Runtime Verification: From Propositional to First-Order Temporal Logic

Klaus Havelund[1] and Doron Peled[2](✉)

[1] Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA
[2] Department of Computer Science, Bar Ilan University, Ramat Gan, Israel
doron.peled@gmail.com

Abstract. Runtime Verification is a branch of formal methods concerned with analysis of execution traces for the purpose of determining the state or general quality of the executing system. The field covers numerous approaches, one of which is specification-based runtime verification, where execution traces are checked against formal specifications. The paper presents syntax, semantics, and monitoring algorithms for respectively propositional and first-order temporal logics. In propositional logics the observed events in the execution trace are represented using atomic propositions, while first-order logic allows universal and existential quantification over data occurring as arguments in events. Monitoring of the first-order case is drastically more challenging than the propositional case, and we present a solution for this problem based on BDDs. We furthermore discuss monitorability of temporal properties by dividing them into different classes representing different degrees of monitorability.

1 Introduction

Runtime verification (RV) [2,13] allows monitoring (analysis) of executions of a system, directly, without the need for modeling the system. It has some commonality with other formal methods such as testing, model checking and formal verification, including the use of a specification formalisms[1]. However, it differs a lot in goals, the algorithms used, and the complexity and the coverage it suggests. Model checking performs a comprehensive search on a model of the system under test. Testing generates inputs to drive system executions, trying to provide a good coverage, yet keeping the complexity low, at the price of losing

The research performed by the first author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research performed by the second author was partially funded by Israeli Science Foundation grant 2239/15: "Runtime Measuring and Checking of Cyber Physical Systems".

[1] RV can be understood more broadly to mean: any processing of execution traces for the purpose of evaluating a system state or quality. Some approaches do not involve specifications but rather use pre-programmed algorithms as monitors.

C. Colombo and M. Leucker (Eds.): RV 2018, LNCS 11237, pp. 90–112, 2018.
https://doi.org/10.1007/978-3-030-03769-7_7

exhaustiveness. Formal verification attempts full proof of correctness based on deductive techniques. Runtime verification does not directly concern itself with coverage and the selection of execution paths, but rather focuses on analyzing a single execution trace (or a collection thereof). An execution trace is generated by the observed executing system, typically by instrumenting the system to generate events as important transitions take place. Instrumentation can be manual by inserting logging statements in the code, or it can be automated using instrumentation software, such as aspect-oriented programming frameworks.

Runtime verification can take place on-line, as the system executes, or off-line, by processing log files produced by the system. In the case of on-line processing, runtime verification obtains the information about the execution as it unfolds, oftentimes without seeing the complete sequence; yet it is required to provide a verdict as soon as possible. The critical complexity measure here is the *incremental complexity*, which is performed for each new event reported to the monitor. The calculation needs to be fast enough to keep in pace with the executing system.

Following in part [14,23], we present algorithms for the runtime verification of linear temporal logic properties, which is the most common specification formalism used for both runtime verification and model checking. We start with the propositional case, where an execution trace is checked against a future or past time propositional LTL formula. For an online algorithm, which observes the execution trace event by event, a verdict is not guaranteed in any finite time. Runtime monitorability identifies what kind of verdicts can be expected when monitoring an execution against a temporal property. Monitoring temporal properties is often restricted to safety properties. There are two main reasons for this restriction: the first is that the algorithm for checking safety is rather efficient, polynomial in the size of the property; the other reason is that for safety properties we are guaranteed to have a finite evidence for a negative verdict (albeit there is not always a bound on when such an evidence can be given).

After presenting the theory of monitoring propositional temporal logic we move on to the more demanding challenge of monitoring properties that depend on data reported to the monitor. This can be handled by a parametrized version of temporal logic (or using parametrized automata), but more generally it calls for using a first-order version of the temporal logic. We will concentrate on first-order safety properties. One of the challenges here is that the data may, in principle, be unbounded and we only learn about the actual values that are monitored as they appear in reported events. Another problem is that, unlike the propositional case, the amount of data that needs to be kept may keep growing during the execution. This calls for a clever representation that allows fast processing of many data elements. We present an algorithm based on BDDs, which is implemented in the tool Deja Vu [12].

The paper is organized as follows. Section 2 introduces propositional linear temporal logic, including future as well as past time operators, its syntax, semantics, and some pragmatics. Section 3 presents a general theory of monitorability of temporal properties, such as those formulated in LTL. Section 4 outlines algorithms for monitoring propositional LTL properties, first future time, and then past time. Section 5 introduces first-order past time LTL, its syntax, semantics, and some pragmatics. Section 6 outlines an algorithm for monitoring first-order past LTL properties. Finally, Sect. 7 concludes the paper.

2 Propositional LTL

The definition of linear temporal logic including future and past time operators is as follows [20]:

$$\varphi ::= \mathit{true} \,|\, p \,|\, (\varphi \wedge \varphi) \,|\, \neg\varphi \,|\, (\varphi\, \mathcal{U}\, \varphi) \,|\, \bigcirc \varphi | (\varphi \mathcal{S} \varphi) \,|\, \ominus \varphi$$

where p is a proposition from a finite set of propositions P, with $\mathcal{U}$ standing for *until*, $\bigcirc$ standing for *next-time*, $\mathcal{S}$ standing for *since*, and $\ominus$ standing for *previous-time*. One can also write $(\varphi \vee \psi)$ instead of $\neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi)$ instead of $(\neg\varphi \vee \psi)$, $\Diamond\varphi$ (*eventually* φ) instead of $(\mathit{true}\, \mathcal{U}\, \varphi)$, $\Box\varphi$ (*always* φ) instead of $\neg\Diamond\neg\varphi$, $\mathbf{P}\, \varphi$ (*past* φ) instead of $(\mathit{true}\ \mathcal{S}\ \varphi)$ and $\mathbf{H}\ \varphi$ (*history* φ) instead of $\neg\mathbf{P}\ \neg\varphi$.

LTL formulas are interpreted over an infinite sequence of events $\xi = e_1.e_2.e_3\ldots$, where $e_i \subseteq P$ for each $i > 0$. These are the propositions that *hold* in that event. LTL's semantics is defined as follows:

- $\xi, i \models \mathit{true}$.
- $\xi, i \models p$ iff $p \in e_i$.
- $\xi, i \models \neg\varphi$ iff not $\xi, i \models \varphi$.
- $\xi, i \models (\varphi \wedge \psi)$ iff $\xi, i \models \varphi$ and $\xi, i \models \psi$.
- $\xi, i \models \bigcirc\varphi$ iff $\xi, i+1 \models \varphi$.
- $\xi, i \models (\varphi\, \mathcal{U}\, \psi)$ iff for some $j \geq i$, $\xi, j \models \psi$, and for all $i \leq k < j$ it holds that $\xi, k \models \varphi$.
- $\xi, i \models \ominus\varphi$ iff $i > 1$ and $\xi, i-1 \models \varphi$.
- $\xi, i \models (\varphi\, \mathcal{S}\, \psi)$ iff $\xi, i \models \psi$ or the following hold[2]: $i > 1$, $\xi, i \models \varphi$ and $\xi, i-1 \models (\varphi\, \mathcal{S}\, \psi)$.

Then $\xi \models \varphi$ when $\xi, 1 \models \varphi$.

This definition of propositional temporal logic contains both future modalities ($\mathcal{U}$, $\Box$, $\Diamond$ and $\bigcirc$) and past modalities ($\mathcal{S}$, $\mathbf{H}$, $\mathbf{P}$ and $\ominus$). However, we do not always need to use all off them:

- Removing the past temporal operators does not affect the expressiveness of the logic [10]. On the other hand, there are examples of properties that are much more compact when expressed using both the past and the present operators.

[2] This definition is equivalent to the traditional definition $\xi, i \models (\varphi\ \mathcal{S}\ \psi)$ iff for some $0 < j \leq i$, $\xi, j \models \psi$, and for all $j < k \leq i$ it holds that $\xi, k \models \varphi$, but is more intuitive for the forthcoming presentation of the RV algorithm.

- Properties of the form $\Box\varphi$, where φ does *not* contain the future operators form an important class. There are several reasons for restricting runtime verification to such properties. These properties correspond to temporal *safety properties* [1,19]: failure can always be detected on a finite prefix [5]. Moreover, expressing safety properties in this form allows an efficient runtime verification algorithm that is only polynomial in the size of the specification [14][3].

3 Monitorability of Propositional Temporal Logic

Online runtime verification of LTL properties inspects finite prefixes of the execution. Assume an observed system S, and assume further that a finite execution of S up to a certain point is captured as an execution trace $\xi = e_1.e_2.\ \ldots\ .e_n$, which is a sequence of observed events, each of type $\mathbb{E}$. Each event e_i captures a snapshot of S's execution state. Then the RV problem can be formulated as constructing a program M with the type $M : \mathbb{E}^* \rightarrow D$, which when applied to the trace ξ, as in $M(\xi)$, returns some data value $d \in D$ in a domain D of interest. Typically M is generated from a formal specification, given as a temporal logic formula or a state machine. Because online RV observes at each time only a finite part of the execution, it can sometimes provide only a partial verdict on the satisfaction and violation of the inspected property [4,22]. This motivates providing three kinds of verdicts as possible values for the domain D:

failed when the current prefix cannot be extended in any way into an execution that satisfies the specification,
satisfied when any possible extension of the current prefix satisfies the specification, and
undecided when the current prefix can be extended to satisfy the specification but also extended to satisfy its negation.

For example, the property $\Box p$ (for some atomic proposition p), which asserts that p always happens, can be refuted by a runtime monitor if p does not hold in some observed event. At this point, no matter which way the execution is extended, the property will not hold, resulting in a *failed* verdict. However, no finite prefix of an execution can establish that $\Box p$ holds. In a similar way, the property $\Diamond p$ cannot be refuted, since p may appear at any time in the future; but once p happens, we know that the property is satisfied, independent on any continuation, and we can issue a *satisfied* verdict. For the property $(\Box p \vee \Diamond q)$ we may not have a verdict at any finite time, in the case where all the observed events satisfy both p and $\neg q$. On the other hand, we may never "lose hope" to have such a verdict, as a later event satisfying q will result in a positive verdict; at this point we can abandon the monitoring, since the property cannot be further violated. On the other hand, for the property $\Box\Diamond p$ we can never provide a verdict in finite

[3] There are examples of safety properties that are much more compact when expressed with the past temporal operators [21], and for symmetrical considerations also vice versa.

time: for whatever happens, p can still appear an infinite number of times, and we cannot guarantee or refute that this property holds when observing any finite prefix of an execution. The problem of monitorability of a temporal property was studied in [5,9,24], basically requiring that at any point of monitoring we still have a possibility to obtain a finite positive or negative verdict.

Safety and liveness temporal properties were defined informally on infinite execution sequences by Lamport [19] as *something bad cannot happen* and *something good will happen*. These informal definitions were later formalized by Alpern and Schneider [1]. Guarantee properties where used in an orthogonal characterization by Manna and Pnueli [20]. Guarantee properties are the dual of safety properties, that is, the negation of a safety property is a guarantee property and vice versa.

These classes of properties can be seen as characterizing finite monitorability of temporal properties: if a safety property is violated, there will be a finite prefix witnessing it; on the other hand, for a liveness property, one can never provide such a finite negative evidence. We suggest the following alternative definitions of classes of temporal properties.

AFR/safety Always Finitely Refutable: when the property does not hold on an infinite execution, a *failed* verdict can be identified after a finite prefix.
AFS/guarantee Always Finitely Satisfiable: when the property is satisfied on an infinite execution, a *satisfied* verdict can be identified after a finite prefix.
NFR/liveness Never Finitely Refutable: when the property does not hold on an infinite execution, refutation can never be identified after a finite prefix.
NFS/morbidity Never Finitely Satisfiable: When the property is satisfied on an infinite execution, satisfaction can never be identified after a finite prefix.

It is easy to see that the definitions of the classes AFR and safety in [1] are the same and so are those for AFS and guarantee. A liveness property φ is defined to satisfy that any finite prefix can be extended to an execution that satisfies φ. The definition of the class NFR only mentions prefixes of executions that do not satisfy φ; but for prefixes of executions that satisfy φ this trivially holds. The correspondence between NFS and morbidity is shown in a symmetric way.

The above four classes of properties, however, do not cover the entire set of possible temporal properties, independent of the actual formalism that is used to express them. The following two classes complete the classification.

SFR Sometimes Finitely Refutable: for some infinite executions that violate the property, refutation can be identified after a finite prefix; for other infinite executions violating the property, this is not the case.
SFS Sometimes Finitely Satisfiable: for some infinite executions that satisfy the property, satisfaction can be identified after a finite prefix; for other infinite executions satisfying the property, this is not the case.

Bauer, Leucker and Schallhart [5] define three categories of prefixes of elements from 2^P.

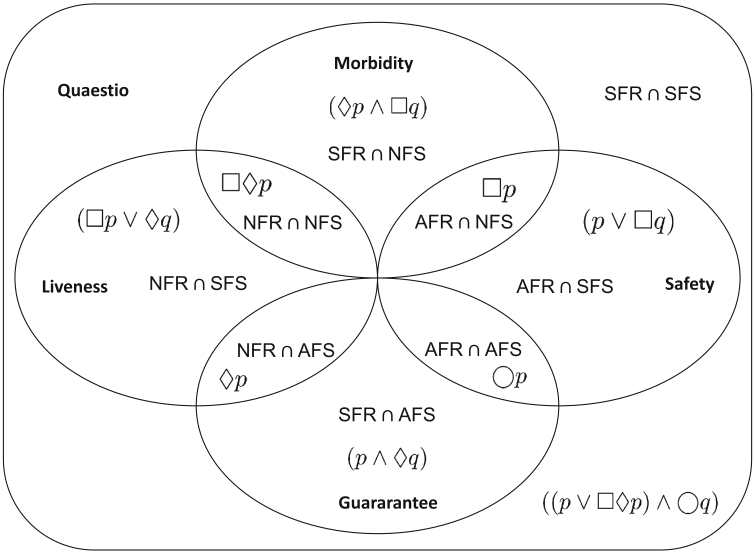


Fig. 1. Classification of properties: safety, guarantee, liveness, morbidity and quaestio.

- A *good* prefix is one where all its extensions (infinite sequences of elements from 2^P) satisfy the monitored property φ.
- A *bad* prefix is one where none of its infinite extensions satisfy φ.
- An *ugly* prefix cannot be extended into a good or a bad prefix.

When identifying a good or a bad finite prefix, we are done tracing the execution and can announce that the monitored property is satisfied or failed, respectively. After an ugly prefix, satisfaction or refutation of φ depends on the entire infinite execution, and cannot be determined in finite time. Note that a property has a good prefix if it is *not* a morbidity property, and a bad prefix if it is *not* a liveness property. *Monitorability* of a property φ is defined in [5] as the lack of ugly prefixes for the property φ. This definition is consistent with [24].

Any property that is in AFR (safety) or in AFS (guarantee) is monitorable [5, 9]. A property that is NFR ∩ NFS is non-monitorable. In fact no verdict is ever expected on any sequence that is monitored against such a property. This leaves the three classes SFR ∩ SFS, SFR ∩ NFS and NFR ∩ SFS, for which some properties are monitorable and others are not. This is demonstrated in the following table.

Class	Monitorable example	Non-monitorable example
SFR ∩ SFS	$((\Diamond r \vee \Box\Diamond p) \wedge \bigcirc q)$	$((p \vee \Box\Diamond p) \wedge \bigcirc q)$
SFR ∩ NFS	$(\Diamond p \wedge \Box q)$	$(\Box\Diamond p \wedge \bigcirc q)$
NFR ∩ SFS	$(\Box p \vee \Diamond q)$	$(\Box\Diamond p \vee \bigcirc q)$

The set of all properties *Prop* is not covered by safety, guarantee, liveness and morbidity. The missing properties are in SFR $\cap$ SFS. We call the class of such properties **Quaestio** (Latin for *question*). Figure 1 presents the relationship between the different classes of properties and their intersections, with LTL specification examples.

4 Runtime Verification for Propositional LTL

4.1 Runtime Verification for Propositional Future LTL

We present three algorithms. The first one is a classical algorithm for runtime verification of LTL (or Büchi automata) properties. The second algorithm can be used to check during run time what kind of verdicts can still be produced given the current prefix. The third algorithm can be used to check whether the property is monitorable.

Algorithm 1: Monitoring Sequences Using Automata

Kupferman and Vardi [17] provide an algorithm for detecting good and bad prefixes. For good prefixes, start by constructing a Büchi automaton $\mathcal{A}_{\neg\varphi}$ for $\neg\varphi$, e.g., using the translation in [11]. Note that this automaton is not necessarily deterministic [27]. States of $\mathcal{A}_{\neg\varphi}$, from which one cannot reach a cycle that contains an accepting state, are deleted. Checking for a positive verdict for φ, one keeps for each monitored prefix the set of states that $\mathcal{A}_{\neg\varphi}$ would be in after observing that input. One starts with the set of initial states of the automaton $\mathcal{A}_{\neg\varphi}$. Given the current set of successors S and an event $e \in 2^P$, the next set of successors S' is set to the successors of the states in S according to the transition relation Δ of $\mathcal{A}_{\neg\varphi}$. That is, $S' = \{s' | s \in S \wedge (s, e, s') \in \Delta\}$. Reaching the empty set of states, the monitored sequence is good, and the property must hold since the current prefix cannot be completed into an infinite execution satisfying $\neg\varphi$.

This is basically a *subset construction* for a deterministic automaton $\mathcal{B}_{\varphi}$, whose initial state is the set of initial states of $\mathcal{A}_{\neg\varphi}$, accepting state is the empty set, and transition relation as described above. The size of this automaton is $\mathcal{O}(2^{2^{|P|}})$, resulting in double exponential explosion from the size of the checked LTL property. But in fact, we do not need to construct the entire automaton $\mathcal{B}_{\varphi}$ in advance, and can avoid the double exponential explosion by calculating its current state on-the-fly, while performing runtime verification. Thus, the incremental processing per each event is exponential in the size of the checked LTL property. Unfortunately, a single exponential explosion is unavoidable [17].

Checking for a *failed* verdict for φ is done with a symmetric construction, translating φ into a Büchi automaton $\mathcal{A}_{\varphi}$ and then the deterministic automaton $\mathcal{B}_{\neg\varphi}$ (or calculating its states on-the-fly) using a subset construction as above. Note that $\mathcal{A}_{\neg\varphi}$ is used to construct $\mathcal{B}_{\varphi}$ and $\mathcal{A}_{\varphi}$ is used to construct $\mathcal{B}_{\neg\varphi}$. Runtime verification of φ uses both automata for the monitored input, reporting a *failed* verdict if $\mathcal{B}_{\neg\varphi}$ reaches an accepting state, a *satisfied* verdict if $\mathcal{B}_{\varphi}$ reaches an accepting state, and an *undecided* verdict otherwise. The algorithm guarantees to report a positive or negative verdict on the minimal good or bad prefix that is observed.

Algorithm 2: Checking Availability of Future Verdicts

We alter the above runtime verification algorithm to check whether positive or negative verdicts can still be obtained after the current monitored prefix at runtime. Applying DFS on $\mathcal{B}_\varphi$, we search for states from which one cannot reach the accepting state. We replace these states with a single state $\bot$ with a self loop, obtaining the automaton $\mathcal{C}_\varphi$. Reaching $\bot$, after monitoring a finite prefix σ with $\mathcal{C}_\varphi$ happens exactly when we will not have a good prefix anymore. This means that after σ, a *satisfied* verdict cannot be issued anymore for φ.

Similarly, we perform BFS on $\mathcal{B}_{\neg\varphi}$ to find all the states in which the accepting state is not reachable, then replace them by a single state $\top$ with a self loop, obtaining $\mathcal{C}_{\neg\varphi}$. Reaching $\top$ after monitoring a prefix means that we will not be able again to have a bad prefix, hence a *failed* verdict cannot be issued anymore for φ.

We can perform runtime verification while updating the state of both automata, $\mathcal{C}_\varphi$ and $\mathcal{C}_{\neg\varphi}$ on-the-fly, upon each input event. However, we need to be able to predict if, from the current state, an accepting state is not reachable. While this can be done in space exponential in φ, it makes an incremental calculation whose time complexity is doubly exponential in the size of φ, as is the algorithm for that by Pnueli and Zaks [24]. This is hardly a reasonable complexity for the incremental calculation performed between successive monitored events for an on-line algorithm. Hence, a pre-calculation of these two automata before the monitoring starts is preferable, leaving the incremental time complexity exponential in φ, as in Algorithm 1.

Algorithm 3: Checking Monitorability

A small variant on the construction of $\mathcal{C}_\varphi$ and $\mathcal{C}_{\neg\varphi}$ allows checking if a property is monitorable. The algorithm is simple: construct the product $\mathcal{C}_\varphi \times \mathcal{C}_{\neg\varphi}$ and check whether the state $(\bot, \top)$ is reachable. If so, the property is non-monitorable, since there is a prefix that will transfer the product automaton to this state and thus it is ugly. It is not sufficient to check separately that $\mathcal{C}_\varphi$ can reach $\top$ and that $\mathcal{C}_{\neg\varphi}$ can reach $\bot$. In the property $(\neg(p \wedge r) \wedge ((\neg p\, \mathcal{U}\, (r \wedge \Diamond q)) \vee (\neg r\, \mathcal{U}\, (p \wedge \Box q))))$: both $\bot$ and $\top$ can be reached, separately, depending on which of the predicates r or p happens first. But in either case, there is still a possibility for a good or a bad extension, hence it is a monitorable property.

If the automaton $\mathcal{C}_\varphi \times \mathcal{C}_{\neg\varphi}$ consists of only a single state $(\bot, \top)$, then there is no information whatsoever that we can obtain from monitoring the property.

The above algorithm is simple enough to construct, however its complexity is doubly exponential in the size of the given LTL property. This may not be a problem, as the algorithm is performed off-line and the LTL specifications are often quite short.

We show that checking monitorability is in EXPSPACE-complete. The upper bound is achieved by a binary search version of this algorithm[4]. For the lower bound we show a reduction from checking if a property is (not) a liveness property, a problem known to be in EXPSPACE-complete [18,25].

- We first neutralize bad prefixes. Now, when ψ is satisfiable, then $\Diamond\psi$ is monitorable (specifically, any prefix can be completed into a *good* prefix) iff ψ has a good prefix.
- Checking satisfiability of a property ψ is in PSPACE-complete [26][5].
- ψ has a good prefix iff ψ is not a morbidity property, i.e., if $\varphi = \neg\psi$ is not a liveness property.
- Now, φ is *not* a liveness property iff either φ is not satisfiable or $\Diamond\neg\varphi$ is monitorable.

4.2 Runtime Verification for Propositional Past LTL

Algorithm

The algorithm for past LTL, first presented in [14], is based on the observation that the semantics of the past time formulas $\ominus\varphi$ and $(\varphi\,\mathcal{S}\,\psi)$ in the current step i is defined in terms of the semantics in the previous step $i-1$ of a subformula, here recalled from Sect. 2:

- $\xi, i \models \ominus\varphi$ iff $i > 1$ and $\xi, i-1 \models \varphi$.
- $\xi, i \models (\varphi\,\mathcal{S}\,\psi)$ iff $\xi, i \models \psi$ or the following hold: $i > 1$, $\xi, i \models \varphi$ and $\xi, i-1 \models (\varphi\,\mathcal{S}\,\psi)$.

One only needs to look one step, or event, backwards in order to compute the new truth value of a formula and of its subformulas. The algorithm, shown below, operates on two vectors (arrays) of values indexed by subformulas: pre for the state before that event, and now for the current state (after the last seen event).

```
1. Initially, for each subformula $\varphi$, now($\varphi$) := $\mathit{false}$.
2. Observe a new event (as a set of ground predicates) $s$ as input.
3. Let pre := now.
4. Make the following updates for each subformula. If $\varphi$ is a subformula of $\psi$
   then now($\varphi$) is updated before now($\psi$).
   – now($\mathit{true}$) := $\mathit{true}$.
   – now(($\varphi \wedge \psi$)) := now($\varphi$) $\mathit{and}$ now($\psi$).
   – now($\neg\varphi$) := $\mathit{not}$ now($\varphi$).
   – now(($\varphi\ \mathcal{S}\ \psi$)) := now($\psi$) $\mathit{or}$ (now($\varphi$) $\mathit{and}$ pre(($\varphi\mathcal{S}\psi$))).
   – now($\ominus\ \varphi$) := pre($\varphi$).
5. Goto step 2.
```

[4] To show that a property is not monitorable, one needs to guess a state of $\mathcal{B}_\varphi \times \mathcal{B}_{\neg\varphi}$ and check that (1) it is reachable, and (2) one cannot reach from it an empty component, both for $\mathcal{B}_\varphi$ and for $\mathcal{B}_{\neg\varphi}$. (There is no need to construct $\mathcal{C}_\varphi$ or $\mathcal{C}_{\neg\varphi}$.).

[5] Proving that liveness was PSPACE-hard was shown in [3].

An Example

As an example[6], consider the formula $\mathit{close} \wedge \ominus \mathit{open}$, and suppose we evaluate it against the trace $\mathit{open}.\mathit{close}$ at step $i = 2$ (after seeing the close event). The algorithm performs the following assignments, resulting in the formula becoming true (assuming that $\mathsf{pre}(\mathit{open})$ is true):

$$\begin{aligned}
\mathsf{now}(\mathit{open}) &:= \mathit{false}\\
\mathsf{now}(\mathit{close}) &:= \mathit{true}\\
\mathsf{now}(\ominus \mathit{open}) &:= \mathsf{pre}(\mathit{open})\\
\mathsf{now}(\mathit{close} \wedge \ominus \mathit{open}) &:= \mathsf{now}(\mathit{close}) \wedge \mathsf{now}(\ominus \mathit{open})
\end{aligned}$$

The above suggested algorithm *interprets* a formula on a trace. As an alternative we can *synthesize* a program that is specialized for monitoring the property as in [14]. Figure 2 (left) shows a generated monitor program for the property. Two Boolean valued arrays pre for the previous state and now are declared and operated on. The indices $0 \ldots 3$ correspond to the enumeration of the subformulas shown in the Abstract Syntax Tree (AST) in Fig. 2 (right). For each observed event, the function evaluate() computes the now array from highest to lowest index, and returns true (property is satisfied in this position of the trace) iff now(0) is *true*.

```
class Formula_p extends Formula {
  var pre: Array[Boolean] = Array. fill (4)( false )
  var now: Array[Boolean] = Array. fill (4)( false )
  var tmp: Array[Boolean] = null

  override def evaluate(): Boolean = {
    now(3) = isTrue("open")
    now(2) = pre(3)
    now(1) = isTrue("close")
    now(0) = now(1).and(now(2))
    val result = now(0)
    tmp = now; now = pre; pre = tmp
    result
  }
}
```

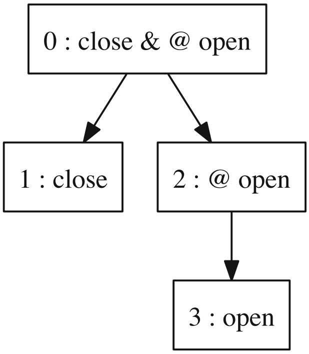


Fig. 2. Monitor (left) and AST (right) for propositional property.

[6] All examples of safety properties henceforth will omit the implied $\Box$ operator.

5 First-Order Past LTL

First-order past LTL allows universal and existential quantification over data occurring as parameters in events. Such events are referred to as *predicates* (or *parametric events*). Consider a predicate $open(f)$, indicating that a file f is being opened, and a predicate $close(f)$ indicating that f is being closed. We can formulate that a file cannot be closed unless it was opened before with the following first-order past LTL formula:

$$\forall f\,(close(f) \longrightarrow \mathbf{P}\, open(f)) \tag{1}$$

Here **P** is the "sometimes in the past" temporal operator. This property must be checked for every monitored event. Already in this very simple example we see that we need to store *all* the names of files that were previously opened so we can compare to the files that are being closed. A more refined specification would be the following, requiring that a file can be closed only if it was opened before, and has not been closed since. Here, we use the temporal operators $\ominus$ ("at previous step") and $\mathcal{S}$ ("since"):

$$\forall f\,(close(f) \longrightarrow \ominus(\neg close(f)\,\mathcal{S}\, open(f))) \tag{2}$$

One problem we need to solve is the unboundedness caused by negation. For example, assume that we have only observed so far one *close* event $close$("ab"). The subformula $close(f)$ is therefore satisfied for the value $f =$ "ab". The subformula $\neg close(f)$ is satisfied by all values from the domain of f *except* for "ab". This set contains those values that we have not seen yet in the input within a *close* event. We need a representation of finite and infinite sets of values, upon which applying complementation is efficient. We present a first-order past time temporal logic, named QTL (Quantified Temporal Logic), and an implementation, named DEJA VU based on a BDD (Binary Decision Diagram) representation of sets of assignments of values to the free variables of subformulas.

5.1 Syntax

Assume a finite set of domains $D_1, D_2, \ldots$. Assume further that the domains are infinite, e.g., they can be the integers or strings[7]. Let V be a finite set of *variables*, with typical instances x, y, z. An *assignment* over a set of variables W maps each variable $x \in W$ to a value from its associated domain $domain(x)$. For example $[x \rightarrow 5, y \rightarrow$ "abc"$]$ maps x to 5 and y to "abc". Let T be a set of *predicate names* with typical instances p, q, r. Each predicate name p is associated with some domain $domain(p)$. A predicate is constructed from a predicate name, and a variable or a constant of the same type. Thus, if the predicate name p and the variable x are associated with the domain of strings, we have predicates like p("gaga"), p("baba") and $p(x)$. Predicates over constants are called *ground predicates*. An *event* is a finite set of ground predicates. For example, if $T =$

[7] For dealing with finite domains see [12].

$\{p, q, r\}$, then $\{p(\text{“xyzzy”}), q(3)\}$ is a possible event. An *execution* $\sigma = s_1 s_2 \dots$ is a finite sequence of events.

For runtime verification, a property φ is interpreted on prefixes of a monitored sequence. We check whether φ holds for every such prefix, hence, conceptually, check whether $\Box\varphi$ holds, where $\Box$ is the “always in the future” linear temporal logic operator. The formulas of the core logic QTL are defined by the following grammar. For simplicity of the presentation, we define here the logic with unary predicates, but this is not due to any principle limitation, and, in fact, our implementation supports predicates with multiple arguments.

$$\varphi ::= \mathit{true} \mid p(a) \mid p(x) \mid (\varphi \wedge \varphi) \mid \neg\varphi \mid (\varphi\ \mathcal{S}\ \varphi) \mid \ominus \varphi \mid \exists x\ \varphi$$

The formula $p(a)$, where a is a constant in $\mathit{domain}(p)$, means that the ground predicate $p(a)$ occurs in the most recent event. The formula $p(x)$, for a variable $x \in V$, holds with a binding of x to the value a if a ground predicate $p(a)$ appears in the most recent event. The formula $(\varphi_1\ \mathcal{S}\ \varphi_2)$ means that φ_2 held in the past (possibly now) and since then φ_1 has been true. The property $\ominus\ \varphi$ means that φ was true in the previous event. We can also define the following additional operators: $\mathit{false} = \neg\mathit{true}$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \longrightarrow \psi) = (\neg\varphi \vee \psi)$, $\mathbf{P}\ \varphi = (\mathit{true}\ \mathcal{S}\ \varphi)$ (*previously* φ), $\mathbf{H}\ \varphi = \neg\mathbf{P}\ \neg\varphi$ (*historically* φ, or φ *always in the past*), and $\forall x\ \varphi = \neg\exists x \neg\varphi$. The operator $[\varphi, \psi)$, borrowed from [16], has the same meaning as $(\neg\psi\ \mathcal{S}\ \varphi)$, but reads more naturally as an interval.

5.2 Semantics

Predicate Semantics

Let $\mathit{free}(\varphi)$ be the set of free (i.e., unquantified) variables of a subformula φ. Then $(\gamma, \sigma, i) \models \varphi$, where γ is an assignment over $\mathit{free}(\varphi)$, and $i \geq 1$, if φ holds for the prefix $s_1 s_2 \dots s_i$ of the execution σ with the assignment γ. We denote by $\gamma|_{\mathit{free}(\varphi)}$ the restriction (projection) of an assignment γ to the free variables appearing in φ and by ϵ the empty assignment. The semantics of QTL can be defined as follows.

- $(\epsilon, \sigma, i) \models \mathit{true}$.
- $(\epsilon, \sigma, i) \models p(a)$ if $p(a) \in \sigma[i]$.
- $([x \mapsto a], \sigma, i) \models p(x)$ if $p(a) \in \sigma[i]$.
- $(\gamma, \sigma, i) \models (\varphi \wedge \psi)$ if $(\gamma|_{\mathit{free}(\varphi)}, \sigma, i) \models \varphi$ and $(\gamma|_{\mathit{free}(\psi)}, \sigma, i) \models \psi$.
- $(\gamma, \sigma, i) \models \neg\varphi$ if not $(\gamma, \sigma, i) \models \varphi$.
- $(\gamma, \sigma, i) \models (\varphi\, \mathcal{S}\, \psi)$ if $(\gamma|_{\mathit{free}(\psi)}, \sigma, i) \models \psi$ or the following hold: $i > 1$, $(\gamma|_{\mathit{free}(\varphi)}, \sigma, i) \models \varphi$, and $(\gamma, \sigma, i-1) \models (\varphi\, \mathcal{S}\, \psi)$.
- $(\gamma, \sigma, i) \models \ominus\varphi$ if $i > 1$ and $(\gamma, \sigma, i-1) \models \varphi$.
- $(\gamma, \sigma, i) \models \exists x\ \varphi$ if there exists $a \in \mathit{domain}(x)$ such that[8] $(\gamma\, [x \mapsto a], \sigma, i) \models \varphi$.

[8] $\gamma\, [x \mapsto a]$ is the overriding of γ with the binding $[x \mapsto a]$.

Set Semantics

It helps to present the BDD-based algorithm by first redefining the semantics of the logic in terms of sets of assignments satisfying a formula. Let $I[\varphi, \sigma, i]$ be the semantic function, defined below, that returns a set of assignments such that $\gamma \in I[\varphi, \sigma, i]$ iff $(\gamma, \sigma, i) \models \varphi$. The empty set of assignments $\emptyset$ behaves as the Boolean constant 0 and the singleton set that contains an assignment over an empty set of variables $\{\epsilon\}$ behaves as the Boolean constant 1. We define the union and intersection operators on sets of assignments, even if they are defined over non identical sets of variables. In this case, the assignments are extended over the union of the variables. Thus intersection between two sets of assignments A_1 and A_2 is defined like a database "join" operator; i.e., it consists of the assignments whose projection on the *common* variables agrees with an assignment in A_1 and with an assignment in A_2. Union is defined as the dual operator of intersection. Let A be a set of assignments over the set of variables W; we denote by $\mathit{hide}(A, x)$ (for "hiding" the variable x) the set of assignments obtained from A after removing from each assignment the mapping from x to a value. In particular, if A is a set of assignments over only the variable x, then $\mathit{hide}(A, x)$ is $\{\epsilon\}$ when A is nonempty, and $\emptyset$ otherwise. $A_{\mathit{free}(\varphi)}$ is the set of all possible assignments of values to the variables that appear free in φ. We add a 0 position for each sequence σ (which starts with s_1), where I returns the empty set for each formula. The assignment-set semantics of QTL is shown in the following. For all occurrences of i, it is assumed that $i > 0$.

- $I[\varphi, \sigma, 0] = \emptyset$.
- $I[\mathit{true}, \sigma, i] = \{\epsilon\}$.
- $I[p(a), \sigma, i] =$ if $p(a) \in \sigma[i]$ then $\{\epsilon\}$ else $\emptyset$.
- $I[p(x), \sigma, i] = \{[x \mapsto a] | p(a) \in \sigma[i]\}$.
- $I[(\varphi \wedge \psi), \sigma, i] = I[\varphi, \sigma, i] \bigcap I[\psi, \sigma, i]$.
- $I[\neg\varphi, \sigma, i] = A_{\mathit{free}(\varphi)} \setminus I[\varphi, \sigma, i]$.
- $I[(\varphi\ \mathcal{S}\ \psi), \sigma, i] = I[\psi, \sigma, i] \bigcup (I[\varphi, \sigma, i] \bigcap I[(\varphi \mathcal{S} \psi), \sigma, i-1])$.
- $I[\ominus\varphi, \sigma, i] = I[\varphi, \sigma, i-1]$.
- $I[\exists x\ \varphi, \sigma, i] = \mathit{hide}(I[\varphi, \sigma, i], x)$.

As before, the interpretation for the rest of the operators can be obtained from the above using the connections between the operators.

6 Runtime Verification for First-Order Past LTL

We describe an algorithm for monitoring QTL properties, first presented in [12] and implemented in the tool DEJA VU. To give a brief overview of the contents of this section, instead of storing the data values occurring in events, we enumerate these data values as soon as we see them and use Boolean encodings of this enumeration. We use BDDs to represent sets of such enumerations. For example, if the runtime verifier sees the input events *open*("a"), *open*("b"), *open*("c"), it will encode them as 000, 001 and 010 (say, we use 3 bits b_0, b_1 and b_2 to represent each enumeration, with b_2 being the most significant bit). A BDD that represents the set of values {"a", "c"} would be equivalent to a Boolean function $(\neg b_0 \wedge \neg b_2)$ that returns 1 for 000 and 010 (the value of b_1 can be arbitrary). This approach has the following benefits:

- It is highly compact. With k bits used for representing enumerations, the BDD can grow to $2^{\mathcal{O}(k)}$ nodes [6]; but BDDs usually compact the representation very well [8]. In fact, we often do not pay much in overhead for keeping surplus bits. Thus, we can start with an overestimated number of bits k such that it is unlikely to see more than 2^k different values for the domain they represent. We can also incrementally extend the BDD with additional bits when needed at runtime.
- Complementation (negation) is efficient, by just switching between the 0 and 1 leaves of the BDD. Moreover, even though at any point we may have not seen the entire set of values that will show up during the execution, we can safely (and efficiently) perform complementation: values that have not appeared yet in the execution are being accounted for and their enumerations are reserved already in the BDD before these values appear.
- Our representation of sets of assignments as BDDs allows a very simple algorithm that naturally extends the dynamic programming monitoring algorithm for propositional past time temporal logic shown in [14] and summarized in Sect. 4.2.

6.1 BDDs

We represent a set of assignments as an Ordered Binary Decision Diagram (OBDD, although we write simply BDD) [7]. A BDD is a compact representation for a Boolean valued function of type $\mathbb{B}^k \to \mathbb{B}$ for some $k > 0$ (where $\mathbb{B}$ is the Boolean domain $\{0, 1\}$), as a directed acyclic graph (DAG). A BDD is essentially a compact representation of a Boolean tree, where compaction glues together isomorphic subtrees. Each non-leaf node is labeled with one of the Boolean variables $b_0, \ldots, b_{k-1}$. A non-leaf node b_i is the source of two arrows leading to other nodes. A dotted-line arrow represents that b_i has the Boolean value 0, while a thick-line arrow represents that it has the value 1. The nodes in the DAG have the same order along all paths from the root. However, some of the nodes may be absent along some paths, when the result of the Boolean function does not depend on the value of the corresponding Boolean variable. Each path leads to a leaf node that is marked by either a 0 or a 1, representing the Boolean value returned by the function for the Boolean values on the path. Figure 3 contains five BDDs (a)-(e), over three Boolean variables b_0, b_1, and b_2 (referred to by their subscripts 0, 1, and 2), as explained below.

6.2 Mapping Data to BDDs

Assume that we see p("ab"), p("de"), p("af") and q("fg") in subsequent states in a trace, where p and q are predicates over the domain of strings. When a value associated with a variable appears for the first time in the current event (in a ground predicate), we add it to the set of values of that domain that were seen. We assign to each new value an *enumeration*, represented as a binary number, and use a hash table to point from the value to its enumeration.

Consistent with the DEJA VU implementation, the least significant bit in an enumeration is denoted in Fig. 3 (and in the rest of this paper) by BDD variable 0, and the most significant bit by BDD variable $n-1$, where n is the number of bits. Using e.g. a three-bit enumeration $b_2b_1b_0$, the first encountered value "ab" can be represented as the bit string 000, "de" as 001,"af" as 010 and "fg" as 011. A BDD for a subset of these values returns a 1 for each bit string representing an enumeration of a value in the set, and 0 otherwise. E.g. a BDD representing the set {"de","af"} (2nd and 3rd values) returns 1 for 001 and 010. This is the Boolean function $\neg b_2 \wedge (b_1 \leftrightarrow \neg b_0)$. Figure 3 shows the BDDs for each of these values as well as the BDD for the set containing the values "de" and"af".

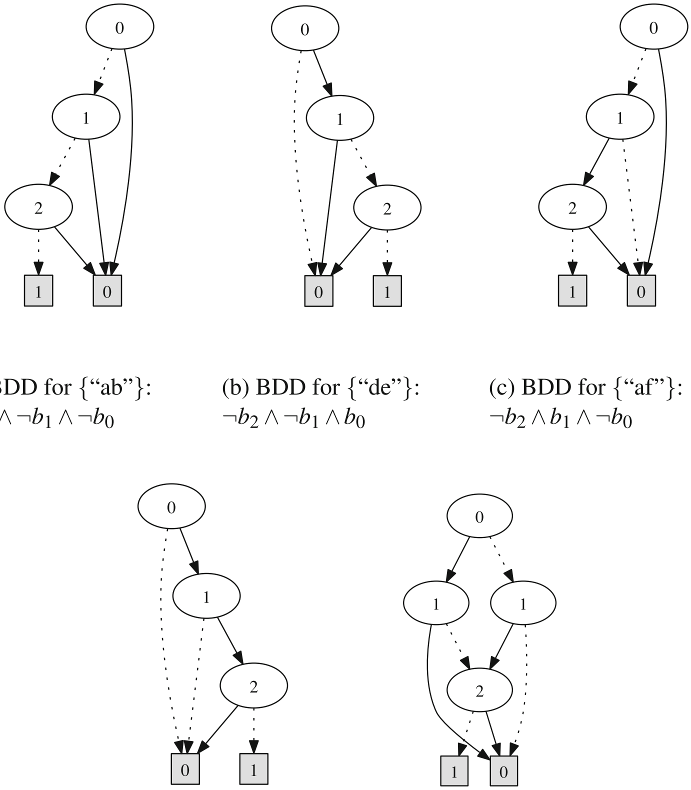
When representing a set of assignments for e.g. two variables x and y with k bits each, we will have Boolean variables $x_0, \ldots, x_{k-1}, y_0, \ldots, y_{k-1}$. A BDD will return a 1 for each bit string representing the concatenation of enumerations that correspond to the represented assignments, and 0 otherwise. For example, to represent the assignments $[x \mapsto \text{“de”}, y \mapsto \text{“af”}]$, where "de" is enumerated as 001 and"af" with 010, the BDD will return a 1 for 001010.

6.3 The BDD-based Algorithm

Given some ground predicate $p(a)$ observed in the execution matching with $p(x)$ in the monitored property, let **lookup**(x, a) be the enumeration of a. If this is a's first occurrence, then it will be assigned a new enumeration. Otherwise, **lookup** returns the enumeration that a received before. We can use a counter, for each variable x, counting the number of different values appearing so far for x. When a new value appears, this counter is incremented, and the value is converted to a Boolean representation. Enumerations that were not yet used represent the values not seen yet. In the next section we introduce data reclaiming, which allows reusing enumerations for values that no longer affect the checked property. This involves a more complicated enumeration mechanism.

The function **build**(x, A) returns a BDD that represents the set of assignments where x is mapped to (the enumeration of) v for $v \in A$. This BDD is independent of the values assigned to any variable other than x, i.e., they can have any value. For example, assume that we use three Boolean variables (bits) x_0, x_1 and x_2 for representing enumerations over x (with x_0 being the least significant bit), and assume that $A = \{a, b\}$, **lookup**$(x, a) = 011$, and **lookup**$(x, b) = 001$. Then **build**(x, A) is a BDD representation of the Boolean function $x_0 \wedge \neg x_2$.

Intersection and union of sets of assignments are translated simply to conjunction and disjunction of their BDD representation, respectively, and complementation becomes BDD negation. We will denote the Boolean BDD operators as **and**, **or** and **not**. To implement the existential (universal, respectively) operators, we use the BDD existential (universal, respectively) operators over the



(a) BDD for {“ab”}: $\neg b_2 \wedge \neg b_1 \wedge \neg b_0$

(b) BDD for {“de”}: $\neg b_2 \wedge \neg b_1 \wedge b_0$

(c) BDD for {“af”}: $\neg b_2 \wedge b_1 \wedge \neg b_0$

(d) BDD for {“fg”}: $\neg b_2 \wedge b_1 \wedge b_0$

(e) BDD for {“de”, “af”}: $\neg b_2 \wedge (b_1 \leftrightarrow \neg b_0)$

Fig. 3. BDDs for the trace: $p(\text{“ab”}).p(\text{“de”}).p(\text{“af”}).q(\text{“fg”})$.

Boolean variables that represent (the enumerations of) the values of x. Thus, if B_φ is the BDD representing the assignments satisfying φ in the current state of the monitor, then $\mathbf{exists}(\langle x_0, \ldots, x_{k-1}\rangle, B_\varphi)$ is the BDD that represents the assignments satisfying $\exists x\ \varphi$ in the current state. Finally, BDD($\bot$) and BDD($\top$) are the BDDs that return always 0 or 1, respectively.

The dynamic programming algorithm, shown below, works similarly to the algorithm for the propositional case shown in Sect. 4.2. That is, it operates on two vectors (arrays) of values indexed by subformulas: pre for the state before that event, and now for the current state (after the last seen event). However, while in the propositional case the vectors contain Boolean values, here they contain BDDs. The algorithm follows.

1. Initially, for each subformula φ, $\mathsf{now}(\varphi) := \mathrm{BDD}(\bot)$.
2. Observe a new event (as a set of ground predicates) s as input.
3. Let $\mathsf{pre} := \mathsf{now}$.
4. Make the following updates for each subformula. If φ is a subformula of ψ then $\mathsf{now}(\varphi)$ is updated before $\mathsf{now}(\psi)$.
   - $\mathsf{now}(\mathit{true}) := \mathrm{BDD}(\top)$.
   - $\mathsf{now}(p(a)) :=$ if $p(a) \in s$ then $\mathrm{BDD}(\top)$ else $\mathrm{BDD}(\bot)$.
   - $\mathsf{now}(p(x)) := \mathbf{build}(x, A)$ where $A = \{a | p(a) \in s\}$.
   - $\mathsf{now}((\varphi \wedge \psi)) := \mathbf{and}(\mathsf{now}(\varphi), \mathsf{now}(\psi))$.
   - $\mathsf{now}(\neg\varphi) := \mathbf{not}(\mathsf{now}(\varphi))$.
   - $\mathsf{now}((\varphi\ \mathcal{S}\ \psi)) := \mathbf{or}(\mathsf{now}(\psi), \mathbf{and}(\mathsf{now}(\varphi), \mathsf{pre}((\varphi \mathcal{S} \psi))))$.
   - $\mathsf{now}(\ominus\ \varphi) := \mathsf{pre}(\varphi)$.
   - $\mathsf{now}(\exists x\ \varphi) := \mathbf{exists}(\langle x_0, \ldots, x_{k-1}\rangle, \mathsf{now}(\varphi))$.
5. Goto step 2.

An important property of the algorithm is that, at any point during monitoring, enumerations that are not used in the pre and now BDDs represent all values that have *not* been seen so far in the input. This can be proved by induction on the size of temporal formulas and the length of the input sequence. We specifically identify one enumeration to represent all values not seen yet, namely the largest possible enumeration, given the number of bits we use, $11\ldots11$. We let $(11\ldots11)$ denote the BDD that returns 1 exactly for this value. This trick allows us to use a finite representation and quantify existentially and universally over *all* values in infinite domains.

6.4 An Example

DEJA VU is implemented in SCALA. DEJA VU takes as input a specification file containing one or more properties, and synthesizes a self-contained SCALA program (a text file) - the monitor, as already illustrated for the propositional case in Sect. 4.2. This program (which first must be compiled) takes as input the trace file and analyzes it. The tool uses the JavaBDD library for BDD manipulations [15]. We shall illustrate the monitor generation using an example. Consider the following property stating that if a file f is closed, it must have been opened in the past with some access mode m (e.g. ‘read’ or ‘write’ mode):

$$\forall f\ (close(f) \longrightarrow \exists m\ \mathbf{P}\ open(f,m))$$

The property-specific part[9] of the synthesized monitor, shown in Fig. 4 (left), relies on the enumeration of the subformulas shown in Fig. 4 (right). As in the propositional case, two arrays are declared, indexed by subformula indexes: pre for the previous state and now for the current state, although here storing BDDs instead of Boolean values. For each observed event, the function evaluate() computes the now array from highest to lowest index, and returns true (property is satisfied in this position of the trace) iff now(0) is not BDD(⊥). At composite subformula nodes, BDD operators are applied. For example for subformula 4, the new value is now(5).or(pre(4)), which is the interpretation of the formula **P** open(f, m) corresponding to the law: $\mathbf{P}\,\varphi = (\varphi \vee \ominus\, \mathbf{P}\,\varphi)$. As can be seen, for each new event, the evaluation of a formula results in the computation of a BDD for each subformula.

```
class Formula_p extends Formula {
  var pre: Array[BDD] = Array. fill (6)(False)
  var now: Array[BDD] = Array. fill (6)(False)
  var tmp: Array[BDD] = null
  val var_f :: var_m :: Nil =
    declareVariables("f", "m")

  override def evaluate(): Boolean = {
    now(5) = build("open")(V("f"),V("m"))
    now(4) = now(5).or(pre(4))
    now(3) = now(4).exist(var_m)
    now(2) = build("close")(V("f"))
    now(1) = now(2).not().or(now(3))
    now(0) = now(1).forAll(var_f)
    val result = !now(0).isZero
    tmp = now; now = pre; pre = tmp
    result
  }
}
```

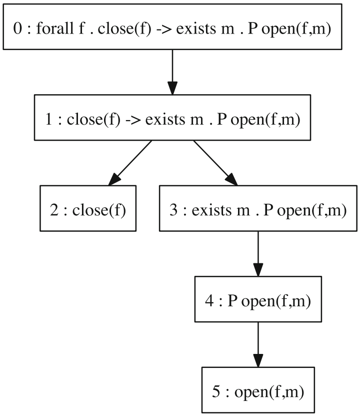

0 : forall f . close(f) -> exists m . P open(f,m)
1 : close(f) -> exists m . P open(f,m)
2 : close(f)
3 : exists m . P open(f,m)
4 : P open(f,m)
5 : open(f,m)

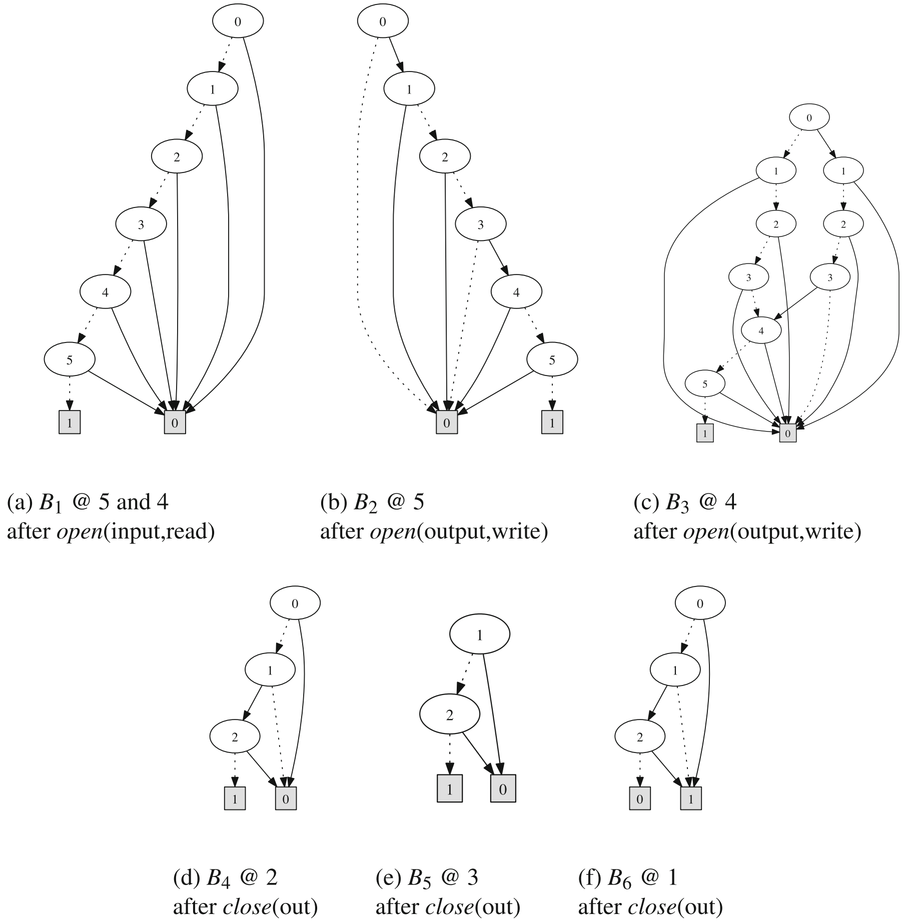
Fig. 4. Monitor (left) and AST (right) for the property.

We shall briefly evaluate the example formula on a trace. Assume that each variable f and m is represented by three BDD bits. Consider the input trace, consisting of three events[10]:

$$open(\text{input,read}).open(\text{output,write}).close(\text{out})$$

[9] An additional 600+ lines of property independent boilerplate code is generated.

[10] Traces accepted by the tool are concretely in CSV format. For example the first event is a single line of the form: `open,input,read`.



(a) B_1 @ 5 and 4
after *open*(input,read)

(b) B_2 @ 5
after *open*(output,write)

(c) B_3 @ 4
after *open*(output,write)

(d) B_4 @ 2
after *close*(out)

(e) B_5 @ 3
after *close*(out)

(f) B_6 @ 1
after *close*(out)

Fig. 5. Selected BDDs, named $B_1, \ldots, B_6$, computed after each event at various subformula nodes, indicated by BDD B_i @ *node* (see Fig. 4), during processing of the trace: *open*(input,read)*.open*(output,write)*.close*(out).

When the monitor evaluates subformula 5 on the first event *open*(input, read), it will create a bit string composed of a bit string for each parameter f and m. As previously explained, bit strings for each variable are allocated in increasing order: 000, 001, 010,..., hence the first bit string representing the assignment [$\mathsf{f} \mapsto$ input, $\mathsf{m} \mapsto$ read] becomes 000000 where the three rightmost bits represent the assignment of input to f, and the three leftmost bits represent the assignment of read to m. Figure 5a shows the corresponding BDD B_1. Recall that most significant bits are implemented lower in the BDD, and that for each bit (node) in the BDD, the dotted arrow corresponds to this bit being 0 and the full drawn arrow corresponds to this bit being 1. In this BDD all bits have to be zero in order to be accepted by the function represented by the BDD. We will not show how all the tree nodes evaluate, except observe that node 5 assumes the same BDD value as node 4 (all the seen values in the past), and conclude that since no *close*(...) event has been observed, the top-level formula (node 0) is true at this position in the trace.

Upon the second *open*(output,write) event, new values (output,write) are observed as argument to the *open* event. Hence a new bit string for each variable f and m is allocated, in both cases 001 (the next unused bit string for each variable). The new combined bit string for the assignments satisfying subformula 5 then becomes 001001, forming a BDD representing the assignment [$\mathsf{f} \mapsto$ output, $\mathsf{m} \mapsto$ write], and appearing in Fig. 5b as B_2. The computation of the BDD for node 4 is computed by $\mathsf{now(4)} = \mathsf{now(5).or(pre(4))}$, which results in the BDD B_3, representing the set of the two so far observed assignments ($B_3 = \mathbf{or}(B_1, B_2)$).

Upon the third *close*(out) event, a new value out for f is observed, and allocated the bit pattern 010, represented by the BDD B_4 for subformula 2. At this point node 4 still evaluates to the BDD B_3 (unchanged from the previous step), and the existential quantification over m in node 3 results in the BDD B_5, where the bits 3, 4 and 5 for m have been removed, and the BDD compacted. Node 1 is computed as $\mathbf{or}(\mathbf{not}(B_4),\ B_5)$, which results in the BDD B_6. This BDD represents all bit patterns for f that are **not** 010, corresponding to the value: out. So for all such values the formula is true. This means, however, that the top-level formula in node 0 is not true (violated by bit pattern 010), and hence the formula is violated on the third event.

6.5 Dynamic Data Reclamation

Consider Property 1 on page 10 that asserts that each file that is closed was opened before. If we do not remember for this property *all* the files that were opened, then we will not be able to check when a file is closed whether it was opened before. Consider now the more refined Property 2 on page 10, requiring that a file can be closed only if it was opened before, and has not been closed. Observe here that if a file was opened and subsequently closed, then if it is closed again before opening, the property would be invalidated just as in the case where it was not opened at all. This means that we can "forget" that a file was opened when it is closed without affecting our ability to monitor the formula. This allows reusing enumerations of data values, when this does not affect the decision whether the property holds or not.

Let A be a set of assignments over some variables that include x. Denote by $A[x = a]$ the set of assignments from A in which the value of x is a. We say that the values a and b are *analogous* for variable x in A, if $\mathit{hide}(A[x = a], x) = \mathit{hide}(A[x = b], x)$. This means that a and b, as values of the variable x, are related to all other values in A in the same way. A value can be reclaimed if it is analogous to the values not seen yet in all the assignments represented in $\mathsf{pre}(\psi)$, for each subformula ψ.

We shall now identify enumerations that can be reclaimed, and remove the values in the hash table that map to them, such that the enumerations can later be reused to represent new values. The search for reclaimable enumerations in a particular step during monitoring is performed on the pre BDDs. Recall that the enumeration $11 \dots 11$ represents all the values that were *not* seen so far. Thus, we can check whether a value a for x is analogous to the values not seen so far for x by performing the checks on the pre BDDs between the enumeration of a and the enumeration $11 \dots 11$. In fact, we do not have to perform the checks enumeration by enumeration, but use a BDD expression that constructs a BDD representing (returning 1 for) all enumerations that can be reclaimed for a variable x.

Assume that a subformula ψ has three free variables, x, y and z, each with k bits, i.e., $x_0, \dots, x_{k-1}$, $y_0, \dots, y_{k-1}$ and $z_0, \dots, z_{k-1}$. The following expression returns a BDD representing the enumerations for values of x in assignments represented by $\mathsf{pre}(\psi)$ that are related to enumerations of y and z in the same way as $11 \dots 11$.

$$I_{\psi,x} = \forall y_0 \dots \forall y_{k-1} \forall z_0 \dots \forall z_{k-1} (\mathsf{pre}(\psi)[x_0 \setminus 1, \dots x_{k-1} \setminus 1] \leftrightarrow \mathsf{pre}(\psi))$$

We now conjoin the above formula over each subformula that has a temporal operator at the outermost level, and subtract from this conjunction the $11 \dots 11$ enumeration. This becomes the BDD *avail* of available enumerations. Any enumeration that is in *avail* can be reclaimed, and later reused as the enumeration of a new value. The selection of a "free" enumeration from *avail* can be performed by a SAT solver that picks any enumeration that satisfies it, followed by removing that enumeration from *avail* to indicate that it is no longer available. Note that if a value later reappears after its enumeration was reclaimed, it is likely to be assigned a different enumeration.

7 Conclusion

We presented a collection of runtime verification algorithms for linear temporal logics. First we introduced propositional temporal logic, including future as well as past time operators. We presented a theory of monitorability of temporal properties, introducing classes that reflect different degrees of monitorability. The notion of monitorability identifies the kinds of verdicts that one can obtain from observing finite prefixes of an execution. We then presented monitoring algorithms for the future time case as automata, and for the past time case as an instance of dynamic programming. We also provided algorithms for checking what kind of verdict (positive or negative) we can expect after monitoring a certain prefix against a given property, and whether a property is monitorable or not. We then introduced first-order past time linear temporal logic, and a monitoring algorithm for it. While the propositional version is independent of the length of the prefix seen so far, the first-order version may need to represent an amount of values that can grow linearly with the number of data values observed so far. The challenge is to provide a compact representation that will grow slowly and can be updated quickly with each incremental calculation that is performed per each new monitored event. We used a BDD representation of sets of assignments for the variables that appear in the monitored property.

References

1. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distrib. Comput. **2**(3), 117–126 (1987)
2. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
3. Basin, D.A., Jiménez, C.C., Klaedtke, F., Zalinescu, E.: Deciding safety and liveness in TPTL. Inf. Process. Lett. **114**(12), 680–688 (2014)
4. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 126–138. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77395-5_11
5. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Method. **20**(4), 14:1–14:64 (2011)
6. Bryant, R.E.: On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. IEEE Trans. Comput. **40**(2), 205–213 (1991)
7. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
8. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: LICS 1990, pp. 428–439 (1990)
9. Falcone, Y., Fernandez, J.-C., Mounier, L.: What can you verify and enforce at runtime? STTT **14**(3), 349–382 (2012)
10. Gabbay, D., Pnueli, A., Shelah, S., Stavi, J.: On the temporal analysis of fairness. In: POPL 1980, pp. 163–173. ACM (1980)
11. Gerth, R., Peled, D.A., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: PSTV 1995, pp. 3–18 (1995)
12. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with BDDs. In: FMCAD 2017, pp. 116–123. IEEE (2017)
13. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring events that carry data. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 61–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_3

14. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_24
15. JavaBDD. http://javabdd.sourceforge.net
16. Kim, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: a run-time assurance tool for Java. In: RV 2001. Elsevier (2001). ENTCS **55**(2), 218–235
17. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods Syst. Des. **19**(3), 291–314 (2001)
18. Kupferman, O., Vardi, G.: On relative and probabilistic finite counterability. Formal Methods Syst. Des. **52**(2), 117–146 (2018)
19. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **3**(2), 125–143 (1977)
20. Manna, Z., Pnueli, A.: Completing the temporal picture. Theor. Comput. Sci. **83**, 91–130 (1991)
21. Markey, N.: Temporal logic with past is exponentially more succinct, concurrency column. Bull. EATCS **79**, 122–128 (2003)
22. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT **14**(3), 249–289 (2012). Springer
23. Peled, D., Havelund, K.: Refining the safety-liveness classification of temporal properties according to monitorability. LNCS (2018, submitted)
24. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: Misra, Jayadev, Nipkow, Tobias, Sekerinski, Emil (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006). https://doi.org/10.1007/11813040_38
25. Sistla, A.P.: Safety, liveness and fairness in temporal logic. Formal Aspects Comput. **6**(5), 495–512 (1994)
26. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. In: STOC 1982, pp. 159–168 (1982). J. ACM (JACM), **32**(3), 733–749, July 1985. JACM Homepage archive
27. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pp. 133–192 (1990)