Report No. UIUCDCS-R-2005-2530 UILU-ENG-2005-1722

Efficient Monitoring of ω-languages

by

Marcelo d'Amorim and Grigore Roşu

March 2005

Efficient Monitoring of ω-languages

Marcelo d'Amorim and Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin, Urbana, IL, 61801, USA
{damorim, grosu}@uiuc.edu

Abstract. We present a technique for generating efficient monitors for ω-regular-languages. We show how Büchi automata can be reduced in size and transformed into special, statistically optimal nondeterministic finite state machines, called *binary transition tree finite state machines (BTT-FSMs)*, which recognize precisely the minimal bad prefixes of the original ω-regular-language. The presented technique is implemented as part of a larger monitoring framework and is available for download.

1 Introduction

There is increasing recent interest in the area of *runtime verification* [15, 29], which is an area which aims at bridging testing and formal verification. In runtime verification, monitors are generated from system requirements. These monitors observe online executions of programs and check them against requirements. The checks can be either *precise*, with the purpose of detecting existing errors in the observed execution trace, or *predictive*, with the purpose of detecting errors that have not occurred in the observed execution but were "close to happening" and could possibly occur in other executions of the (typically concurrent) system. Runtime verification can be used either during testing, to catch errors, or during operation, to detect and recover from errors. Since monitoring unavoidably adds runtime overhead to a monitored program, an important technical challenge in runtime verification is that of synthesizing *efficient* monitors from specifications.

Requirements of systems can be expressed in a variety of formalisms, not all of them necessarily easily monitorable. As perhaps best shown by the immense success of programming languages like Perl and Python, regular patterns can be easily devised and understood by ordinary software developers. ω-regular-languages [4, 30] add infinite repetitions to regular languages, thus allowing one to specify properties of reactive systems [22]. The usual acceptance condition in finite state machines (FSM) needs to be modified in order to recognize infinite words, thus leading to Büchi automata [7]. Logics like linear temporal logics (LTL) [22] often provide a more intuitive and compact means to specify system requirements than ω-regular patterns. It is therefore not surprising that a large amount of work has been dedicated to generating (small) Büchi automata from, and verifying programs against, LTL formulae [13, 30, 8, 11].

Based on the belief that ω-languages represent a powerful and convenient formalism to express requirements of systems, we address the problem of generating efficient monitors from ω-languages expressed as Büchi automata. More precisely, we generate monitors that recognize the *minimal bad prefixes* [21] of such languages. A bad prefix is a *finite* sequence of events which cannot be the prefix of any accepting trace. A bad prefix is minimal if it does not contain any other bad prefix. Therefore, our goal is to develop efficient techniques that read events of the monitored program incrementally, and precisely detect when a *bad prefix* has occurred. Dual to the notion of bad prefix is that of a good prefix, meaning that the trace will be accepted for any infinite extension of the prefix.

We present a technique that transforms a Büchi automaton into a special (nondeterministic) finite state machine, called a *binary transition tree finite state machine (BTT-FSM)*, that can be used as a monitor: by maintaining a set of possible states which is updated as events are available. A sequence of events is a bad prefix iff the set of states in the monitor becomes empty. One interesting aspect of the generated monitors is that they may contain a special state, called *neverViolate*, which, once reached, indicates that the specification is *not monitorable* from that moment on. That can mean either that the specification has been fulfilled (e.g., a specification $\diamond(x > 0)$ becomes fulfilled when x is first seen larger than 0), or that

from that moment on, there will always be some possible continuation of the execution trace. For example, the monitor generated for $\Box(a \rightarrow \diamond b)$ will have exactly one state, *neverViolate*, reflecting the intuition that liveness properties cannot be monitored.

As usual, a program state is abstracted as a set of relevant atomic predicates that hold in that state. However, in the context of monitoring, the evaluation of these atomic predicates can be the most expensive part of the entire monitoring process. One predicate, for example, can say whether the vector $v[1...1000]$ is sorted. Assuming that each atomic predicate has a given evaluation cost and a given probability to hold, which can be estimated apriori either by static or by dynamic analysis, the BTT-FSM generated from a Büchi automaton executes a "conditional program", called a *binary transition tree (BTT)*, evaluating atomic predicates *by need* in each state in order to statistically optimize the decision to which states to transit. One such BTT is shown in Fig. 2.

The work presented in this paper is part of a larger project focusing on *monitoring-oriented programming (MOP)* [5, 6] which is a tool-supported software development framework in which monitoring plays a foundational role. MOP aims at reducing the gap between specification and implementation by integrating the two through monitoring: specifications are checked against implementations at runtime, and recovery code is provided to be executed when specifications are violated. MOP is specification-formalism-independent: one can add one's favorite or domain-specific requirements formalism via a generic notion of *logic plug-in*, which encapsulates a formal logical syntax plus a corresponding monitor synthesis algorithm. The work presented in this paper is implemented and provided as part of the LTL logic plugin in our MOP framework. It is also available for online evaluation and download on the MOP website [1].

Some Background and Related Work. Automata theoretic model-checking is a major application of Büchi automata. Many model-checkers, including most notably SPIN [17], use this technique. So a significant effort has been put into the construction of small Büchi automata from LTL formulae. Gerth *et al.* [13] show a tableaux procedure to generate on-the-fly Büchi automata of size $2^{O(|\varphi|)}$ from LTL formulae φ. Kesten *et al.* [19] describe a backtracking algorithm, also based on tableaux, to generate Büchi automata from formulae involving both past and future modalities (PTL), but no complexity results are shown. It is known that LTL model-checking is PSPACE-complete [28] and PTL is as expressive and as hard as LTL [23], though exponentially more succinct [23]. Recently, Gastin and Oddoux [11] showed a procedure to generate standard Büchi automata of size $2^{O(|\varphi|)}$ from PTL via alternating automata. Several works [8, 13] describe simplifications to reduce the size of Büchi automata. Algebraic simplifications can also be applied *apriori* on the LTL formula. For instance, $a \,\mathcal{U}\, b \wedge c \,\mathcal{U}\, b \equiv (a \wedge c) \,\mathcal{U}\, b$ is a valid LTL congruence that will reduce the size of the generated Büchi automaton. All these techniques producing small automata are very useful in our monitoring context because the smaller the original Büchi automaton for the ω-language, the smaller the BTT-FSM. Simplifications of the automaton *with respect to monitoring* are the central subject of this paper.

Kupferman *et al.* [21] classify safety according to the notion of *informativeness*. Informative prefixes are those that "tell the whole story": they witness the violation (or validation) of a specification. Unfortunately, not all bad prefixes are informative; e.g., the language denoted by $\Box(q \vee \circ(\Box(p))) \wedge \Box(r \vee \circ(\Box(\neg p)))$ does not include any word whose prefix is $\{q, r\}, \{q\}, \{!p\}$. This is a (minimal) bad but not informative prefix, since it does not witness the violation taking place in the next state. One can use the construction described in [21] to build an automaton of size $O(2^{2^{|\varphi|}})$ which recognizes all bad prefixes but, unfortunately, this automaton may be too large to be stored. Our fundamental construction is similar in spirit to theirs but we do not need to apply a subset construction on the input Büchi since we already maintain the set of possible states that the running program can be in. Geilen [12] shows how Büchi automata can be turned into monitors. The construction builds a tableaux similar to [13] in order to produce an FSM of size $O(2^{|\varphi|})$ for recognizing informative good prefixes. Here we detect all the minimal bad prefixes, rather than just the informative ones. Unlike model-checking where a user hopes to see a counter-example that witnesses the violation, when monitoring critical applications one might want to observe a problem as soon as it occurs.

The technique illustrated here is implemented as a plug-in in the MOP *runtime verification (RV)* framework [5, 6]. Other RV tools include JAVA-MAC [20], JPAX [14], JMPAX [27], and EAGLE [3]. JAVA-MAC uses a special interval temporal logic as the specification language, while JPAX and JMPAX support variants of LTL. These systems instrument the JAVA bytecode to emit events to an external monitor observer. JPAX was used to analyze NASA's K9 Mars Rover code [2]. JMPAX extends JPAX with predictive capa-

bilities. EAGLE is a finite-trace temporal logic and tool for runtime verification, defining a logic similar to the μ-calculus with data-parameterization.

2 Preliminaries: Büchi Automata

Büchi automata and their ω-languages have been studied extensively during the past decades. They are well suited to program verification because one can check satisfaction of properties represented as Büchi automata statically against transition systems [30, 7]. LTL is an important but proper subset of ω-languages.

Definition 1. *A (nondeterministic) standard* ***Büchi automaton*** *is a tuple $\langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$, where Σ is an* ***alphabet****, S is a set of* ***states****, $\delta \colon S \times \Sigma \to 2^S$ is a* ***transition function****, $S_0 \subseteq S$ is the set of* ***initial states****, and $\mathcal{F} \subseteq S$ is a set of* ***accepting*** *states.*

In practice, Σ typically refers to events or actions in a system to be analyzed.

Definition 2. *A Büchi automaton $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ is said to* ***accept*** *an infinite word $\tau \in \Sigma^\omega$ iff there is some* ***accepting run*** *in the automaton, that is, a map $\rho \colon \mathcal{N}at \to S$ such that $\rho_0 \in S_0$, $\rho_{i+1} \in \delta(\rho_i, \tau_i)$ for all $i \geq 0$, and $inf(\rho) \cap \mathcal{F} \neq \emptyset$, where $inf(\rho)$ contains the states occurring infinitely often in ρ. The* ***language of*** *$\mathcal{A}$, $\mathcal{L}(\mathcal{A})$, consists of all words it accepts.*

Therefore, ρ can be regarded as an infinite path in the automaton that starts with an initial state and contains at least one accepting state appearing infinitely often in the trace. Fig. 1 shows a nondeterministic Büchi automaton for the ω-regular expression $(a+b)^*b^\omega$ that contains all the infinite words over a and b with finitely many as.

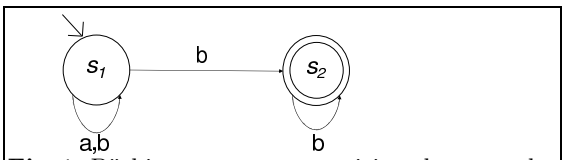

Fig. 1. Büchi automaton recognizing the ω-regular expression $(a+b)^*b^\omega$

Definition 3. *Let $\mathcal{L}(\mathcal{A})$ be the language of a Büchi automaton $\mathcal{A}=\langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$. A finite word $x \in \Sigma^*$ is a* ***bad prefix of*** *$\mathcal{A}$ iff for any $y \in \Sigma^\omega$ the concatenation $xy \notin \mathcal{L}(\mathcal{A})$. A bad prefix is* ***minimal*** *if no other bad prefix is a prefix of it.*

Therefore, no bad prefix of the language of a Büchi automaton can be extended to an accepted word. Similarly to [7], from now on we may tacitly assume that Σ is defined in terms of propositions over atoms. For instance, the self-transitions of s_1 in Fig. 1 can be represented as one self-transition, $a \vee b$.

3 Multi-Transitions and Binary Transition Trees

Büchi automata cannot be used unchanged as monitors. For the rest of the paper we explore structures suitable for monitoring as well as techniques to transform Büchi automata into such structures. Deterministic multi-transitions (*MT*) and binary-transition trees (*BTTs*) were introduced in [16, 26]. In this section we extend their original definitions with nondeterminism.

Definition 4. *Let S and A be sets of* ***states*** *and* ***atomic predicates****, respectively, and let P_A denote the set of* ***propositions*** *over atoms in A, using the usual boolean operators. If $\{s_1, s_2, ..., s_n\} \subseteq S$ and $\{p_1, p_2, ..., p_n\} \subseteq P_A$, we call the n-tuple $[p_1 \colon s_1, p_2 \colon s_2, ..., p_n \colon s_n]$ a* ***(nondeterministic) multi-transition (MT)*** *over P_A and S. Let $MT(P_A, S)$ denote the set of MTs over P_A and S.*

Intuitively, if a monitor is in a state associated to an MT $[p_1 \colon s_1, p_2 \colon s_2, \ldots, p_n \colon s_n]$ then $p_1, p_2, ..., p_n$ can be regarded as guards allowing the monitor to nondeterministically transit to one of the states $s_1, s_2, ..., s_n$.

Definition 5. *Maps $\theta \colon A \to \{true, false\}$ are called* ***A-events****, or simply events. Given an A-event θ, we define its* ***multi-transition extension*** *as the map $\theta_{MT} : MT(P_A, S) \to 2^S$, where $\theta_{MT}([p_1 \colon s_1, p_2 \colon s_2, ..., p_n \colon s_n]) = \{s_i \mid \theta \models p_i\}$.*

The role of A-events is to transmit the monitor information regarding the running program. In any program state, the map θ assigns atomic propositions to *true* iff they hold in that state, otherwise to *false*. Therefore, A-events can be regarded as abstractions of the program states. Moreover, technically speaking, A-events are in a bijective map to P_A. For an $MT\ \mu$, the set of states $\theta_{MT}(\mu)$ is often called the set of *possible continuations* of μ under θ.

Example 1. If $\mu = [a \vee \neg b\colon s_1, \neg a \wedge b\colon s_2, c\colon s_3]$, and $\theta(a)$=*true*, $\theta(b)$=*false*, and $\theta(c)$=*true*, then the set of possible continuations of μ under θ, $\theta_{MT}(\mu)$, is $\{s_1, s_3\}$.

Definition 6. *A **(nondeterministic) binary transition tree (BTT)** over A and S is inductively defined as either a set in 2^S or a structure of the form $a\ ?\ \beta_1\colon \beta_2$, for some atom a and for some binary transition trees β_1 and β_2. Let $BTT(A,S)$ denote the set of BTTs over the set of states S and atoms A.*

Definition 7. *Given an event θ, we define its **binary transition tree extension** as the map $\theta_{BTT}\colon BTT(A,S) \to 2^S$, where:*
$\theta_{BTT}(Q) = Q$ *for any set of states* $Q \subseteq S$,
$\theta_{BTT}(a\ ?\ \beta_1\ :\ \beta_2) = \theta_{BTT}(\beta_1)$ *if* $\theta(a) =$ *true, and*
$\theta_{BTT}(a\ ?\ \beta_1\ :\ \beta_2) = \theta_{BTT}(\beta_2)$ *if* $\theta(a) =$ *false.*

Definition 8. *A BTT β **implements** an MT μ, written $\beta \models \mu$, iff for any event θ, it is the case that $\theta_{BTT}(\beta) = \theta_{MT}(\mu)$.*

Example 2. The BTT $b?\ (a?\ (c?\ s_1\ s_3\colon s_1)\ :\ (c?\ s_2\ :\ \emptyset))\ :\ (c?\ s_1\ s_3\colon s_3)$ implements the multi-transition shown in Example 1.

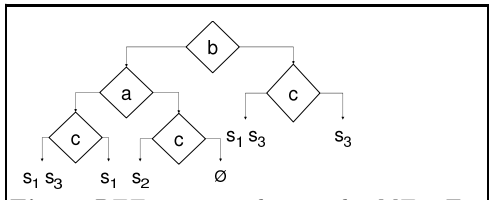

Fig. 2. BTT corresponding to the MT in Example 1

Fig. 2 represents this BTT graphically. The right branch of the node labeled with **b** corresponds to the BTT expression $(c\ ?\ s_1 s_3 : s_3)$, and similarly for the left branch and every other node. Atomic predicates can be any host programming language boolean expressions. For example, one may be interested if a variable x is positive or if a vector $v[1...100]$ is sorted. Some atomic predicates typically are more expensive to evaluate than others. Since our purpose is to generate *efficient monitors*, we need to take the evaluation costs of atomic predicates into consideration. Moreover, some predicates can hold with higher probability than others; for example, some predicates may be simple "sanity checks", such as checking whether the output of a sorting procedure is indeed sorted. We next assume that atomic predicates are given evaluation costs and probabilities to hold. These may be estimated *apriori*, either statically or dynamically.

Definition 9. *If $\varsigma\colon A \to \mathcal{R}^+$ and $\pi\colon A \to [0,1]$ are cost and probability functions for events in A, respectively, then let $\gamma_{\varsigma,\pi}\colon BTT(A,S) \to \mathcal{R}^+$ defined as:*
$\gamma_{\varsigma,\pi}(Q) = 0$ *for any* $Q \subseteq S$, *and*
$\gamma_{\varsigma,\pi}(a\ ?\ \beta_1\ :\ \beta_2) = \varsigma(a) + \pi(a) * \gamma_{\varsigma,\pi}(\beta_1) + (1 - \pi(a)) * \gamma_{\varsigma,\pi}(\beta_2)$,
be the **expected (evaluation) cost** function on BTTs in $BTT(A,S)$.

Example 3. Given $\varsigma = \{(a,10),(b,5),(c,20)\}$ and $\pi = \{(a,0.2),(b,0.5),(c,0.5)\}$, the expected evaluation cost of the BTT defined in Example 2 is 30.

With the terminology and motivations above, the following problem develops as an interesting and important problem in monitor synthesis:

Problem: Optimal $BTT(A,S)$.
Input: A multi-transition $\mu = [p_1 : s_1, p_2 : s_2, ..., p_n : s_n]$ with associated cost $\varsigma\ : A \to \mathcal{R}^+$ and probability $\pi\ :\ A \to [0,1]$.
Output: A minimal cost $BTT\ \beta$ with $\beta \models \mu$.

Binary decision trees (BDTs) and diagrams (BDDs) have been studied as models and data-structures for several problems in artificial intelligence [24] and program verification [7]. Appendix B discusses BDTs and how they relate to BTTs. Moret [24] shows that a simpler version of this problem, using BDTs, is NP-hard.

In spite of this result, in general the number of atoms in formulae is relatively small, so it is not impractical to exhaustively search for the optimal *BTT*. We next informally describe a backtracking algorithm that we are currently using in our implementation to compute the minimal cost *BTT* by exhaustive search. Start with the sequence of all atoms in A. Pick one atom, say a, and make two recursive calls to this procedure, each assuming one boolean assignment to a. In each call, pass the remaining sequence of atoms to test, and simplify the set of propositions in the multi-transition according to the value of a. The product of the *BTTs* is taken when the recursive calls return in order to compute all *BTTs* starting with a. This procedure repeats until no atom is left in the sequence. We select the minimal cost *BTT* amongst all computed.

4 Binary Transition Tree Finite State Machines

We next define an automata-like structure, formalizing the desired concept of an *effective runtime monitor*. The transitions of each state are all-together encoded by a *BTT*, in practice the statistically optimal one, in order for the monitor to efficiently transit as events take place in the monitored program. Violations occur when one cannot further transit to any state for any event. A special state, called *neverViolate*, will denote a configuration in which one can no longer detect a violation, so one can stop the monitoring session if this state is reached.

Definition 10. *A **binary transition tree finite state machine (BTT-FSM)** is a tuple $\langle A, S, btt, S_0\rangle$, where A is a set of atoms, S is a set of states potentially including a special state called "neverViolate", btt is a map associating a BTT in BTT(A,S) to each state in S where $btt(neverViolate)=\{neverViolate\}$ when $neverViolate \in S$, and $S_0 \subseteq S$ is a subset of initial states.*

Definition 11. *Let $\langle A, S, btt, S_0\rangle$ be a BTT-FSM. For an event θ and $Q, Q' \subseteq S$, we write $Q \xrightarrow{\theta} Q'$ and call it a **transition between sets of states**, whenever $Q' = \bigcup_{s\in Q} \theta_{BTT}(btt(s))$. A **trace of events** $\theta_1\theta_2...\theta_j$ generates a **sequence of transitions** $Q_0 \xrightarrow{\theta_1} Q_1 \xrightarrow{\theta_2} ... \xrightarrow{\theta_j} Q_j$ in the BTT-FSM, where $Q_0 = S_0$ and $Q_i \xrightarrow{\theta_{i+1}} Q_{i+1}$, for all $0 \leq i < j$. The trace is **rejecting** iff $Q_j = \{\}$.*

Note that no finite extension of a trace $\theta_1\theta_2...\theta_j$ will be ***rejected*** if $neverViolate \in Q_j$. The state *neverViolate* denotes a configuration in which violations can no longer be detected for any finite trace extension. This means that the set Q_k will not be empty, for any $k > j$, when $neverViolate \in Q_j$. Therefore, the monitoring session can stop at event j if $neverViolate \in Q_j$, because we are only interested in violations of requirements.

5 Generating a *BTT-FSM* from a Büchi automaton

Not any property can be monitored. For example, in order to check a liveness property one needs to ensure that certain propositions hold infinitely often, which cannot be verified at runtime. This section describes how to transform a Büchi automaton into an efficient *BTT-FSM* that rejects precisely the minimal bad prefixes of the denoted ω-language.

Definition 12. *A **monitor FSM (MFSM)** is a tuple $\langle \Sigma, S, \delta, S_0\rangle$, where $\Sigma = P_A$ is an alphabet, S is a set of states potentially including a special state "neverViolate", $\delta\colon S \times \Sigma \to 2^S$ is a transition function with $\delta(neverViolate, true) = \{neverViolate\}$ when $neverViolate \in S$, and $S_0 \subseteq S$ are initial states.*

Note that we take Σ to be P_A, the set of propositions over atoms in A. Like *BTT-FSMs*, *MFSMs* may also have a special *neverViolate* state.

Definition 13. *Let $Q_0 \xrightarrow{\theta_1} Q_1 \xrightarrow{\theta_2} ... \xrightarrow{\theta_j} Q_j$ be a sequence of transitions in the MFSM $\langle \Sigma, S, \delta, S_0\rangle$, generated from $t=\theta_1\theta_2...\theta_j$, where $Q_0=S_0$ and $Q_{i+1}=\bigcup_{s\in Q_i}\{\delta(s,\sigma) \mid \theta_{i+1} \models \sigma\}$, for all $0 \leq i < j$. We say that the MFSM **rejects** t iff $Q_j = \{\}$.*

No finite extension of t will be ***rejected*** if $neverViolate \in Q_j$.

From Büchi to *MFSM*. We next describe two simplification procedures on a Büchi automaton that are sound w.r.t. monitoring, followed by the construction of an *MFSM*. The first procedure identifies segments

of the automaton which cannot lead to acceptance and can therefore be safely removed. As we will show shortly, this step is necessary in order to guarantee the soundness of the monitoring procedure. The second simplification identifies states with the property that if they are reached then the corresponding requirement cannot be violated by any *finite* extension of the trace, so monitoring is ineffective from there on. Note that reaching such a state does not necessarily mean that a good prefix has been recognized, but only that the property is *not monitorable* from there on.

Definition 14. *Let $\langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ be a Büchi automaton, C a connected component of its associated graph, and nodes(C) the states associated to C. We say that C is **isolated** iff for any $s \in nodes(C)$ and $\sigma \in \Sigma$, it is the case that $\delta(s, \sigma) \subseteq nodes(C)$. We say that C is **total** iff for any $s \in nodes(C)$ and event θ, there are transitions σ such that $\theta \models \sigma$ and $\delta(s, \sigma) \cap nodes(C) \neq \emptyset$.*

Therefore, there is no way to escape from an isolated connected component, and regardless of the upcoming event, it is always possible to transit from any node of a total connected component to another node in that component.

Removing Bad States. The next procedure removes states of the Büchi automaton which cannot be part of any accepting run (see Definition 2). Note that any state appearing in such an accepting run must eventually *reach* an accepting state. This procedure is fundamentally inspired by strongly-connected-component-analysis [19, 30], used to check emptiness of the language denoted by a Büchi automaton. Given a Büchi automaton $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$, let $U \subseteq S$ be the largest set of states such that the language of $\langle \Sigma, S, \delta, U, \mathcal{F} \rangle$ is empty. The states in U are unnecessary in $\mathcal{A}$, because they cannot change its language. Fortunately, U can be calculated effectively as the set of states that *cannot reach* any cycle in the graph associated to $\mathcal{A}$ which contains at least one accepting state in $\mathcal{F}$. Fig. 3 shows an algorithm to do this.

```
INPUT : A Büchi automaton A
OUTPUT : A smaller Büchi automaton A' such that L(A') = L(A).
REMOVE_BAD_STATES :
    for each maximal connected component C of A
      if (C is isolated and nodes(C) ∩ F=∅) then mark all states in C "bad"
    DFS_MARK_BAD ; REMOVE_BAD
```

Fig. 3. Removing bad states

The loop identifies maximal isolated connected components which do not contain any accepting states. The nodes in these components are marked as "bad". The procedure `DFS_MARK_BAD` performs a depth-first-search in the graph and marks nodes as "bad" when all outgoing edges lead to a "bad" node. Finally, the procedure `REMOVE_BAD` removes all the bad states. The runtime complexity of this algorithm is dominated by the computation of maximal connected components. In our implementation, we used Tarjan's $O(V + E)$ double DFS [7]. The proof of correctness is simple and it appears in Appendix A. The Büchi automaton $\mathcal{A}'$ produced by the algorithm in Fig. 3 has the property that there is some proper path from any of its states to some accepting state. One can readily generate an *MFSM* from a Büchi automaton $\mathcal{A}$ by first applying the procedure `REMOVE_BAD_STATES` in Fig. 3, and then ignoring the acceptance conditions.

Theorem 1. *The MFSM generated from a Büchi automaton $\mathcal{A}$ as above rejects precisely the minimal bad prefixes of $\mathcal{L}(\mathcal{A})$.*

Proof. Let $\mathcal{A}=\langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ be the original Büchi automaton, let $\mathcal{A}'= \langle \Sigma, S', \delta', S'_0, \mathcal{F} \rangle$ be the Büchi automaton obtained from $\mathcal{A}$ by applying the algorithm in Fig. 3, and let $\langle \Sigma, S', \delta', S'_0 \rangle$ be the corresponding *MFSM* of $\mathcal{A}'$. For any finite trace $t = \theta_1 ... \theta_j$, let us consider its corresponding sequence of transitions in the *MFSM* $Q_0 \xrightarrow{\theta_1} ... \xrightarrow{\theta_j} Q_j$, where Q_0 is S'_0. Note that the trace t can also be regarded as a sequence of letters in the alphabet Σ of $\mathcal{A}$, because we assumed Σ is P_A and because there is a bijection between propositions in P_A and A-events. All we need to show is that t is a bad prefix of $\mathcal{A}'$ if and only if $Q_j = \emptyset$. Recall that $\mathcal{A}'$ has the property that there is some non-empty path from any of its states to some accepting state. Thus, one can build an infinite path in $\mathcal{A}'$ starting with any of its nodes, with the property that some accepting state occurs infinitely often. In other words, Q_j is not empty iff the finite trace t is the prefix of some infinite trace in $\mathcal{L}(\mathcal{A}')$. This is equivalent to saying that Q_j is empty iff the trace t is a bad prefix in $\mathcal{A}'$. Since Q_j

empty implies $Q_{j'}$ empty for any $j>j'$, it follows that the *MFSM* rejects precisely the minimal bad prefixes of $\mathcal{A}$. □

Theorem 1 says that the *MFSM* obtained from a Büchi automaton as above can be used as a monitor for the corresponding ω-language. Indeed, one only needs to maintain a current set of states Q, initially S'_0 , and transform it accordingly as new events θ are generated by the observed program: if $Q \xrightarrow{\theta} Q'$ then set Q to Q'; if Q ever becomes empty then report violation. Theorem 1 tells us that a violation will be reported as soon as a bad prefix is encountered.

Collapsing Never-Violate States. Reducing runtime overhead is crucial in runtime verification. There are many situations when the monitoring process can be safely stopped, because the observed finite trace cannot be finitely extended to any bad prefix. The following procedure identifies states in a Büchi automaton which cannot lead to the violation of any *finite* computation. For instance, the Büchi automaton in Fig. 4 can only reject infinite words in which the state s_2 occurs finitely many times; moreover, at least one transition is possible at any moment. Therefore, the associated *MFSM* will never report a violation, even though there are infinite words that are not accepted. We call such an automaton *non-monitorable*. This example makes it clear that if a state like s_1 is ever reached by the monitor, it does *not* mean that we found a good prefix, but that we could *stop looking* for bad prefixes.

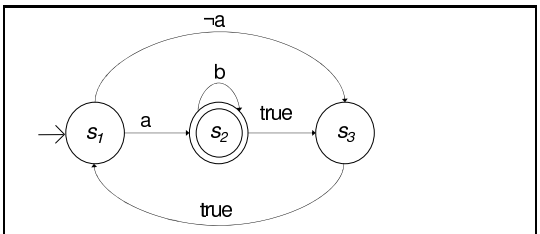

Fig. 4. Non-monitorable automaton

Let $\mathcal{A}=\langle \Sigma, S, \delta, S_0, \mathcal{F}\rangle$ be a Büchi automaton *simplified with* `REMOVE_BAD_STATES`. The procedure in Fig. 5 finds states which, if reached by a monitor, then the monitor can no longer detect violations regardless of what events will be observed in the future. The procedure first identifies the total connected components. According to the definition of totality, once a monitor reaches a state of a total connected component, the monitor will have the possibility to always transit within that connected component, thus never getting a chance to report violation. All states of a total component can therefore be marked as "never violate". Other states can also be marked as such if, for any events, it is possible to transit from them to states already marked "never violate"; that is the reason for the disjunction in the second conditional. The procedure finds such nodes in a depth-first-search. Finally, `COLLAPSE-NEVER_VIOLATE` collapses all components marked "never violate", if any, to a distinguished node, *neverViolate*, having just a *true* transition to itself. If any collapsed node was in the initial set of states, then the entire automaton is collapsed to *neverViolate*. The procedure `GENERATE_MFSM` produces an *MFSM* by ignoring accepting conditions.

```
INPUT : A Büchi automaton A, cost function ς, and probability function π.
OUTPUT : An effective BTT-FSM monitor rejecting the bad prefixes of L(A).
COLLAPSE_NEVER_VIOLATE :
    for each maximal connected component C of A
      if ( C is total ) then mark all states in C as "never violate"
    for each s in depth-first-search visit
      if ( ⋁{σ | δ(s,σ) contains some state marked "never violate"} )
      then mark s as "never violate"
    COLLAPSE-NEVER_VIOLATE ; GENERATE_MFSM ; GENERATE_BTT-FSM
```

Fig. 5. Collapsing non-monitorable states

Taking as input this *MFSM*, say $\langle \Sigma, S', \delta', S'_0\rangle$, cost function ς, and probability function π, `GENERATE_BTT-FSM` constructs a *BTT-FSM* $\langle A, S', btt, S'_0\rangle$, where A corresponds to the set of atoms from which the alphabet Σ is built, and the map *btt*, here represented by a set of pairs, is defined as follows:

$$btt = \{(neverViolate, \{neverViolate\}) \mid neverViolate \in S'\} \cup$$
$$\{(s, \beta_s) \mid s \in S'\text{-}\{neverViolate\} \wedge \beta_s \models \mu_s\}, \text{ where}$$
$$\beta_s \text{ optimally implements } \mu_s \text{ w.r.t. } \varsigma \text{ and } \pi, \text{ with } \mu_s = \oplus(\bigcup\{[\sigma : s'] \mid s' \in \delta'(s,\sigma)\})$$

The symbol $\oplus$ denotes concatenation on a set of multi-transitions. Optimal *BTTs* β_s are generated like in Section 3. Proof of correctness appears in Appendix A.

6 Monitor Generation and MOP

We have shown that one can generate from a Büchi automaton a *BTT-FSM* recognizing precisely its bad prefixes. However, it is still necessary to integrate the *BTT-FSM* monitor within the program to be observed.

Monitoring-oriented programming (MOP) [6] aims at merging specification and implementation through generation of runtime monitors from specifications and integration of those within implementation. In MOP, the task of generating monitors is divided into defining a logic engine and a language shell. The logic engine is concerned with the translation of specifications given as logical formulae into monitoring (pseudo-)code. The shell is responsible for the integration of the monitor within the application.

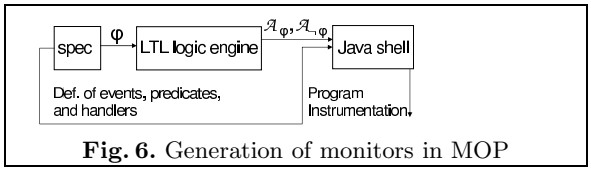

Fig. 6. Generation of monitors in MOP

Fig. 6 captures the essence of the synthesis process of LTL monitors in MOP using the technique described in this paper. The user defines specifications either as annotations in the code or in a separate file. The specification contains definitions of events and state predicates, as well as LTL formulae expressing trace requirements. These formulae treat events and predicates as atomic propositions. Handlers are defined to track violation or validation of requirements. For instance, assume the events a and b denote the login and the logoff of the same user, respectively. Then the formula $\Box(a \rightarrow \circ(\neg a \,\mathcal{U}\, b))$ states that the user cannot be logged in more than once. A violation handler could be declared to track the user who logged in twice. The logic engine is responsible for the translation of the formulae φ and $\neg\varphi$ into two *BTT-FSM* monitors. One detects violation and the other validation of φ. Note that if the user is just interested in validation (no violation handler), then only the automaton for negation is generated. Finally, the language shell reads the definition of events and instruments the code so that the monitor will receive the expected notifications.

We used LTL2BA [25] to generate standard Büchi automata from LTL formulae. The described procedures are implemented in Java. This software and a WWW demo are available from the MOP website [1].

6.1 Evaluation

Table 1 shows *BTT-FSM* monitors for some LTL formulae. The *BTT* definition corresponding to a state follows the arrow ($\rightsquigarrow$). Initial states appear in brackets. For producing this table, we used the same cost and probabilities for all events and selected the smallest *BTT*. The first formula cannot be validated by monitoring and presents the permanent possibility to be violated; that is why its *BTT-FSM* does not have a *neverViolate* state. The second formula can never be violated since event a followed by event b can always occur in the future, so its *BTT-FSM* consists of just one state *neverViolate*. The last formula shows that our procedure does not aim at distinguishing validating from non-violating prefixes.

Temporal Formula	*BTT-FSM*
$\Box(a \rightarrow b\,\mathcal{U}\,c)$	$[s_0] \rightsquigarrow c\ ?\ (b\ ?\ s_0 s_1 :\ s_0) :\ (a\ ?\ (b\ ?\ s_1 :\ \emptyset) :\ (b\ ?\ s_0 s_1 :\ s_0))$ $s_1 \rightsquigarrow b\ ?\ (c\ ?\ s_0 s_1 :\ s_1) : (c\ ?\ s_0 :\ \emptyset)$
$\Box(a \rightarrow \diamond b)$	$[neverViolate] \rightsquigarrow \{neverViolate\}$
$a\,\mathcal{U}\,b\,\mathcal{U}\,c$	$[s_0] \rightsquigarrow c\ ?\ neverViolate\ :\ (a\ ?\ (b\ ?\ s_0 s_1 :\ s_0)\ :\ (b\ ?\ s_1 :\ \emptyset))$ $s_1 \rightsquigarrow c\ ?\ neverViolate\ :\ (b\ ?\ s_1 :\ \emptyset)$ $neverViolate \rightsquigarrow \{neverViolate\}$

Table 1. *BTT-FSMs* generated from temporal formulae

Table 2 shows that our technique can not only identify non-monitorable formulae, but also reduce the cost of monitoring by collapsing large parts of the Büchi automaton. We use the symbols ♡, ♣, and ♠ to denote, respectively, the effectiveness of `REMOVE_BAD_STATES`, the first, and the second loop of `COLLAPSE_NEVER_VIOLATE`. The first group contains non-monitorable formulae. The next contains formulae where monitor size could not be reduced by our procedures. The third group shows formulae where our simplifications could significantly reduce the monitor size. The last group shows examples of "accidentally" safe and "pathologically" safe formulae from [21]. A formula φ is accidentally safe iff not all bad prefixes are "informative" [21] (i.e., can serve as a witness for violation) but all computations that violate φ have an informative bad prefix. A formula φ is pathologically safe if there is a computation that violates φ and has no informative bad prefix. Since we detect *all* minimal bad prefixes, informativeness does not play any role in our approach. Both formulae are monitorable. For the last formula, in particular, a minimal bad prefix will be detected as soon as the monitor observes a $\neg a$, having previously observed a $\neg b$. One can generate and visualize the *BTT-FSMs*of all these formulae, and many others, online at [1].

Temporal Formula	# states	# transitions	symplif.
$\diamond a$	2 , 1	3 , 1	♣
$a \ \mathcal{U} \circ(\diamond b)$	3 , 1	5 , 1	♣♠
$\Box(a \wedge b \rightarrow \diamond c)$	2 , 1	4 , 1	♣
$a \,\mathcal{U}\, (b \,\mathcal{U}\, (c \,\mathcal{U}\, (\diamond d)))$	2 , 1	3 , 1	♣
$a \,\mathcal{U}\, (b \,\mathcal{U}\, (c \,\mathcal{U}\, \Box(d \rightarrow \diamond e)))$	5 , 1	15 , 1	♣♠
$\neg\, a \,\mathcal{U}\, (b \,\mathcal{U}\, (c \,\mathcal{U}\, \Box(d \rightarrow \diamond e)))$	12 , 1	51 , 1	♣
$\neg \diamond a$	1 , 1	1 , 1	
$\Box(a \rightarrow b \,\mathcal{U}\, c)$	2 , 2	4 , 4	
$a \,\mathcal{U}\, (b \,\mathcal{U}\, (c \,\mathcal{U}\, d))$	4 , 4	10 , 10	
$a \wedge \circ(\diamond b) \wedge \diamond(\Box(e))$	5 , 4	11 , 6	♣
$a \wedge \circ(\diamond b) \wedge \circ(\diamond c) \wedge \diamond(\Box(e))$	9 , 6	29 , 12	♣
$a \wedge \circ(\diamond b) \wedge \circ(\diamond c) \wedge \circ(\diamond d) \wedge \diamond(\Box(e))$	17 , 10	83 , 30	♣
$a \wedge \circ(\neg(\Box(b \rightarrow c \,\mathcal{U}\, d))) \wedge \diamond(\Box(e))$	7 , 5	20 , 10	♣
$\Box(a \vee \circ(\Box(c))) \wedge \Box(b \vee \circ(\Box(\neg c)))$	3 , 3	5 , 5	
$(\Box(a \vee \diamond(\Box(c))) \wedge \Box(b \vee \diamond(\Box(\neg c)))) \vee \Box(a) \vee \Box(b)$	12 , 6	43 , 22	♡♣

Table 2. Number of states and transitions before and after monitoring simplifications

7 Conclusions

Not all properties a Büchi automaton can express are monitorable. This paper describes transformations that can be applied to extract the monitorable components of Büchi automata, reducing their size and the cost of runtime verification. The resulting automata are called *monitor finite state machines* (*MFSMs*). The presented algorithms have polynomial running time in the size of the original Büchi automata and have already been implemented. Another contribution of this paper is the definition and use of *binary transition trees* (*BTTs*) and corresponding finite state machines (*BTT-FSMs*), as well as a translation from *MFSMs* to *BTT-FSMs*. These special-purpose state machines encode optimal evaluation paths of boolean propositions in transitions.

We used LTL2BA [25] to generate Büchi automata from LTL, and JAVA to implement the presented algorithms. Our algorithms, as well as a graphical HTML interface, are available at [1]. This work is motivated by, and is part of, a larger project aiming at promoting monitoring as a foundational principle in software development, called *monitoring-oriented programming* (MOP). In MOP, the user specifies formulae, atoms, cost and probabilities associated to atoms, as well as violation and validation handlers. Then all these are used to automatically generate monitors and integrate them within the application.

This work is concerned with monitoring *violations* of requirements. In the particular case of LTL, *validations* of formulae can also be checked using the same technique by monitoring the negation of the input formula. Further work includes implementing the algorithm defined in [11] for generating Büchi automata of size $2^{O(|\varphi|)}$ from PTL, combining multiple formulae in a single automaton as showed by Ezick [9] so as to reduce redundancy of proposition evaluations, and applying further (standard) NFA simplifications to *MFSM*.

References

1. MOP website, LTL plugin. http://fsl.cs.uiuc.edu/mop/logic-plugins/ltl.
2. C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Păsăreanu, G. Roşu, W. Visser, and R. Washington. Automated Testing using Symbolic Execution and Temporal Monitoring. *Theoretical Computer Sci.*, to appear, 2005.
3. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of VMCAI'04*, volume 2937 of *LNCS*, pages 44–57, 2004.
4. J. R. Büchi. *On a Decision Method in Restricted Second Order Arithmetic*. Logic, Methodology and Philosophy of Sciences. Stanford University Press, 1962.
5. F. Chen, M. d'Amorim, and G. Roşu. A Formal Monitoring-Based Framework for Software Development and Analysis. In *Proceedings of ICFEM'04*, volume 3308 of *LNCS*, pages 357–372, 2004.
6. F. Chen and G. Roşu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In *Proceedings of RV'03*, volume 89 of *ENTCS*, pages 106–125, 2003.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
8. K. Etessami and G. Holzmann. Optimizing Büchi Automata. In *Proc. of Int. Conf. on Concurrency Theory*, volume 1877 of *LNCS*, pages 153–167. Springer, 2000.

9. J. Ezick. An Optimizing Compiler for Batches of Temporal Logic Formulas. In *Proceedings of ISSTA'04*, pages 183–194. ACM Press, 2004.
10. M. Garey. Optimal Binary Identification Procedures. *SIAM Journal on Applied Mathematics*, 23(2):173–186, 1972.
11. P. Gastin and D. Oddoux. LTL with Past and Two-Way Very-Weak Alternating Automata. In *Proceedings of MFCS'03*, number 2747 in LNCS, pages 439–448. Springer, 2003.
12. M. Geilen. On the Construction of Monitors for Temporal Logic Properties. In *Proceedings of RV'01*, volume 55 of *ENTCS*. Elsevier Science, 2001.
13. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the 15th IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18. Chapman & Hall, Ltd., 1996.
14. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of RV'01*, volume 55 of *ENTCS*. Elsevier Science, 2001.
15. K. Havelund and G. Roşu. *Workshops on Runtime Verification (RV'01, RV'02, RV'04)*, volume 55, 70(4), to appear of *ENTCS*. Elsevier, 2001, 2002, 2004.
16. K. Havelund and G. Roşu. Synthesizing Monitors for Safety Properties. In *Proceedings of TACAS'02*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002.
17. G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
18. L. Hyafil and R. L. Rivest. Computing Optimal Binary Decision Trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.
19. Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A Decision Algorithm for Full Propositional Temporal Logic. In *Proceedings of CAV'93*, volume 697 of *LNCS*, pages 97–109. Springer, 1993.
20. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of RV'01*, volume 55 of *ENTCS*. Elsevier Sci., 2001.
21. O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Proceedings of CAV '99*, volume 1633 of *LNCS*, pages 172–183. Springer, 1999.
22. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, New York, 1995.
23. N. Markey. Temporal Logic with Past is Exponentially more Succinct. *EATCS Bulletin*, 79:122–128, 2003.
24. B. Moret. Decision Trees and Diagrams. *ACM Comp. Surv.*, 14(4):593–623, 1982.
25. D. Oddoux and P. Gastin. LTL2BA. http://www.liafa.jussieu.fr/~oddoux/ltl2ba/.
26. G. Roşu and K. Havelund. Rewriting-Based Techniques for Runtime Verification. *Journal of Automated Software Engineering*, 2005. to appear.
27. K. Sen, G. Roşu, and G. Agha. Online Efficient Predictive Safety Analysis of Multithreaded Programs. In *Proceedings of TACAS'04*, volume 2988 of *LNCS*, pages 123–138. Springer, 2002.
28. A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the ACM*, 32(3):733–749, 1985.
29. O. Sokolsky and M. Viswanathan. *Workshop on Runtime Verification (RV'03)*, volume 89 of *ENTCS*. Elsevier, 2003.
30. P. Wolper. Constructing Automata from Temporal Logic Formulas: a Tutorial. volume 2090 of *LNCS*, pages 261–277. Springer, 2002.

Appendix A. Proof of Correctness

(*Correctness of* `REMOVE_BAD_STATES`) Let $\mathcal{A} = \langle \Sigma, S, \delta, S_0, \mathcal{F} \rangle$ be the input and $\mathcal{A}' = \langle \Sigma, S', \delta', S'_0, \mathcal{F} \rangle$ the output Büchi automata of the procedure. We want to show that their denoted languages are the same. Let ρ be an accepting run of $\mathcal{A}$. Then ρ can be fragmented in $\rho'\rho''$ where ρ' is the prefix whose states appear only finitely many times in ρ. Each state in ρ'' is therefore reachable from any other and therefore must be in a connected component where some states are in $\mathcal{F}$. The converse is also true, i.e., any connected component reachable from the set of initial states having at least one state in $\mathcal{F}$ generates accepting runs (from [7] pp. 129). It is then immediate to notice that no accepting run ρ contains states that can never reach a state in $\mathcal{F}$. It turns out that any state belonging to an isolated component with no accepting states can be trivially removed as well as any state that can *only* make transitions to others which never reach an accepting state. This reachability analysis is performed in a depth-first-search as usual. Since only states that will never appear in accepting runs of $\mathcal{A}$ are removed, it follows that $\mathcal{L}(\mathcal{A})$=$\mathcal{L}(\mathcal{A}')$. □

(*Correctness of* `COLLAPSE_NEVER_VIOLATE`) First, we need to show that the *MFSM* produced by `GENERATE_MFSM` still detects bad prefixes of $\mathcal{L}(\mathcal{A})$, where $\mathcal{A}$ is the input Büchi automaton having the property that all states can reach accepting states. From the observation thal all states that have been collapsed to *neverViolate* can reach an accepting state, the proof of the first part is similar to that of Theorem 1 and is omitted. So, the *MFSM* is still a monitor for the bad prefixes of $\mathcal{L}(\mathcal{A})$. In other words, this simplification is sound w.r.t. monitoring minimal bad prefixes. In addition to this, we need to show that the state *neverViolate* is reached *only if* violations of the requirement can no longer occur. We prove this second correctness criteria by a case analysis on the shape of the *MFSM* associated to the input Büchi. Each case corresponds to a loop in `COLLAPSE_NEVER_VIOLATE`.

(case 1) Let $Q_0 \xrightarrow{\theta_1} Q_1 \xrightarrow{\theta_2} ... \xrightarrow{\theta_j} Q_j$ be a sequence of transitions on the trace $\theta_1\theta_2...\theta_j$ produced by the *MFSM* corresponding to the input Büchi automaton $\mathcal{A}$. Recall that we assume `REMOVE_BAD_STATES` is called before `COLLAPSE_NEVER_VIOLATE`. That is, all states in the input automaton reach some state in $\mathcal{F}$. If some node in a total connected component C of the graph associated to $\mathcal{A}$ belongs to Q_j then it is not possible for any finite trace with prefix $\theta_1\theta_2...\theta_j$ to be rejected by the corresponding *MFSM* since it is always possible to make a transition between nodes in C from the definition of totality. Those states in C can be marked as "never violate" meaning that they denote the special *neverViolate* state.

(case 2) Let $Q_0 \xrightarrow{\theta_1} Q_1 \xrightarrow{\theta_2} ... \xrightarrow{\theta_j} Q_j$ be a sequence of transitions produced on the trace $\theta_1\theta_2...\theta_j$ by the *MFSM* defined as before. If any state belongs to Q_j with the property that for further events θ_{j+1} there exists a transition to a state marked "never violate", then there must exist some state marked "never violate" in Q_{j+1}. Such states can be found by checking the validity of the disjunction of propositions labeling the edges of transitions to states marked "never violate". Since the monitor will definitely reach a state marked "never violate" in the trace $\theta_1\theta_2...\theta_j\theta_{j+1}$, it is therefore safe to reach "never violate" in Q_j as well, since violations are impossible from event j on.

The states marked with the "never violate" label can therefore be collapsed, since a violation cannot occur if any of these states is reached. The *BTT-FSM* is generated from the *MFSM* according to the defined construction. □

Appendix B. Complexity Results

We use the definition of Binary Decision Trees (BDTs) due to Garey [10]. As we will show next, Garey's BDTs serve as a procedure to identify one among a set of possible objects characterizing some aspect of interest. Hyafil and Rivest [18] proved the Optimal BDT problem NP-complete. Moret [24] shows that these BDTs are a restricted form of decision trees similar to binary transition trees as defined here.

Let $X = \{x_1, ..., x_n\}$ be a finite set of objects and $\mathcal{T} = \{T_1, ..., T_t\}$ a finite set of tests. For each test T_i, $1 \leq i \leq t$, and object $x_j, 1 \leq j \leq n$, we either have $T_i(x_j) = \mathit{true}$ or $T_i(x_j) = \mathit{false}$. A binary decision tree associates tests in the root and all other internal nodes, and associates objects in X to terminal nodes. The *optimal decision tree* problem is to determine whether there exists a decision tree with cost less than or equal to w which completely identifies each element in X, given $\mathcal{T}$ and X. The cost of this tree is defined as $\Sigma_{x_i \in X}\ p(x_i)$, where $p(x_i)$ is the length of the path from the root to the terminal identifying x_i.

One might think of the objects in X as possible answers to a question. The tests in T serve to prune the data set of answers. A decision tree defines possible sequences of questions to ask in order to give a unique answer among those in X. The table of Fig. 7 appears in [10] and denotes the set of tests available to use in some binary decision tree.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
T_1	T	F	T	F	T	F	F
T_2	F	T	F	F	F	T	T
T_3	T	F	F	F	F	F	F
T_4	F	F	F	F	T	F	F
T_5	F	F	F	F	F	T	F
T_6	F	F	F	F	F	F	T

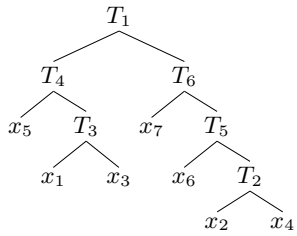

Fig. 7. Table of tests and a possible BDT for this table

The definition above [10] and the NP-completeness proof for the Optimal BDT problem [18] assume the table provided as input is unambiguous and completely identifies the objects in X. That is, any object in X can be distinguished by applying some sequence of tests. The decision tree in Fig. 7 appears in [10] as an identification procedure for the tests and objects defined in the adjacent table. The subtree to the left of a node denotes objects whose answers to the test labeling that node are *true* and the result of all other tests applied in the path to the root are consistent. The right subtree is similar. This is similar to binary transition trees. However, it is worth noting that in order to identify x_j one does not need to test all T_i having $T_i(x_j) = true$. Observe this in particular for x_6 and x_7. It means that decision trees prune the data set in each test they make. A decision can be made whenever it is possible to identify an object. For instance, no object but x_7 assigns *false* to T_1 and *true* to T_6. So this is a sufficient condition to identify x_7. Moret showed that the construction of optimal binary decision trees, similar to $BTTs$ (where tests take the form of boolean proposition over a set of atoms) is an NP-hard [24] problem. That implies that our BTT problem is also NP-hard.