence is the largest equivalence which is identity on visible sorts and which is compatible with the operations in Γ.

Behavioral satisfaction of equations can now be naturally defined in terms of behavioral equivalence. A hidden Σ-algebra A *Γ-behaviorally satisfies* a Σ-equation $(\forall X)\ t = t'$, say e, iff for each $\theta : X \to A$, $\theta(t) \equiv^\Gamma_\Sigma \theta(t')$; in this case we write $A \models^\Gamma_\Sigma e$. If E is a set of Σ-equations we then write $A \models^\Gamma_\Sigma E$ when A Γ-behaviorally satisfies each Σ-equation in E. We may omit Σ and/or Γ from $\models^\Gamma_\Sigma$ when they are clear.

A *behavioral Σ-specification* is a triple (Σ, Γ, E) where Σ is a hidden signature, Γ is a hidden subsignature of Σ, and E is a set of Σ-sentences equations. Non-data Γ-operations (i.e., in $\Gamma - \Sigma{\restriction}_V$) are called *behavioral*. A Σ-algebra A *behaviorally satisfies* a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$ iff $A \models^\Gamma_\Sigma E$, in which case we write $A \models \mathcal{B}$; also $\mathcal{B} \models e$ iff $A \models \mathcal{B}$ implies $A \models^\Gamma_\Sigma e$.

LTL can be very naturally defined as a behavioral specification. The enormous benefit of doing so is that the behavioral inference, including most importantly coinduction, provide a *decision procedure* for good-bad prefix equivalence.

Example 1. A behavioral specification of LTL defines a set of two visible sorts $V = \{\mathit{Triple}, \mathit{State}\}$, one hidden sort $H = \{\mathit{Ltl}\}$, one behavioral attribute $\mathit{GB} : \mathit{Ltl} \to \mathit{Triple}$ (defined as an operator in Subsection 2.1) and one behavioral method, the derivative, $\_\{\_\} : \mathit{Ltl} \times \mathit{State} \to \mathit{Ltl}$, together with all the other operations in Section 2 defining LTL, including the states in S which are defined as visible constants of sort *State*, and all the equations in Subsection 2.1. The sort *Triple* consists of three constants $0, 1$, and ?. We call this the *LTL behavioral specification* and we use $\mathcal{B}_{LTL/GB}$ to denote it.

Since the only behavioral operators are the test for equivalence to true and false and the derivative, it follows that the experiments have exactly the form $\mathit{GB}(*\{s_1\}\{s_2\}...\{s_n\})$, for any states s_1, s_2, ..., s_n. In other words, an experi-

ment consists of a series of derivations followed by an application of the operator GB, and therefore two LTL formulae are *behavioral equivalent* if and only if they cannot be distinguished by such experiments. Such behavioral equivalence is exactly same as good-bad prefix equivalence. In the specification of $\mathcal{B}_{LTL/GB}$ if we replace the attribute GB by B (or G), as defined in Subsection 2.1, the behavioral equivalence becomes same as bad prefix (or good prefix) equivalence. We denote such specifications by $\mathcal{B}_{LTL/B}$ (or $\mathcal{B}_{LTL/G}$). Notice that the above reasoning applies within *any algebra* satisfying the presented behavioral specification. The one we are interested in is, of course, the *free* one, whose set carriers contain exactly the LTL formulae as presented in Section 2, and the operations have the obvious interpretations. We informally call it the *LTL algebra*.
Letting $\equiv_b$ denote the behavioral equivalence relation generated on the LTL algebra, then Theorem 1 immediately yields the following important result.

Theorem 2. *If ϕ_1 and ϕ_2 are two LTL formulae then $\phi_1 \equiv_b \phi_2$ in $\mathcal{B}_{LTL/GB}$ iff ϕ_1 and ϕ_2 are good-bad prefix equivalent. Similarly, $\phi_1 \equiv_b \phi_2$ in $\mathcal{B}_{LTL/B}$ (or $\mathcal{B}_{LTL/G}$) if and only if ϕ_1 and ϕ_2 are bad prefix (or good prefix) equivalent.*

This theorem allows us to prove good-bad prefix equivalence (or bad prefix or good prefix equivalence) of LTL formulae by making use of behavioral inference in the LTL behavioral specification $\mathcal{B}_{LTL/GB}$ (or $\mathcal{B}_{LTL/B}$ or $\mathcal{B}_{LTL/G}$) including (especially) circular coinduction. The next section shows how circular coinduction works and how it can be used to show LTL formulae good-bad prefix equivalent (or bad prefix equivalent or good prefix equivalent). From now onwards we will refer $\mathcal{B}_{LTL/GB}$ simply by $\mathcal{B}$.

3.3 Circular Coinduction as an Inference Rule

In the simplified version of hidden logics defined above, the usual equational inference rules, i.e., reflexivity, symmetry, transitivity, substitution and congruence [22] are all sound for behavioral satisfaction. However, equational reasoning can derive only a very limited amount of interesting behavioral equalities. For that reason, *circular coinduction* has been developed as a very powerful automated technique to show behavioral equivalence. We let $\Vdash$ denote the relation being defined by the equational rules plus circular coinduction, for deduction from a specification to an equation.

Before formally defining circular coinduction, we give the reader some intuitions by duality to structural induction. The reader who is only interested in using the presented procedure or who is not familiar with structural induction, can skip this paragraph. Inductive proofs show equality of terms $t(x), t'(x)$ over a given variable x (seen as a constant) by showing $t(\sigma(x))$ equals $t'(\sigma(x))$ for all σ in a basis, while circular coinduction shows terms t, t' behaviorally equivalent by showing equivalence of $\delta(t)$ and $\delta(t')$ for all behavioral operations δ. Coinduction applies behavioral operations at the top, while structural induction applies generator/constructor operations at the bottom. Both induction and circular coinduction assume some "frozen" instances of t, t' equal when checking the inductive/coinductive step: for induction, the terms are frozen at the bottom by replacing the induction variable by a constant, so that no other terms can be placed beneath the induction variable, while for coinduction, the terms are frozen at the top, so that they cannot be used as subterms of other terms (with some important but subtle exceptions which are not needed here; see [6]).

Freezing terms at the top is elegantly handled by a simple trick. Suppose every specification has a special visible sort b, and for each (hidden or visible) sort s in the specification, a special operation $[\_] : \; s \to b$. No equations are assumed for these operations and no user defined sentence can refer to them; they are there for technical reasons. Thus, with just the equational inference rules, for any behavioral specification $\mathcal{B}$ and any equation $(\forall X)\ t = t'$, it is necessarily the case that $\mathcal{B} \ \Vdash (\forall X)\ t = t'$ iff $\mathcal{B} \ \Vdash (\forall X)\ [t] = [t']$. The rule below preserves this property. Let the sort of t, t' be hidden; then
Circular Coinduction:

$$\frac{\mathcal{B} \cup \{(\forall X)\ [t] = [t']\} \ \Vdash (\forall X, W)\ [\delta(t, W)] = [\delta(t', W)], \text{for all appropriate } \delta \in \Gamma}{\mathcal{B} \ \Vdash (\forall X)\ t = t'}$$

We call the equation $(\forall X)\ [t] = [t']$ added to $\mathcal{B}$ a **circularity**; it could just as well have been called a coinduction hypothesis or a co-hypothesis, but we find the first name more intuitive because from a coalgebraic point of view, coinduction is all about finding circularities.

Theorem 3. *The usual equational inference rules together with* Circular Coinduction *are sound. That means that if $\mathcal{B} \ \Vdash (\forall X)\ t = t'$ and sort$(t, t') \neq b$, or if $\mathcal{B} \ \Vdash (\forall X)\ [t] = [t']$, then $\mathcal{B} \ \models\!\equiv (\forall X)\ t = t'$.*

Circular coinductive rewriting[5, 6] iteratively rewrites proof tasks to their normal forms followed by an one step coinduction if needed. Since the rules in $\mathcal{B}_{LTL/GB}$, $\mathcal{B}_{LTL/B}$, and $\mathcal{B}_{LTL/G}$ are ground Church-Rosser and terminating, this provides us with a decision procedure for good-bad prefix equivalence, bad prefix equivalence, and good prefix equivalence of LTL formulae respectively.

Theorem 4. *If ϕ_1 and ϕ_2 are two LTL formulae, then $\phi_1 \equiv_{GB} \phi_2$ if and only if $\mathcal{B}_{LTL/GB} \ \Vdash \ \phi_1 = \phi_2$. Similarly, if ϕ_1 and ϕ_2 are two LTL formulae, then $\phi_1 \equiv_B \phi_2$ (or $\phi_1 \equiv_G \phi_2$) if and only if $\mathcal{B}_{LTL/B} \ \Vdash \ \phi_1 = \phi_2$ (or $\mathcal{B}_{LTL/G} \ \Vdash \ \phi_1 = \phi_2$). Moreover,* circular coinductive rewriting *provides us with a decision procedure for good-bad prefix equivalence, bad prefix equivalence, and good prefix equivalence of LTL formulae.*

Proof. By soundness of behavioral reasoning (Theorem 3), one implication follows immediately via Theorem 2. For the other implication, assume that ϕ_1 and ϕ_2 are good-bad prefix equivalent (or good prefix or bad prefix equivalent, respectively) and that the equality $\phi_1 = \phi_2$ is not derivable from $\mathcal{B}_{LTL/GB}$ (or $\mathcal{B}_{LTL/G}$ or $\mathcal{B}_{LTL/B}$, respectively). By Theorem 1, the number of formulae into which any LTL formula can be derived via a sequence of events is finite, which means that the total number of equalities $\phi_1' = \phi_2'$ that can be derived via the circular coinduction rule is also finite. That implies that the only reason for which the equality $\phi_1 = \phi_2$ *cannot* be proved by circular coinduction is because it is in fact *disproved* by some experiment, which implies the existance of some events a_1, ..., a_n such that $GB(\phi_1\{a_1\} \cdots \{a_n\}) \neq GB(\phi_2\{a_1\} \cdots \{a_n\})$ (or the equivalent ones for B or G). However, this is obviously a contradition because if ϕ_1 and ϕ_2 are good-bad (or good or bad) prefix equivalent that so are $\phi_1\{a_1\} \cdots \{a_n\}$ and $\phi_2\{a_1\} \cdots \{a_n\}$, and GB (or G or B) preserve this equivalence.

4 Generating Optimal Monitors by Coinduction

We now show how one can use the set of circularities generated by applying the circular coinduction rules in order to generate, from any LTL formula, an optimal monitor that can detect both good and bad prefixes. The optimal monitor thus generated will be a minimal deterministic finite automaton containing two final states true and false. We call such a monitor *GB-automaton.* We conclude the section by modifying the algorithm to generate smaller monitors that can detect either bad or good prefixes. We call such monitors *B-automaton* and *G-automaton* respectively. The main idea behind the algorithm is to associate states in GB-automaton to LTL formulae obtained by deriving the initial LTL formula; when a new LTL formula is generated, it is tested for good-bad prefix equivalence with all the other already generated LTL formulae by using the coinductive procedure presented in the previous section. A crucial observation which significantly reduces the complexity of our procedure is that once a good-bad prefix equivalence is proved by circular coinductive rewriting, the entire set of circularities accumulated represent good-bad prefix equivalent LTL formulae. These can be used to quickly infer the other good-bad prefix equivalences, without having to generate the same circularities over and over again.

Since BOBJ does not (yet) provide any mechanism to return the set of circularities accumulated after proving a given behavioral equivalence, we were unable to use BOBJ to implement our optimal monitor generator. Instead, we have implemented our own version of coinductive rewriting engine for LTL formulae, which is described below.

We are given an initial LTL formula ϕ_0 over atomic propositions P. Then $\sigma = 2^P$ is the set of possible states that can appear in an execution trace; note that σ will be the set of alphabets in the GB-automaton. Now, from ϕ_0 we want to generate a GB-automaton $D = (S, \sigma, \delta, s_0, \{\text{true}, \text{false}\})$, where S is the set of states of the GB-automaton, $\delta : S \times \sigma \to S$ is the transition function, s_0 is the initial state of the GB-automaton, and $\{\text{true}, \text{false}\} \subseteq S$ is the set of final states of the DFA. The coinductive rewriting engine explicitly accumulates the proven circularities in a set. The set is initialized to an empty set at the beginning of the algorithm. It is updated with the accumulated circularities whenever we prove good-bad prefix equivalence of two LTL formulae in the algorithm. The algorithm maintains the set of states S in the form of non good-bad prefix equivalent LTL formulae. At the beginning of the algorithm S is initialized with two elements, the constant formulae true and false. Then, we check if the initial LTL formula ϕ_0 is equivalent to true or false. If ϕ_0 is equivalent to true or false, we set s_0 to true or false respectively and return D as the GB-automaton. Otherwise, we set s_0 to ϕ_0, add ϕ_0 to the set S, and invoke the procedure **dfs** (see Fig 1) on ϕ_0.

The procedure **dfs** generates the derivatives of a given formula ϕ for all $x \in \sigma$ one by one. A derivative $\phi_x = \phi\{x\}$ is added to the set S, if the set does not contain any LTL formula good-bad prefix equivalent to the derivative ϕ_x. We then extend the transition function by setting $\delta(\phi, x) = \phi_x$ and recursively invoke **dfs** on ϕ_x. On the other hand, if an LTL formula ϕ' equivalent to the derivative already exists in the set S, we extend the transition function by setting

$\delta(\phi, x) = \phi'$. To check if an LTL formula, good-bad prefix equivalent to the derivative ϕ_x, already exists in the set S, we sequentially go through all the elements of the set S and try to prove its good-bad prefix equivalence with ϕ_x. In testing the equivalence we first add the set of circularities to the initial $\mathcal{B}_{LTL/GB}$. Then we invoke the coinductive procedure. If for some LTL formula $\phi' \in S$, we are able to prove that $\phi' \equiv_{GB} \phi_x$ i.e $\mathcal{B}_{LTL/GB} \cup Eq_{\text{all}} \cup Eq_{\text{new}} \Vdash \phi' = \phi_x$, then we add the new equivalences Eq_{new}, created by the coinductive procedure, to the set of circularities. Thus we reuse the already proven good-bad prefix equivalences in future proofs.

```
S ← {true, false}
dfs(φ)
begin
    foreach $x \in \sigma$ do
        $\phi_x \leftarrow \phi\{x\}$;
        if $\exists\phi' \in S$ such that $\mathcal{B}_{LTL/GB} \cup Eq_{\text{all}} \cup Eq_{\text{new}} \Vdash \phi' = \phi_x$ then
            $\delta(\phi, x) = \phi'$;   $Eq_{\text{all}} \leftarrow Eq_{\text{all}} \cup Eq_{\text{new}}$
        else $S \leftarrow S \cup \{\phi_x\}$;   $\delta(\phi, x) = \phi_x$;   dfs($\phi_x$); fi
    endfor
end
```

Fig. 1. LTL to optimal monitor generation algorithm

The GB-automaton generated by the procedure **dfs** may now contain some states which are non-final and from which the GB-automaton can never reach a final state. We remove these redundant states by doing a breadth first search in backward direction from the final states. This can be done in time linear in the size of the GB-automaton. If the resultant GB-automaton contains the initial state s_0 then we say that the LTL formula is monitorable. That is for the LTL formula to be monitorable there must be path from the initial state to a final state i.e. to true or false state. Note that the GB-automaton may now contain non-final states from which there may be no transition for some $x \in \sigma$. Also note that no transitions are possible from the final states.

The correctness of the algorithm is given by the following theorem.

Theorem 5. *If D is the GB-automaton generated for a given LTL formula ϕ by the above algorithm then*
1) $\mathcal{L}(D)$ is the language of good and bad prefixes of ϕ,
2) D is the minimal deterministic finite automaton accepting the good and bad prefixes of ϕ.

Proof. 1) Suppose $s_1 s_2 \ldots s_n$ be a good or bad prefix of ϕ. Then by Theorem 1, $GB(\phi\{s_1\}\{s_2\}\ldots\{s_n\}) \in \{0, 1\}$. Let $\phi_i = \phi\{s_1\}\{s_2\}\ldots\{s_i\}$; then $\phi_{i+1} = \phi_i\{a_{i+1}\}$. To prove that $s_1 s_2 \ldots s_n \in \mathcal{L}(D)$, we use induction to show that for each $1 \leq i \leq n$, $\phi_i \equiv_{GB} \delta(\phi, s_1 s_2 \ldots s_i)$. For the base case if $\phi_1 \equiv_{GB} \phi\{s_1\}$ then **dfs** extends the transition function by setting $\delta(\phi, s_1) = \phi$. Therefore, $\phi_1 \equiv_{GB} \phi = \delta(\phi, s_1)$. If $\phi_1 \not\equiv_{GB} \phi$ then **dfs** extends δ by setting $\delta(\phi, s_1) = \phi_1$. So $\phi_1 \equiv_{GB} \delta(\phi, s_1)$ holds in this case also. For the induction step let us assume that $\phi_i \equiv_{GB} \phi' = \delta(\phi, s_1 s_2 \ldots s_i)$. If $\delta(\phi', s_{i+1}) = \phi''$ then from the **dfs** procedure we can see that $\phi'' \equiv_{GB} \phi'\{s_{i+1}\}$. However, $\phi_i\{s_{i+1}\} \equiv_{GB} \phi'\{s_{i+1}\}$, since $\phi_i \equiv_{GB} \phi'$ by induction hypothesis. So $\phi_{i+1} \equiv_{GB} \phi'' = \delta(\phi', s_{i+1}) = \delta(\phi, s_1 s_2 \dots s_{i+1})$. Also notice $GB(\phi_n \equiv_{GB} \delta(\phi, s_1 s_2 \dots s_n)) \in \{0, 1\}$; this implies that $\delta(\phi, s_1 s_2 \dots s_n)$ is a final state and hence $s_1 s_2 \dots s_n \in \mathcal{L}(D)$.

Now suppose $s_1 s_2 \dots s_n \in \mathcal{L}(D)$. The proof that $s_1 s_2 \dots s_n$ is a good or bad prefix of ϕ goes in a similar way by showing that $\phi_i \equiv_{GB} \delta(\phi, s_1 s_2 \dots s_i)$.
2) If the automaton D is not minimal then there exists at least two states p and q in D such that p and q are equivalent [12] i.e. $\forall w \in \sigma^* : \delta(p, w) \in F$ if and only if $\delta(q, w) \in F$, where F is the set of final states. This means, if ϕ_1 and ϕ_2 are the LTL formulae associated with p and q respectively in **dfs** then $\phi_1 \equiv_{GB} \phi_2$. But **dfs** ensures that no two LTL formulae representing the states of the automaton are good-bad prefix equivalent. So we get a contradiction. □

The GB-automaton thus generated can be used as a monitor for the given LTL formula. If at any point of monitoring we reach the state true in the GB-automaton we say that the monitored finite trace satisfies the LTL formula. If we reach the state false we say that the monitored trace violates the LTL formula. If we get stuck at some state i.e. we cannot take a transition, we say that the monitored trace is not monitorable. Otherwise we continue monitoring by consuming another state of the trace.

In the above procedure if we use the specification $\mathcal{B}_{LTL/B}$ (or $\mathcal{B}_{LTL/G}$) instead of $\mathcal{B}_{LTL/GB}$ and consider false (or true) as the only final state, we get a B-automaton (or G-automaton). These automata can detect either bad or good prefixes. Since the final state is either false or true the procedure to remove redundant states will result in smaller automata compared to the corresponding GB-automaton.

We have an implementation of the algorithm adapted to extended regular expressions which is available for evaluation on the internet via a CGI server reachable from `http://fsl.cs.uiuc.edu/rv/`.

5 Time and Space Complexity

Any possible derivative of an LTL formula ϕ, in its normal form, is an *exclusive or* of *conjunctions* of temporal subformulae (subformulae having temporal operators at the top) in ϕ. The number of such temporal subformulae is $O(m)$, where m is the size of ϕ. Hence, by counting argument, the number of possible conjuncts is $O(2^m)$. The number of possible exclusive ors of these conjuncts is then $O(2^{2^m})$. Therefore, the number of possible distinct derivatives of ϕ is $O(2^{2^m})$. Since the number states of the GB-automaton accepting good and bad prefixes of ϕ cannot be greater than the number of derivatives, 2^{2^m} is an upper bound on the number of possible states of the GB-automaton. Hence, the size of the GB-automaton is $O(2^{2^m})$. Thus we get the following lemma:

Lemma 1. *The size of the minimal GB-automaton accepting the good and bad prefixes of any LTL formula of size m is $O(2^{2^m})$.*

For the lower bound on the size of the automata we consider the language

$$L_k = \{\sigma \# w \# \sigma' \$ w \mid w \in \{0, 1\}^k \text{ and } \sigma, \sigma' \in \{0, 1, \#\}^*\}.$$

This language was previously used in several works [15, 16, 23] to prove lower bounds. The language can be expressed by the LTL formula [16] of size $O(k^2)$:

$$\phi_k = [(\neg\$)\,\mathcal{U}\,(\$\,\mathcal{U}\,\circ\Box(\neg\$))]\wedge\Diamond[\#\wedge\circ^{n+1}\#\wedge\bigwedge_{i=1}^{n}((\circ^i 0\wedge\Box(\$\to\circ^i 0))\vee(\circ^i 1\wedge\Box(\$\to\circ^i 1)))].$$

For this LTL formula the following result holds.

Lemma 2. *Any GB-automaton accepting good and bad prefixes of ϕ_k will have size $\Omega(2^{2^k})$.*

Proof: In order to prove the lower bound, the following equivalence relation on strings over $(0+1+\#)^\star$ is useful. For a string $\sigma \in (0+1+\#)^\star$, define $S(\sigma) = \{w \in (0+1)^k \mid \exists\lambda_1,\lambda_2.\ \lambda_1\#w\#\lambda_2 = \sigma\}$. We will say that $\sigma_1 \equiv_k \sigma_2$ iff $S(\sigma_1) = S(\sigma_2)$. Now observe that the number of equivalence classes of $\equiv_k$ is 2^{2^k}; this is because for any $S \subseteq (0+1)^k$, there is a σ such that $S(\sigma) = S$.

We will prove this lower bound by contradiction. Suppose A is a GB-automaton that has a number of states less than 2^{2^k} for the LTL formula ϕ_k. Since the number of equivalence classes of $\equiv_k$ is 2^{2^k}, by pigeon hole principle, there must be two strings $\sigma_1 \not\equiv_k \sigma_2$ such that the state of A after reading $\sigma_1\$$ is the same as the state after reading $\sigma_2\$$. In other words, A will reach the same state after reading inputs of the form $\sigma_1\$w$ and $\sigma_2\$w$. Now since $\sigma_1 \not\equiv_k \sigma_2$, it follows that $(S(\sigma_1)\setminus S(\sigma_2) \cup (S(\sigma_2)\setminus S(\sigma_1)) \neq \emptyset$. Take $w \in (S(\sigma_1)\setminus S(\sigma_2) \cup (S(\sigma_2)\setminus S(\sigma_1))$. Then clearly, exactly one out of $\sigma_1\$w$ and $\sigma_2\$w$ is in L_k, and so A gives the wrong answer on one of these inputs. Therefore, A is not a correct GB-automaton. □

Combining the above two results we get the following theorem.

Theorem 6. *The size of the minimal GB-automaton accepting the good and bad prefixes of any LTL formula of size m is $O(2^{2^m})$ and $\Omega(2^{2^{\sqrt{m}}})$.*

The space and time complexity of the algorithm is given by the following:

Theorem 7. *The LTL to optimal monitor generation algorithm requires $2^{O(2^m)}$ space and $c2^{O(2^m)}$ time for some constant c.*

Proof: The number of distinct derivatives of an LTL formula of size m can be $O(2^{2^m})$. Each such derivative can be encoded in space $O(2^m)$. So the number of circularities that are generated in the algorithm can consume $O(2^{2^m}2^m 2^m)$ space. The space required by the algorithm is thus $2^{O(2^m)}$. □

The number of iterations that the algorithm makes is less than the number of distinct derivatives. In each iteration the algorithm generates a set of circularities that can be at most $2^{O(2^m)}$. So the total time taken by the algorithm is $c2^{O(2^m)}$ for some constant c.

6 Conclusion and Future Work

In this paper we give a behavioral specification for LTL, which has the appealing property that two LTL formulae are *equivalent with respect to monitoring* if and only if they are indistinguishable under carefully chosen experiments. To our knowledge, this is the first coalgebraic formalization of LTL. The major benefit of this formalization is that one can use *coinduction* to prove LTL formulae monitoring-equivalent, which can further be used to generate optimal LTL monitors on a single go. As future work we want to apply our coinductive techniques to generate monitors for other logics.

7 Acknowledgements

The work is supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract number F30602-00-2-0586, the DARPA IXO NEST Program, contract number F33615-01-C-1907), the ONR Grant N00014-02-1-0715, the Motorola Grant MOTOROLA RPS #23 ANT, and the joint NSF/NASA grant CCR-0234524. We would like to thank Klaus Havelund and Predrag Tosic for reading a previous version of this paper and giving us valuable feedback and Howard Barringer for very helpful discussions, valuable feedback on the paper and pointers to references.

References

1. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.
2. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.
3. M. Geilen. On the construction of monitors for temporal logic properties. In K. Havelund and G. Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier, 2001.
4. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
5. J. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. In *Proceedings, Automated Software Engineering '00*, pages 123–131. IEEE, 2000. (Grenoble, France).
6. J. Goguen, K. Lin, and G. Rosu. Conditional circular coinductive rewriting with case analysis. In *Recent Trends in Algebraic Development Techniques (WADT'02)*, Lecture Notes in Computer Science, to appear, Frauenchiemsee, Germany, September 2002. Springer-Verlag.
7. K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, Apr. 2000.
8. K. Havelund and G. Roşu. Java PathExplorer – A Runtime Verification Tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, Montreal, Canada, June 18 - 21, 2001.
9. K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.

10. K. Havelund and G. Roşu. *Runtime Verification 2002*, volume 70(4) of *Electronic Notes in Theoretical Computer Science.* Elsevier Science, 2002. Proceedings of a *Computer Aided Verification (CAV'02)* satellite workshop.
11. K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.
12. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.
13. J. Hsiang. Refutational theorem proving using term rewriting systems. *Artificial Intelligence*, 25:255–300, 1985.
14. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science.* Elsevier Science, 2001.
15. O. Kupferman and M. Y. Vardi. Freedom, Weakness, and Determinism: From linear-time to branching-time. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 81–92, 1998.
16. O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Proceedings of the Conference on Computer-Aided Verification*, 1999.
17. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
18. Z. Manna, N. Bjørner, and A. B. et al. An update on STeP: Deductive-algorithmic verification of reactive systems. In *Tool Support for System Specification, Development and Verification*, LNCS. Springer-Verlag, 1998.
19. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety.* Springer-Verlag N.Y., Inc., 1995.
20. T. O'Malley, D. Richardson, and L. Dillon. Efficient Specification-Based Oracles for Critical Systems. In *In Proceedings of the California Software Symposium*, 1996.
21. D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-Based Test Oracles for Reactive Systems. In *Proceedings of the Fourteenth International Conference on Software Engineering, Melbourne, Australia*, pages 105–118, 1992.
22. G. Roşu. *Hidden Logic.* PhD thesis, University of California at San Diego, 2000.
23. G. Roşu and M. Viswanathan. Testing extended regular language membership incrementally by rewriting. In *Rewriting Techniques and Applications (RTA'03)*, Lecture Notes in Computer Science, to appear. Springer-Verlag, 2003.
24. K. Sen and G. Roşu. Generating Optimal Monitors for Extended Regular Expressions. In *Proceedings of Runtime Verification (RV'03) (To appear)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science.* Elsevier Science, 2003.
25. K. Sen, G. Roşu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. In *ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC '03) (To Appear)*, Helsinki, Finland, 2003.
26. S. D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–244. Springer, 2000.
27. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering.* IEEE CS Press, Sept. 2000.