First-Order Timed Runtime Verification using BDDs*

Klaus Havelund[1] and Doron Peled[2]

[1]Jet Propulsion Laboratory,
California Institute of Technology, USA
[2] Department of Computer Science
Bar Ilan University, Israel

Abstract. Runtime Verification (RV) expedites the analyses of execution traces for detecting system errors and for statistical and quality analysis. Having started modestly, with checking temporal properties that are based on propositional (yes/no) values, the current practice of RV often involves properties that are parameterized by the data observed in the input trace. The specifications are based on various formalisms, such as automata, temporal logics, rule systems and stream processing. Checking execution traces that are data intensive against a specification that requires strong dependencies between the data poses a non-trivial challenge; in particular if runtime verification has to be performed online, where many events that carry data appear within small time proximities. Towards achieving this goal, we recently suggested to represent relations over the observed data values as BDDs, where data elements are enumerated and then converted into bit vectors. We extend here the capabilities of BDD-based RV with the ability to express timing constraints, where the monitored events include clock values. We show how to efficiently operate on BDDs that represent both relations on (enumerations of) values and time dependencies. We demonstrate our algorithm with an efficient implementation and provide experimental results.

1 Introduction

Runtime verification provides techniques for monitoring system executions, online and offline, against a formal specification. The monitored system is instrumented to report to the monitor on the occurrence of relevant events that may also include related data values. The monitor observes the input events and keeps some internal *summary* of the prefix of the execution observed so far, which allows computing whether an evidence for a violation of the specification is already available. RV can complement the use of testing and verification techniques during the system development, e.g. by performing offline log file analysis. It can also be used online as part of protecting a system against an unwanted situation and averting it [26]. This is particularly important in safety-critical systems such as aerospace systems, transportation systems, power plants, and medicine.

* 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 1464/18: "Efficient Runtime Verification for Systems with Lots of Data and its Applications".

One main challenge in applying RV is to increase the scope of the properties that can be monitored. The goal is to provide algorithms for monitoring richer, and yet succinct specification formalism while ensuring that the algorithms are efficient enough to catch up with the speed of information arrival; especially if we want to apply them online. We recently suggested to use BDDs [11, 12] to represent relations between data elements that appear during the execution. We extend this approach and present here a BDD-based algorithm for full first-order linear temporal logic with time constraints. Consider the following property (the syntax and semantics will be described later).

$$(close \rightarrow \mathbf{P}\, open) \tag{1}$$

It expresses that when *close* happens, *open* must have already happened (**P** stands for *previously*). To monitor this property, it is enough to remember if *open* was reported to the monitor so that it can be checked when *close* is reported. The classical algorithm [23] keeps two sets of Boolean variables, pre and now, in the summary, for the *previous* and the *current* value of each subformula, respectively. These variables are updated every time a new event is reported. For example, for the property $\mathbf{P}\, open$ (*open* has happened in the past), we keep $\mathsf{pre}(\mathbf{P}\, open)$ and $\mathsf{now}(\mathbf{P}\, open)$ and update $\mathsf{now}(\mathbf{P}\, open) := \mathsf{now}(open) \vee \mathsf{pre}(\mathbf{P}\, open)$, where $\mathsf{now}(open)$ is *true* if *open* holds in the most recent event. An example of a first-order temporal specification is as follows.

$$\forall f\, (close(f) \rightarrow \mathbf{P}\, open(f)) \tag{2}$$

It asserts that every file that is closed was opened before. Here, we need to keep in the summary a *set* of all the opened files so that we can compare them to the closing of files. In general, the summary in this case extends the one used for the propositional case by keeping for each subformula the set of assignments, essentially a relation between the values assigned to the free variables that satisfy it: pre for the prefix without the last event, and now for the current prefix. These sets can be updated using database operations between relations, corresponding to the Boolean operations in the propositional case.

An extension of the logic, in another dimension, allows the properties to refer to the progress of time. The reported events appear with some integer timing value. We do not assume that the system reports to the monitoring program in each time unit or that only a single event occurs within a time unit. We also leave open the unit of measurement for time values (seconds, minutes, etc.). An example of such a specification is

$$\forall f\, (closed(f) \rightarrow \mathbf{P}_{\leq 20}\, open(f)) \tag{3}$$

which asserts that every file f that is closed was opened not longer than 20 time units before.

An RV algorithm for first-order LTL was presented in [7], and implemented in the MonPoly tool based on two alternative approaches: one allows unrestricted negation and in which the relations are represented as *regular sets* and, subsequently, automata [25]; another one with restricted negation and in which relations are represented explicitly and are subjected to database operators (e.g., join). In [7], an RV algorithm for first-order past temporal logic with time constraints was presented. In [19], an algorithm

that performs RV on first-order logic using BDDs was suggested and a related tool was constructed. BDDs are directed acyclic graphs that can often achieve a very compact representation of Boolean functions. In this context, a BDD represents the relationship between values of free variables that satisfy a given subformula in the summary. In that work, instead of representing the data values themselves, enumerations of these values were used. This allows a relatively short representation of big data values and using BDDs over a relatively small number of bits. It helps obtaining a good compactness for the BDDs due to common patterns in adjacent enumerations. The algorithm for the first-order logic is simple and quite similar to the propositional algorithm. Using a special reserved value to represent all the values that were not seen before allows the algorithm to easily deal with unconstrained negation.

In this work, we build upon this latter BDD-based construction and extend it to include in the temporal logic also timing constraints. This includes adapting the RV algorithm to reflect the timing constraints and extending the BDD representation to represent timing information as well as data values. We do this while keeping the summary compact and easy to update using BDD operations. We show how to perform updates on relations over both (enumerations of) data values and timing values, including Boolean and simple arithmetical operations. This is quite a nontrivial use of BDDs, applied to the context of runtime verification. Albeit the mixed use of the BDD representation and the addition of timing constraints, we manage to keep the basic algorithm similar to the propositional one. We follow the theory with an implementation that extends that of [19] and present experimental results.

Related Work. RV over propositional logic with timing constraints appears in [10, 33]. In [16], an RV algorithm for propositional LTL that returns optimal (minimal or maximal) values that make the specification correct with respect to the observed trace was presented. Other work on data-centric runtime verification include the systems based on trace slicing, where data values are mapped to copies of propositional automata [1, 29, 31], formula rewriting [5, 17], and rule-based monitoring [4, 6, 18], tree-automata [3] and SMT solving [13]. Applying arithmetic operations to sets of values, represented using BDD appeared in [14].

2 Propositional Past LTL with Timing

RV is often restricted to monitoring executions against specification properties that contain only the *past* modalities [27], where it is implicitly assumed that the specification needs to hold for *all* the prefixes of the execution[1]. These properties correspond to temporal *safety properties* [2], where a failure can always be detected on a finite prefix as soon as it occurs [10]. Expressing safety properties in this form allows an efficient runtime verification algorithm that is only polynomial in the size of the specification [23]. The syntax of propositional past timed linear temporal logic is as follows:

$$\varphi ::= \mathit{true} \mid p \mid (\varphi \wedge \varphi) \mid \neg\varphi \mid (\varphi \mathcal{S} \varphi) \mid (\varphi \mathcal{S}_{\leq \delta} \varphi) \mid (\varphi \mathcal{Z}_{\leq \delta} \varphi) \mid (\varphi \mathcal{S}_{> \delta} \varphi) \mid \ominus \varphi$$

[1] This is equivalent to saying that the specification is of the form $\Box\varphi$, where φ contains only past modalities; we omit here the implied $\Box$, which is a *future modality*.

where p is a *proposition* from a finite set of propositions P, with $\mathcal{S}$ standing for *since*, and $\ominus$ standing for *previous-time*. The formula $(\varphi \mathcal{S} \psi)$ has the standard interpretation that ψ must be true in the past and φ must be true since then. The formula $\ominus\varphi$ is true in the current state if φ is true in the previous state. The formula $(\varphi \mathcal{S}_{\leq\delta} \psi)$ has the same meaning as $(\varphi \mathcal{S} \psi)$, except that ψ must have occurred within δ time units. The formula $\varphi \mathcal{Z}_{\leq\delta} \psi$ is similar to $\varphi \mathcal{S}_{\leq\delta} \psi$, except that it requires ψ to be satisfied in the past; it is not sufficient if ψ is satisfied in the current state. Finally, $(\varphi \mathcal{S}_{>\delta} \psi)$ has the same meaning as $(\varphi \mathcal{S} \psi)$, except that ψ must have occurred more than δ time units ago. One can also write $(\varphi \vee \psi)$ instead of $\neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi)$ instead of $(\neg\varphi \vee \psi)$, $\mathbf{P}\,\varphi$ (*previous* φ) instead of $(true\ \mathcal{S}\ \varphi)$ and $\mathbf{H}\,\varphi$ (*history* φ) instead of $\neg\mathbf{P}\neg\varphi$. We also define $P_{\leq\delta}\varphi = (true \mathcal{S}_{\leq\delta} \varphi)$, $\mathbf{P}_{>\delta}\varphi = (true \mathcal{S}_{>\delta} \varphi)$, $\mathbf{H}_{\leq\delta}\varphi = \neg\mathbf{P}_{\leq\delta}\neg\varphi$, $\mathbf{H}_{>\delta}\varphi = \neg\mathbf{P}_{>\delta}\neg\varphi$, $(\varphi \mathcal{R}_{\leq\delta} \psi) = \neg(\neg\varphi \mathcal{S}_{\leq\delta} \neg\psi)$ and $(\varphi \mathcal{R}_{>\delta} \psi) = \neg(\neg\varphi \mathcal{S}_{>\delta} \neg\psi)$.

LTL formulas are interpreted over executions $\xi = \langle P, L, \tau \rangle$, where

- P is a finite set of *propositions*,
- $L : \mathbb{N} \mapsto 2^P$, where $\mathbb{N}$ are the positive integers,
- $\tau : \mathbb{N} \mapsto \mathbb{N}$ is a monotonic function (representing clock values). We may, but do not have to, assume that $\tau(1) = 0$.

We will refer to $\xi(i) = \langle i, L(i), \tau(i) \rangle$ as the i^{th} *event* in ξ, which satisfies the propositions $L(i)$ and occurs at time $\tau(i)$. LTL semantics is defined as follows:

- $\xi, i \models true$.
- $\xi, i \models p$ iff $p \in L[i]$.
- $\xi, i \models \neg\varphi$ iff not $\xi, i \models \varphi$.
- $\xi, i \models (\varphi \wedge \psi)$ iff $\xi, i \models \varphi$ and $\xi, i \models \psi$.
- $\xi, i \models (\varphi \mathcal{S} \psi)$ iff for some $1 < j \leq i$, $\xi, j \models \psi$, and for all $j < k \leq i$ it holds that $\xi, k \models \varphi$.
- $\xi, i \models (\varphi \mathcal{S}_{\leq\delta} \psi)$ iff there exists some $1 \leq j \leq i$, such that $\tau(i) - \tau(j) \leq \delta$ and $\xi, j \models \psi$, and for all $j < k \leq i$ it holds that $\xi, k \models \varphi$.
- $\xi, i \models (\varphi \mathcal{Z}_{\leq\delta} \psi)$ iff there exists some $1 \leq j < i$, such that $\tau(i) - \tau(j) \leq \delta$ and $\xi, j \models \psi$, and for all $j < k \leq i$ it holds that $\xi, k \models \varphi$.
- $\xi, i \models (\varphi \mathcal{S}_{>\delta} \psi)$ iff there exists some $1 \leq j < i$, such that $\tau(i) - \tau(j) > \delta$ and $\xi, j \models \psi$, and for all $j < k \leq i$ it holds that $\xi, k \models \varphi$.
- $\xi, i \models \ominus\varphi$ iff $i > 1$ and $\xi, i-1 \models \varphi$.

We say that an execution ξ satisfies a property φ iff for every i, it holds that $\xi, i \models \varphi$. Note that this is a *discrete time* semantics. We also do not require that every time instance must have a corresponding event. Thus, $(\varphi \mathcal{S} \psi)$ means that φ holds for every reported event since ψ held.

3 Runtime Verification for Propositional Past LTL

3.1 Algorithm for Propositional Past LTL without Time Constraints

The dynamic programming algorithm for propositional past LTL without timing constraints described in [23] 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. The algorithm operates on a summary that includes two vectors (arrays) of Boolean values indexed by subformulas: pre for the previous observed prefix, which excludes the last seen event, and now for the current prefix, which includes the last seen event. The algorithm is as follows.

1. Initially, for each subformula φ of the specification η, $\mathsf{now}(\varphi) := \mathit{false}$.
2. Observe the next event[2] $\langle i, L(i), \tau(i) \rangle$ 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 $\mathsf{now}(\psi)$.
   - $\mathsf{now}(p) := (p \in L(i))$.
   - $\mathsf{now}(\mathit{true}) := \mathit{true}$.
   - $\mathsf{now}((\varphi \wedge \psi)) := \mathsf{now}(\varphi) \wedge \mathsf{now}(\psi)$.
   - $\mathsf{now}(\neg\varphi) := \neg\mathsf{now}(\varphi)$.
   - $\mathsf{now}((\varphi \mathcal{S} \psi)) := \mathsf{now}(\psi) \vee (\mathsf{now}(\varphi) \wedge \mathsf{pre}((\varphi \mathcal{S} \psi)))$.
   - $\mathsf{now}(\ominus\, \varphi) := \mathsf{pre}(\varphi)$.
5. if $\mathsf{now}(\eta) = \mathit{false}$ then report "error".
6. Goto step 2.

3.2 RV for Propositional Past LTL with Timing Constraints

We describe the additions to the algorithm in Section 3.1 for the subformulas that contain timing constraints, i.e., $(\varphi \mathcal{S}_{\leq\delta} \psi)$, $(\varphi \mathcal{Z}_{\leq\delta} \psi)$ and $(\varphi \mathcal{S}_{>\delta} \psi)$. For each of these subformulas, we add to the summary two integer variables $\tau\mathsf{pre}$ and $\tau\mathsf{now}$, which represent timers that measure the time since a point that is relevant for calculating their truth value in the current state. These variables are initialized to -1 and their values will be updated based on the time difference $\Delta = \tau(i) - \tau(i-1)$ between the current event $\xi(i)$ and the previous one $\xi(i-1)$.

The propositional algorithm for $(\varphi \mathcal{S}_{\leq\delta} \psi)$

This subformula asserts that at position i in the trace, there is some earlier position j, where $\tau(e_i) - \tau(e_j) \leq \delta$ and where $(\varphi \mathcal{S} \psi)$ started to hold, until and including the current event. The summary needs to remember not only that ψ has happened and φ kept holding since, but also to update the time duration that has passed. There can be multiple such positions j where ψ held, but we only need to refer to the last (most recent) such position j, since it has the smallest value, hence also the time constraint will be the latest to expire.

The summary has the integer *time* variables $\tau\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi)$ and $\tau\mathsf{pre}(\varphi \mathcal{S}_{\leq\delta} \psi)$, which can have the values $[-1 \ldots \delta]$. This value is the distance from the most recent point where $(\varphi \mathcal{S} \psi)$ started to hold within an interval of δ time units. The values from $[0 \ldots \delta]$ correspond to $\mathsf{pre}/\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi) = \mathit{true}$ and -1 corresponds to $\mathsf{pre}/\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi) = \mathit{false}$. The update rule for $\tau\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi)$ and $\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi)$ is as follows:

[2] We ignore at this point the clock value component $\tau(i)$.

if $\mathsf{now}(\psi)$ **then** $\tau\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi) := 0$ [restart timer]
else if $\tau\mathsf{pre}(\varphi \mathcal{S}_{\leq\delta} \psi) \neq -1$ and $\mathsf{now}(\varphi)$ **then** [$(\varphi \mathcal{S}_{\leq\delta} \psi)$ continues to hold?]
  if $\tau\mathsf{pre}(\varphi \mathcal{S}_{\leq\delta} \psi) + \Delta > \delta$ **then** [distance too big?]
    $\tau\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi) := -1$ [$(\varphi \mathcal{S}_{\leq\delta} \psi)$ does not hold]
  else $\tau\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi) := \tau\mathsf{pre}(\varphi \mathcal{S}_{\leq\delta} \psi) + \Delta$ [update distance]
else $\tau\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi) := -1$; [$(\varphi \mathcal{S}_{\leq\delta} \psi)$ does not hold]
$\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi) := (\tau\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi) \neq -1)$

The propositional algorithm for $(\varphi \mathcal{Z}_{\leq\delta} \psi)$

This subformula is similar to $(\varphi \mathcal{S}_{\leq\delta} \psi)$, but requires that ψ has happened in the past, *excluding* the current time, and not more than δ time units in the past; if ψ holds now, this is not sufficient for $(\varphi \mathcal{Z}_{\leq\delta} \psi)$ to hold. This modality is required to express properties such as

$$\forall f \, open(f) \rightarrow \neg(true \, \mathcal{Z}_{\leq 20} \, open(f))$$

which asserts that we have not witnessed two openings of the same file in proximity of 20 ticks or less. Note that the previous-time $\ominus$ operator does not help in expressing the above property, since $\ominus$ refers to the previous event, which is not guaranteed to have occurred exactly one clock tick earlier. The algorithm sets the timer to the distance from the last event, if φ holds now, and ψ held in the previous event. Then it updates the timer by adding Δ as long as φ continues to hold and we are within the time distance δ.

if $\mathsf{now}(\varphi)$ **then**
  if $\mathsf{pre}(\psi)$ **and** $\Delta \leq \delta$ **then** $\tau\mathsf{now}(\varphi \mathcal{Z}_{\leq\delta} \psi) := \Delta$ [initiate timer]
  else
    if $\tau\mathsf{pre}(\varphi \mathcal{Z}_{\leq\delta} \psi) \neq -1$ **and** $\tau\mathsf{pre}(\varphi \mathcal{Z}_{\leq\delta} \psi) + \Delta \leq \delta$ **then** [distance still OK?]
      $\tau\mathsf{now}(\varphi \mathcal{Z}_{\leq\delta} \psi) := \tau\mathsf{pre}(\varphi \mathcal{Z}_{\leq\delta} \psi) + \Delta$ [update distance]
    else $\tau\mathsf{now}(\varphi \mathcal{Z}_{\leq\delta} \psi) := -1$
else $\tau\mathsf{now}(\varphi \mathcal{Z}_{\leq\delta} \psi) := -1$;
$\mathsf{now}(\varphi \mathcal{Z}_{\leq\delta} \psi) := (\tau\mathsf{now}(\varphi \mathcal{Z}_{\leq\delta} \psi) \neq -1)$

The propositional algorithm for $(\varphi \mathcal{S}_{>\delta} \psi)$

We update $\tau\mathsf{now}(\varphi \mathcal{S}_{>\delta} \psi)$, which is the current time distance to where $(\varphi \mathcal{S} \psi)$ (the untimed version of the subformula) started to hold. We update it according to the *earliest* (i.e., furthest in the past) occurrence where this held, since this is the larger distance, hence the first to satisfy the timing constraint. If this occurrence becomes irrelevant (since φ does not hold in the current prefix) then later observed occurrences become irrelevant too. When this happens, we either zero the counter, in case that ψ currently holds, or otherwide set it to -1 to signal that $(\varphi \mathcal{S} \psi)$ does not currently hold. We restrict the counter to $\delta + 1$; any value that is bigger than that will result in the same conclusion, and we want to keep that value small[3]. Now $\varphi \mathcal{S}_{>\delta} \psi$ currently holds when the value of this counter is bigger than δ.

[3] In fact, when $\Delta > \delta$, we use $\delta + 1$ instead.

if $\mathsf{now}(\varphi) \wedge \tau\mathsf{pre}(\varphi \mathcal{S}_{>\delta} \psi) \geq 0$ **then**
  $\tau\mathsf{now}(\varphi \mathcal{S}_{>\delta} \psi) := \min(\tau\mathsf{pre}(\varphi \mathcal{S}_{>\delta} \psi) + \Delta, \delta + 1)$
else if $\mathsf{now}(\psi)$ **then** $\tau\mathsf{now}(\varphi \mathcal{S}_{>\delta} \psi) := 0$ [restart counter]
else $\tau\mathsf{now}(\varphi \mathcal{S}_{>\delta} \psi) := -1$; [$(\varphi \mathcal{S}_{>\delta} \psi)$ does not hold]
$\mathsf{now}(\varphi \mathcal{S}_{>\delta} \psi) := (\tau\mathsf{pre}(\varphi \mathcal{S}_{>\delta} \psi) > \delta)$

4 First-Order Past LTL

First-order past LTL allows quantification over the values of variables that appear as parameters in the specification. In the context of RV, these values can appear within the monitored events. For example, $close(f)$ indicating that f is being closed. We saw in the introduction Property (2), which asserts that a file cannot be closed unless it was opened before. A more refined specification requires that a file can be closed only if it was opened before, but also has not been closed since:

$$\forall f \, (close(f) \longrightarrow \ominus(\neg close(f) \mathcal{S} \, open(f))) \tag{4}$$

An *assignment* over a set of variables W maps each variable $x \in W$ to a value from its associated domain. For example $[x \rightarrow 5, y \rightarrow$ "abc"$]$ is an assignment that maps x to 5 and y to "abc". A predicate consists of a predicate name and a variable or a constant of the same type[4]. E.g., if the predicate name p and the variable x are associated with the domain of strings, then p("gaga"), p("lady") and $p(x)$ are predicates. The predicates G with constant parameters are called *ground predicates*. A model, i.e., an execution (or a trace), ξ is a pair $\langle L, \tau \rangle$, where

1. $L : \mathbb{N} \mapsto 2^G$, and
2. $\tau : \mathbb{N} \mapsto \mathbb{N}$ is a monotonic function representing integer clock values.

An *event* in ξ is a triple $\xi(i) = \langle i, L(i), \tau(i) \rangle$ for $i \geq 1$.

4.1 Syntax

As in the propositional case, we restrict ourselves to *safety* properties, hence introduce only the past modalities.

$$\varphi ::= true \mid p(a) \mid p(x) \mid (\varphi \wedge \varphi) \mid \neg\varphi \mid (\varphi \mathcal{S} \varphi) \mid (\varphi \mathcal{S}_{\leq\delta} \varphi) \mid (\varphi \mathcal{Z}_{\leq\delta} \varphi) \mid (\varphi \mathcal{S}_{>\delta} \varphi) \mid \ominus\varphi \mid \exists x \, \varphi$$

We can also define $\forall x \varphi$ as $\neg\exists\neg\varphi$, and all the additional operators defined for the propositional case in Section 2.

4.2 Semantics

Let $free(\varphi)$ be the set of free, i.e., unquantified, variables of subformula φ. Let $\gamma[x \mapsto a]$ be an assignment that agrees with the assignment γ, except for the binding $x \mapsto a$. Then $\gamma, \xi, i \models \varphi$, where γ is an assignment that contains $free(\varphi)$, and $i \geq 1$, is defined as follows:

[4] For simplicity of the presentation, but without restricting the algorithms or the implementation, we present here only unary predicates.

- $\gamma, \xi, i \models \mathit{true}$.
- $\gamma, \xi, i \models p(a)$ if $p(a) \in L(i)$.
- $\gamma[x \mapsto a], \xi, i \models p(x)$ if $p(a) \in L[i]$.
- $\gamma, \xi, i \models (\varphi \wedge \psi)$ if $\gamma, \xi, i \models \varphi$ and $\gamma, \xi, i \models \psi$.
- $\gamma, \xi, i \models \neg\varphi$ if not $\gamma, \xi, i \models \varphi$.
- $\gamma, \xi, i \models (\varphi \mathcal{S} \psi)$ if there exists some $1 < j \leq i$, such that $\gamma, \xi, j \models \psi$ and and for all $j < k \leq i$ it holds that $\gamma, \xi, k \models \varphi$.
- $\gamma, \xi, i \models (\varphi\ \mathcal{S}_{\leq \delta}\ \psi)$ if there exists some $1 < j \leq i$, such that $\tau(i) - \tau(j) \leq \delta$ and $\gamma, \xi, j \models \psi$, and for all $j < k \leq i$ it holds that $\gamma, \xi, k \models \varphi$.
- $\gamma, \xi, i \models (\varphi \mathcal{Z}_{\leq \delta} \psi)$ iff there exists some $1 \leq j < i$, such that $\tau(i) - \tau(j) \leq \delta$ and $\gamma, \xi, j \models \psi$, and for all $j < k \leq i$ it holds that $\gamma, \xi, k \models \varphi$.
- $\gamma, \xi, i \models (\varphi \mathcal{S}_{> \delta} \psi)$ iff there exists some $1 \leq j \leq i$, such that $\tau(i) - \tau(j) > \delta$ and $\gamma, \xi, j \models \psi$, and for all $j < k \leq i$ it holds that $\gamma, \xi, k \models \varphi$.
- $\gamma, \xi, i \models \ominus\varphi$ if $i > 1$ and $\gamma, \xi, i-1 \models \varphi$.
- $\gamma, \xi, i \models \exists x\, \varphi$ if there exists $a \in \mathit{domain}(x)$ such that $\gamma[x \mapsto a], \xi, i \models \varphi$.

We write $\xi \models \varphi$ for a formula φ without free variables when $\varepsilon, \xi, i \models \varphi$ for each i, where ε is the empty assignment.

5 RV for First-Order Past LTL using BDDs

We describe an algorithm for monitoring first-order past LTL properties with time constraints. The untimed version and an implementation of it was presented in [19].

5.1 RV for First-order Past LTL without Time Constraints using BDDs

For the purpose of self containment, we first present the RV algorithm for the first-order past LTL without timing constraints, as presented in [19]. Then, in the next section we will show how to expand this into the logic with time constraints.

Using BDDs to represent relations

Our algorithm is based on representing relations between data elements (and, as we discuss later, timers, which are small integers) using Ordered Binary Decision Diagrams (OBDD, although we write simply BDD) [11]. A BDD is a compact representation for a Boolean valued function as a directed acyclic graph (DAG), see, e.g., Figures 1 and 2.

A BDD is obtained from a tree that represents a Boolean formula with some Boolean variables $x_1 \ldots x_k$ by gluing together isomorphic subtrees. Each non-leaf node is labeled with one of the Boolean variables. A non-leaf node x_i is the source of two arrows leading to other nodes. A dotted-line arrow represents that x_i has the Boolean value *false* (i.e., 0), while a thick-line arrow represents that it has the value *true* (i.e., 1). The nodes in the DAG have the same order along all paths from the root (hence the letter 'O' in OBDD). 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 *true* or *false*, corresponding to the Boolean value returned by the function for the Boolean values on the path.

A Boolean function, and consequently a BDD, can represent a set of integer values as follows. Each integer value is, in turn, represented using a bit vector: a vector of bits $x_1 \ldots x_k$ represents the integer value $x_1 \times 1 + x_2 \times 2 + \ldots x_k \times 2^k$, where the bit value of x_i is 1 for *true* and 0 for *false* and where x_1 is the *least* significant bit, and x_k is the *most* significant. For example, the integer 6 can be represented as the bit vector 110 (the most significant bit appears to the left) using the bits $x_1 = 0$, $x_2 = 1$ and $x_3 = 1$. To represent a *set* of integers, the BDD returns *true* for any combination of bits that represent an integer in the set. For example, to represent the set $\{4,6\}$, we first convert 4 and 6 into the bit vectors 100 and 110, respectively. The Boolean function over x_1, x_2, x_3 is $(\neg x_1 \wedge x_3)$, which returns *true* exactly for these two bit vector combinations.

This can be extended to represent relations, or, equivalently, a set of tuples over integers. The Boolean variables are partitioned into n bitstrings $x_1 = x_1^1, \ldots, x_{k_1}^1$, $x_n = x_1^n, \ldots, x_{k_n}^n$, each representing an integer number, forming the bit string[5]:

$$x_1^1, \ldots, x_{k_1}^1, \ldots, x_1^n, \ldots, x_{k_n}^n.$$

Using BDDs over enumerations of values

The summary for the first-order RV algorithm without timing constraints consists of BDDs $\mathsf{pre}(\varphi)$ and $\mathsf{now}(\varphi)$ for all subformulas of the monitored property. In the propositional case, these summary elements have Boolean values. For the first-order case, each summary element for a subformula φ is conceptually a relation between values of the free variables in φ. However, instead of representing these values directly, according to their different domains (e.g., integers, strings), these relations are represented as BDDs over the *enumerations* of values, and not directly over the values themselves.

During RV, when a value (associated with a variable in the specification) appears for the first time in an observed event, we assign to it a new *enumeration*. Values can be assigned consecutive enumeration values; however, a refined algorithm can reuse enumerations that were used for values that can no longer affect the verdict of the RV process, see [21]. We use a hash table to point from the value to its enumeration so that in subsequent appearances of this value the same enumeration will be used. For example, if the runtime verifier sees the input events *open*(a), *open*(b), *open*(c), it may encode them as the bit vectors 000, 001 and 010, respectively.

The described results in several advantages:

1. It allows a shorter representation of very big values in the BDDs; the values are compacted into a smaller number of bits. Furthermore, if a big data value occurs multiple times, we avoid representing that big value multiple times in the BDDs.
2. It contributes to the compactness of the BDDs because enumerations of values that are not far apart often share large bit patterns.
3. The first-order RV algorithm is simple and very similar to the propositional algorithm; the Boolean operators over summary elements: conjunction, disjunction and negation, are replaced by the same operators over BDDs. This also simplifies the implementation.

[5] In the implementation the same number of bits are used for all variables: $k_1 = k_2 = \ldots = k_n$.

4. Given an efficient BDD package, the implementation can be very efficient. On can also migrate between BDD packages.
5. Full use of negation also follows easily.

Example 1 - BDDs without time

As an example consider the following formula concerning the correctness of command execution. It states that for all commands m, if the command succeeds execution, then there must have been a dispatch of that command in the past with some priority p, and no failure since the dispatch:

$$\forall m(suc(m) \rightarrow \exists p(\neg fail(m)\ \mathcal{S}\ dis(m,p))) \tag{5}$$

Let us apply this property to the first two events of the following trace, where each event includes a single ground predicate. It consists of the dispatch of two commands, sending of telemetry data and success of the two commands:

$$\langle dis(stop,1), dis(off,2), tel(speed,2), suc(stop), suc(off)\rangle \tag{6}$$

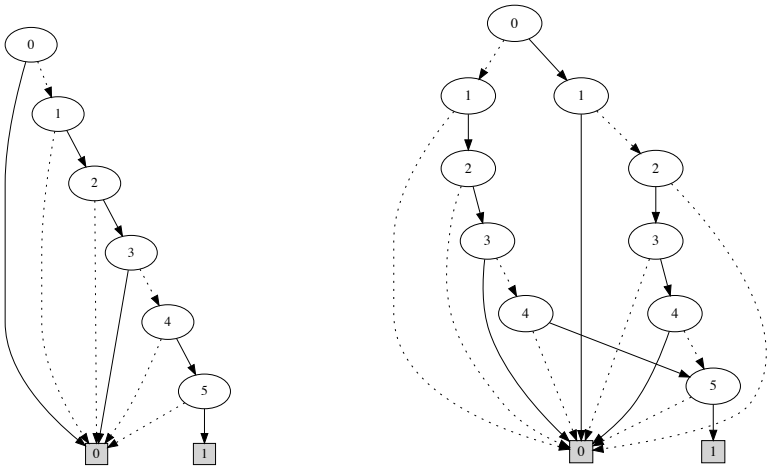
We shall now focus on the current assignments to the free variables m and p satisfying the subformula $\varphi = \neg fail(m)\ \mathcal{S}\ dis(m,p)$, represented as a BDD. After the first event $dis(stop,1)$ this BDD corresponds to the assignment $[m \mapsto stop, p \mapsto 1]$. The algorithm (to be shown below) will for each variable enumerate the data observed in events, in this case[6], assume that *stop* gets enumerated as 6 (binary 110) and 1 also gets enumerated as 6 (binary 110) (note that values for different variables get enumerated individually, and therefore can be mapped to the same enumerations). This mapping is recorded in the hash map for each variable from values to enumerations. Say we represent the enumeration for the value of each of the variables m and p using three bits: $m_1m_2m_3$ and $p_1p_2p_3$, with m_1 and p_1 being the least significant bits. The assignment $[m \mapsto stop, p \mapsto 1]$ will then be represented by a BDD which accepts the bit vector $m_1m_2m_3p_1p_2p_3 = 011011$. This BDD is shown in Figure 1a. The BDD has 6 nodes, named $0,\ldots,5$. The nodes 0, 1 and 2 represent $m_1m_2m_3$, and the nodes 3, 4 and 5 represent $p_1p_2p_3$. Following the arrows from node 0 on the top to the leaf node 1 (*true*) at the bottom, we indeed see the binary pattern 011011 (dotted-line arrows = 0 and thick-line arrows = 1).

Consider now the second event $dis(off,2)$. Here *off* gets enumerated as 5 (binary 101), just as 2 gets enumerated as 5 (binary 101) - again, variables get enumerated individually. The BDD in Figure 1b represents the set of assignments: $\{[m \mapsto stop, p \mapsto 1], [m \mapsto off, p \mapsto 2]\}$. The BDD is the union of the BDD in Figure 1a and a BDD representing the path 101101.

The BDD-based algorithm for first-order past LTL

We use a hash table to map values to their enumerations. When a ground predicate $p(a)$ occurs in the execution matching with $p(x)$ in the monitored property, the procedure

[6] The example BDDs are generated by our tool.



(a) BDD after first event. (b) BDD after second event.

Fig. 1: The BDDs for the formula $(\neg \mathit{fail}(m)\ \mathcal{S}\ \mathit{dis}(m,p))$ after the first event and after the second event.

lookup(x,a) is used to return the enumeration of a: it checks if a is already hashed. If not, i.e., this is a's first occurrence, then it will be hashed and assigned a new enumeration that will be returned by **lookup**. Otherwise, **lookup** returns the value hashed under a, which is the enumeration that a received before. A better compactness is achieved where each value is hashed separately for each variable x that matches it in the specification formula, hence **lookup**(x,a) is not necessarily the same as **lookup**(y,a).

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 (a bit vector). Note, however, that any enumeration scheme is possible, as shown in Example 1 above.

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_1, x_2 and x_3 for representing enumerations over x (with x_1 being the least significant bit), and assume that $A = \{a,b\}$, **lookup**$(x,a) = 001$, and **lookup**$(x,b) = 011$. Then **build**(x,A) is a BDD representation of the Boolean function $x_1 \wedge \neg x_3$.

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 for conjunction, disjunction and negation as $\bigwedge$, $\bigvee$ and $\neg$ (confusion should be avoided with the corresponding operations applying on propositions). To implement the existential (universal,

respectively) operators, we use the BDD existential (universal, respectively) operators over the 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 $\exists x_1,\ldots,x_k(B_\varphi)$ is the BDD that is obtained by applying the BDD existential quantification repeatedly on the BDD variables $x_1 \ldots,x_k$. Finally, BDD($\bot$) and BDD($\top$) are the BDDs that return uniformally *false* or *true*, respectively.

The dynamic programming algorithm, shown below, works similarly to the algorithm for the propositional case shown in Section 3. That is, it operates on two vectors (arrays) of values indexed by subformulas: pre for the state before the last event, and now for the current state after the last event. However, while in the propositional case the vectors contain Boolean values, in the first-order case they contain BDDs.

1. Initially, for each subformula φ of the specification η, $\mathsf{now}(\varphi) := \text{BDD}(\bot)$.
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 $\mathsf{now}(\psi)$.
   - $\mathsf{now}(\mathit{true}) := \text{BDD}(\top)$.
   - $\mathsf{now}(p(a)) :=$ if $p(a) \in s$ then BDD($\top$) else BDD($\bot$).
   - $\mathsf{now}(p(x)) :=$ **build**(x,A) where $A = \{a \mid p(a) \in s\}$.
   - $\mathsf{now}((\varphi \wedge \psi)) := \mathsf{now}(\varphi) \bigwedge \mathsf{now}(\psi)$.
   - $\mathsf{now}(\neg\varphi) := \neg\mathsf{now}(\varphi)$.
   - $\mathsf{now}((\varphi\ \mathcal{S}\ \psi)) := \mathsf{now}(\psi) \bigvee (\mathsf{now}(\varphi) \bigwedge \mathsf{pre}((\varphi \mathcal{S} \psi)))$.
   - $\mathsf{now}(\ominus\, \varphi) := \mathsf{pre}(\varphi)$.
   - $\mathsf{now}(\exists x\, \varphi) := \exists x_1,\ldots,x_k\, \mathsf{now}(\varphi)$.
5. if $\mathsf{now}(\eta) = \text{BDD}(\bot)$ then report "error".
6. Goto step 2.

An important component 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 events. We specifically reserve one enumeration, with bit vector value of $11\ldots11$ (i.e., all ones), to represent all values not seen yet. This trick allows us to use a finite representation and quantify existentially and universally over *all* values in infinite domains while allowing unrestricted use of negation in the temporal specification.

5.2 The BDD-based Algorithm for First-Order Past LTL with Time Constraints

We describe now *changes* to the algorithm in Section 5.1 for handling the subformulas with the timing constraints $(\varphi \mathcal{S}_{\leq\delta} \psi)$, $(\varphi \mathcal{Z}_{\leq\delta} \psi)$ and $(\varphi \mathcal{S}_{>\delta} \psi)$.

BDDs representing relations over data and time

Analogously to the propositional case, in the first-order case we need to add to the summary, for subformulas with timing constraints, in addition to the BDDs for $\mathsf{pre}(\varphi)$

and $\mathsf{now}(\varphi)$, also BDDs of the time $\tau\mathsf{pre}(\varphi)$ and $\tau\mathsf{now}(\varphi)$. These BDDs contain the relevant time that has passed that is needed in order to check the timing constraint.

Each assignment or tuple in such a BDD is over some number of data data variables $x^1 \ldots x^n$ and, in adddition, a timing variable t, forming the BDD bits:

$$x_1^1, \ldots, x_k^1, \ldots, x_1^n, \ldots, x_k^n, t_1, \ldots, t_m$$

These integer values are, either,

1. enumerations of data values, for each x^i, as explained above, or
2. the time t that has passed since the event that causes the tuple of data values to be included.

In order to keep the representation finite and small, $2\delta+1$ is used as the limit on t. That is, after we update t, we compare it against δ. When t goes beyond δ we can store just $\delta+1$ since we just need to know that it passed δ. During computation, when we observe a Δ that is bigger than δ, we cut it down to $\delta+1$ for the same reason, before we add to t. Finally, since adding $\Delta = \delta+1$ to a $t \leq \delta$ gives max $2\delta+1$, then this is the biggest number we need to store in a BDD. Consequently, the number of bits needed to store time is $log_2(2\delta+1)$.

Example 2 - BDDs with time

We add a timing constraint to the formula (5) in Example 1, stating that when a command succeeds it must have been dispatched in the past within 3 time units:

$$\forall m(suc(m) \rightarrow \exists p(\neg fail(m)\ \mathcal{S}_{\leq 3}\ dis(m,p))) \tag{7}$$

Let us apply this property to the first two events of the following trace, which is the trace (6) from Example 1, augmented with clock values following @-signs. We keep the time constraint and clock values small and consecutive, to keep the BDD small for presentation purposes:

$$\langle dis(stop,1)@1, dis(off,2)@2, tel(speed,2)@3, suc(stop)@4, suc(off)@5\rangle \tag{8}$$

The BDD for the subformula $\varphi = \neg fail(m)\ \mathcal{S}_{\leq 3}\ dis(m,p)$ at the third event $tel(speed,2)$, shown in Figure 2, reflects that two (010 in binary) time units have passed since $dis(stop,1)$ occurred, and one time unit (001 in binary) has passed since $dis(off,2)$ has occurred. The BDD is effectively an augmentation of the BDD in Figure 1b, with the additional three nodes 6, 7, and 8, representing respectively the bits t_1, t_2, and t_3 for the timer value, with t_1 (node 6) being the least significant bit.

BDD update operators on relations over data and time constraints

When a new event occurs, depending on the type of the subformula with timing constraint, we need to update the timers in $\tau\mathsf{now}$ that count the time that has passed since a tuple of values has entered. Subsequently, $\tau\mathsf{pre}$ will be updated when the next event will occur. The difference between the clock value of the current event and the clock

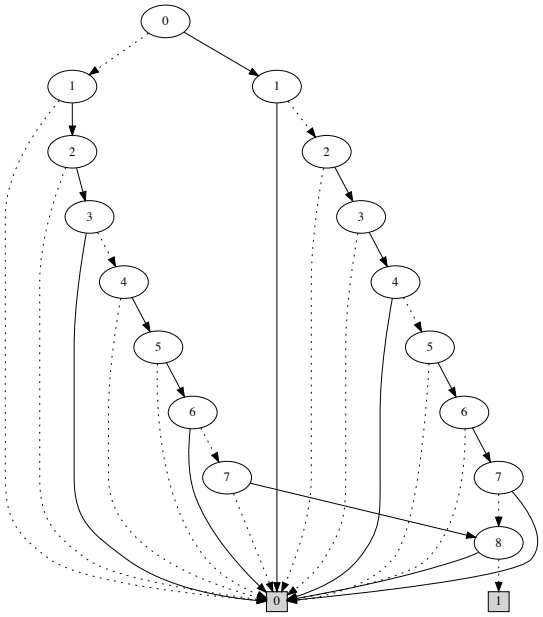


Fig. 2: The BDD for the formula $(\neg \mathit{fail}(m)\ \mathcal{S}_{\leq 3}\ \mathit{dis}(m,p))$ at the third event.

value of the previous one is Δ, and the timer is incremented, as explained above. by $\mathit{min}(\Delta, \delta+1)$. We also need to be able to check whether after adding Δ, the value of the time difference exceeds the time constraint δ.

Abstractly, given a relation R over data elements and time values, we need to construct two relations[7]:

- $R+\Delta = \{(x^1,\ldots,x^n,t+\Delta) | (x^1,\ldots,x^n,t) \in R\}$. This can be done by
  1. Constructing a relation $\mathcal{T} = \{(t,t') \,|\, t \geq 0 \wedge t' = t+\Delta\}$.
  2. Taking the *join* of R and $\mathcal{T}$. The join is basically the tuples that agree on the values of their common variables.
  3. Projecting out the (old) t values, and then renaming the (new) t' values as t.
- $R > \delta = \{(x^1,\ldots,x^n,t) \in R \,|\, t > \delta\}$. This can by done by
  1. Constructing $T_\delta = \{t \,|\, t > \delta\}$.
  2. Taking the join between R and T_δ.

We show now how to translate these set operators into BDDs. For $R+\Delta$, we construct a Boolean formula $\mathit{addconst}(t,t',\Delta)$ that expresses relation $\mathcal{T}$ between the Boolean variables of t and t'. For $R > \delta$, we construct a Boolean formula $\mathit{gtconst}(t,\delta)$ that corresponds to T_δ. These formulas are translated to BDDs. Then, taking the join of two BDDs is done by first completing the two BDDs to be over the same bits; since the BDDs are

[7] Recall that all values are restricted to $2\delta+1$ and if $\Delta > \delta$, then $\delta+1$ is used instead of Δ.

independent of the missing bits, this is trivial, keeping the same BDD structure. Then the intersection between these BDDs is obtained via the BDD conjunction ($\bigwedge$) operator.

The Boolean formula *addconst***.** The Boolean formula $addconst(t,t',\Delta)$ is satisfied by a pair of integer values t and t', represented as the bit vectors $t_1 \dots t_m$ and $t'_1,\dots,t'_m$, respectively, when $t' = t + \Delta$. The integer constant Δ is represented using the bit vector $\Delta_1 \dots \Delta_m$. The formula uses the additional bits $r_1,\dots,r_m$, where r_i is the carry-over from the i^{th} bits, according to Binary addition. This allows presenting the formula in an intuitive way, following standard binary addition and in obtaining a formula that is linear in the number of bits. When translating the formula to a BDD, existential quantification is applied to remove the Boolean variables $r_1,\dots,r_m$.

$addconst(t,t',\Delta) = \wedge_{1\le i\le m}\,(t'_i \leftrightarrow (t_i \oplus \Delta_i \oplus r_i))$
where $r_1 = false$,
for $1 \le i < m$: $r_{i+1} = ((r_i \wedge (t_i \vee \Delta_i)) \vee (\neg r_i \wedge t_i \wedge \Delta_i)))$

The formula *gtconst***.** The formula $gtconst(t,\delta)$ is *true* when t is bigger than δ. Both t and δ are integers represented as bit vectors $t_1 \dots t_m$ and $\delta_1 \dots \delta_m$, respectively. This holds when there is an index $1 \le i \le m$ such that $t_i = 1$ (*true*) and $\delta_i = 0$ (*false*), and where for $m \ge j > i$, $t_j = \delta_j$. When translating the formula to a BDD, existential quantification is applied to the Boolean variables $r_0,\dots,r_m$, which are used to propagate the check from the least to the most significant bit.

$gtconst(t,\delta) = r_m$
where $r_0 = false$,
for $1 \le i \le m$: $r_i = ((t_i \wedge \neg\delta_i) \vee ((t_i \leftrightarrow \delta_i) \wedge r_{i-1}))$

We describe now the additions required in Step 4 of the algorithm presented in Section 5.1.

The first-order algorithm for $(\varphi\ \mathcal{S}_{\le\delta}\ \psi)$

The BDDs $\mathsf{pre}/\mathsf{now}(\varphi\mathcal{S}_{\le\delta}\psi)$ generalize the Boolean summaries for the propositional past LTL, by representing enumerations of the values of the free variables that satisfy this subformula, e.g., with the bits $x_1^1,\dots,x_k^n$. The BDDs $\tau\mathsf{pre}/\tau\mathsf{now}(\varphi\mathcal{S}_{\le\delta}\psi)$ relate the values of the free variables that satisfy this subformula with the timer values that keep the time elapsed since the point where the values of the free variables were observed.

Generalizing from the propositional case, we need to compare and update timing values *per each assignment* to the free variables of a subformula $(\varphi\mathcal{S}_{\le\delta}\psi)$. As an example, the assignments (tuples) $\{[x \mapsto me, y \mapsto 72, t \mapsto 6],]x \mapsto you, y \mapsto 62, t \mapsto 9]\}$ for the subformula $(\varphi\mathcal{S}_{\le\delta}\psi)$, where t is assigned to the time units that has elapsed. We represent that using BDDs, where the values for x and y follow the previous conventions, with the bits $x_1 \dots x_k$ and $y_1 \dots y_k$ encoding the enumerations for the values for x and y, respectively, and the bits $t_1,\dots,t_m$ that represent the time passed since their introduction.

We will also use the following BDD constructions: $rename(B,x,y)$ renames the bits $x_1 \dots x_k$ in the BDD B as $y_1 \dots y_k$ and $\mathrm{BDD0}(x)$ is a BDD where all the x_i bits are a constant 0, representing the Boolean expression $\neg x_1 \wedge \dots \wedge \neg x_k$.

The update of the BDD $\tau\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi)$ is similar to the updates of the *if* statements in the propositional case, applied to all the values of the free variables of this subformula and uses the BDD constructed from the formula *gtconst*. While in the propositional case we kept the values $[-1, 0, \ldots, \delta]$, with -1 representing *false*, here we need only keep the assignments for the free variables of the subformula that correspond to $[0 \ldots \delta]$. Tuples of variable values that do not satisfy the time constraint are simply not represented by the BDD. This simplifies the formalization.

$$\begin{aligned}
&\tau\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi) := (\mathsf{now}(\psi) \bigwedge \mathrm{BDD0}(t)) \bigvee (\neg\mathsf{now}(\psi) \bigwedge \mathsf{now}(\varphi) \bigwedge \\
&\quad \mathit{rename}(\exists t_1 \ldots t_m \, (\mathit{addconst}(t, t', \Delta) \bigwedge \neg \mathit{gtconst}(t', \delta) \bigwedge \tau\mathsf{pre}(\varphi \mathcal{S}_{\leq\delta} \psi)), t', t)) \; ; \\
&\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi) := \exists t_1 \ldots t_m \, \tau\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi)
\end{aligned}$$

That is, either ψ holds now and we reset the timer t to 0, or ψ does not hold now but φ does, and the previous t value is determined by $\tau\mathsf{pre}(\varphi \mathcal{S}_{\leq\delta} \psi))$, to which we add Δ, giving t', which must not be greater than δ. Then t is removed by quantifying over it, and t' renamed to t (t' becomes the new t). The BDD for $\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi)$ is obtained from $\tau\mathsf{now}(\varphi \mathcal{S}_{\leq\delta} \psi)$ by projecting out the timer value.

Note that the Boolean operators $\bigwedge$ and $\bigvee$ on BDDs represent *join* and *cojoin*, respectively. This means that before the operator is applied, its two parameters are extended to have the same BDD variable bits (where the missing bits are assigned to all possible combinations).

The first-order algorithm for $(\varphi \; \mathcal{Z}_{\leq\delta} \; \psi)$

The update of the BDD $\mathsf{now}(\varphi \mathcal{Z}_{\leq\delta} \psi)$ is, conceptually, similar case-wise to the updates of the *if* statements of the propositional case, applied to all the values of the free variables of this formula.

$$\begin{aligned}
&\tau\mathsf{now}(\varphi \mathcal{Z}_{\leq\delta} \psi) := \\
&\quad \mathsf{now}(\varphi) \bigwedge \\
&\qquad ((\mathsf{pre}(\psi) \wedge \Delta \leq \delta \wedge \mathit{EQUAL}(t, \Delta)) \\
&\qquad\quad \bigvee \\
&\qquad (\neg\mathsf{pre}(\psi) \bigwedge \\
&\qquad\quad \mathit{rename}(\exists t_1 \ldots t_m \, (\mathit{addconst}(t, t', \Delta) \bigwedge \neg \mathit{gtconst}(t', \delta) \bigwedge \tau\mathsf{pre}(\varphi \mathcal{Z}_{\leq\delta} \psi)), t', t))); \\
&\mathsf{now}(\varphi \mathcal{Z}_{\leq\delta} \psi) := \exists t_1 \ldots t_m \, \tau\mathsf{now}(\varphi \mathcal{Z}_{\leq\delta} \psi)
\end{aligned}$$

Where $\mathit{EQUAL}(t, c) = \exists z_1 \ldots z_m \, (\mathrm{BDD0}(z) \bigwedge \mathit{addconst}(z, \; t, c))$, expressing that t is equal to c by adding $z = 0$ to c to obtain t. The formula says that φ must hold now and one of two cases must hold. In the first case, ψ holds in the previous state, $\Delta \leq \delta$, and t is initialized to Δ. In the second case, ψ does not hold in the previous state, and (using the same procedure as for the previous subformula) the previous t value is determined by $\tau\mathsf{pre}(\varphi \mathcal{Z}_{\leq\delta} \psi)$, to which we add Δ, giving t', which must not be greater than δ. Then t is removed by quantifying over it, and t' renamed to t (t' becomes the new t). Note that $\Delta \leq \delta$ is a Boolean condition, and, depending on its value, can be translated into the BDD representing the constants *true* or *false*.

The first-order algorithm for $(\varphi \mathcal{S}_{>\delta} \psi)$

Monitoring the subformula $(\varphi \mathcal{S}_{>\delta} \psi)$ is, conceptually, similar case-wise to the propositional case.

$\tau\mathsf{now}(\varphi \mathcal{S}_{>\delta} \psi) :=$
$\quad (\mathsf{now}(\psi) \bigwedge (\neg \mathsf{pre}(\varphi \mathcal{S}_{>\delta} \psi) \bigvee \neg \mathsf{now}(\varphi)) \bigwedge \mathrm{BDD0}(t)) \bigvee$
$\quad (\mathsf{now}(\varphi) \bigwedge \mathit{rename}(\mathit{previous}, t', t))$
where $\mathit{previous} = \exists t_1 \ldots t_m (\tau\mathsf{pre}(\varphi \mathcal{S}_{>\delta} \psi) \bigwedge ((\neg \mathit{gtconst}(t, \delta) \bigwedge \mathit{addconst}(t, t', \Delta)) \bigvee$
$\quad (\mathit{gtconst}(t, \delta) \bigwedge \mathit{EQUAL}(t', \delta + 1))));$
$\mathsf{now}(\varphi \mathcal{S}_{>\delta} \psi) := \exists t_1 \ldots t_m (\tau\mathsf{now}(\varphi \mathcal{S}_{>\delta} \psi) \bigwedge \mathit{gtconst}(t, \delta))$

When ψ currently holds and either $\varphi \mathcal{S}_{>\delta} \psi$ did not hold in the previous state or φ does not hold now, we reset the timer t to 0. When φ holds we compute t' using the **where**-clause as follows and then rename it to t: t takes its value from $\tau\mathsf{pre}(\varphi \mathcal{S}_{>\delta} \psi)$, which is calculated based on the previous step. This means that $(\varphi \mathcal{S}_{>\delta} \psi)$ held in the previous step. If t was then not greater than δ, we add Δ to t to obtain t'. Otherwise (t was already greater than δ), we set t' to $\delta + 1$ to reduce the size of the time values we have to store.

6 Implementation and Evaluation

6.1 Implementation

The DEJAVU tool, previously presented in [19] for the untimed case, was extended to capture the extension of the first-order LTL logic with time. The DEJAVU tool assumes that each state contains one[8] ground predicate, called an *event*. The tool, programmed in Scala, reads a specification containing one or more properties, and generates a Scala program, which can be applied to a log file containing events[9] in CSV (Comma Separated Value) format. The generated monitor program produces a verdict (true or false) for each event in the log, although only failures are reported to the user. It uses the JavaBDD package [24] for generating and operating BDDs. As an example, consider the property (7) from Example 2. The generated monitor uses an enumeration of the subformulas of the original formula in order to evaluate the subformulas bottom up for each new event. Figure 3 (right) shows the decomposition of the formula into subformulas (an Abstract Syntax Tree - AST), indexed by numbers from 0 to 8, satisfying the invariant that if a formula φ_1 is a subformula of a formula φ_2 then φ_1's index is bigger than φ_2's index. The evaluation function of the generated monitor ($\sim$ 900 LOC in total), which is applied for each event, is shown in Figure 3 (left). In each step the evaluate function re-computes the now array from highest to lowest index, and returns true (ok) iff now(0) is not BDD($\bot$).

[8] This restriction from the theory and algorithm presented above is made because our experience shows that this is by far the most common case.

[9] The tool can also be applied for online monitoring with some small adjustments.

```
def evaluate(): Boolean = {
  now(8) = build("dis")(V("m"),V("p"))
  now(7) = build("fail")(V("m"))
  now(6) = now(7).not()
  now(5) = (now(8).and(zeroTime)).or(
    now(6).and(pre(5))
    .and(DeltaBDD).and(deltaBDD)
    .and(addConst(t,tp,D,c))
    .and(gtConst(tp,d).not())
    .exist(var_d).exist(var_D)
    .exist(var_c).exist(var_t)
    .replace(tp_to_t_map)
  )
  now(4) = now(5).exist(var_t)
  now(3) = now(4).exist(var_p)
  now(2) = build("suc")(V("m"))
  now(1) = now(2).not().or(now(3))
  now(0) = now(1).forAll(var_m)
  val error = now(0).isZero
  tmp = now; now = pre; pre = tmp
  !error
}
```

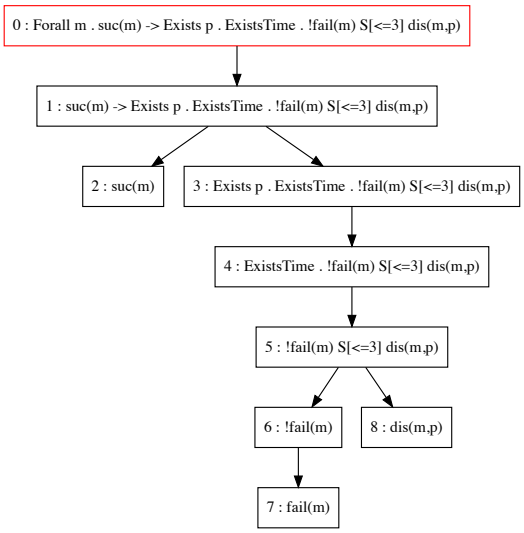


Fig. 3: Monitor (left) and AST (right) for the property.

6.2 Evaluation

We have performed an evaluation of DEJAVU by verifying variants of the properties shown in Figure 4 on a set of traces of varying length and structure. Each property is verified with, and without, time constraints. The *command* property is the previously discussed property (7) in Example 2. The *access* property is similar to a property evaluated in [19]. It states that if a file f is accessed by a user u, then the user should have logged in within 50 time units and not yet logged out, and the file should have been opened within 50 time units and not yet closed. The next properties concern operations of the Mars rover Curiosity [28]. The *boots* property concerns booting of instruments (passed as event parameters). A boot is initiated by a boot-start and terminated by a boot-end. The property states that for any instrument, we do not want to see a double boot (a boot followed by a boot), where the boots last longer than 20 seconds, and where the distance between the boots is less than 5 seconds. Finally, the *mobraces* and *armraces* properties follow the same pattern but for two different constants. The *mobraces* property states that during the execution of the command MOB_PRM (a dispatch of the command followed by the success of the command), which reports mobility parameters to ground; there should be no error in radio transmission of telemetry to ground. In addition the command must succeed in no more than 5 seconds. The *armraces* property states the same for the ARM_PRM command that transmits robotic arm parameters to ground. These two last properties in fact reflect a known (benign) race condition in

the software of the Curiosity rover, caused when a thread servicing the radio is starved and generates the warning tr_err which indicates missing telemetry. This happens because the thread is preempted by higher priority threads that are processing one of two commands MOB _PRM and ARM_PRM.

```
prop commands : Forall m . suc(m) → Exists p . ! fail (m) S[<=50] dis(m,p)

prop access : Forall u . Forall f .
  access(u,f) → ((! logout(u) S[<=50] login(u)) & (! close(f) S[<=50] open(f)))

prop boots: ! Exists i . (boot_e(i) & !P[<=20] boot_s(i) &
  @ (!boot_e(i) S (boot_s(i) & (!boot_e(i) S[<=5] (boot_e(i) &
  !P[<=20] boot_s(i) & @ (!boot_e(i) S boot_s(i)))))))

prop mobraces : suc("MOB_PRM") → (P[<=5] dis("MOB_PRM") &
  @ (!( suc("MOB_PRM") | Exists msg . tr_err(msg)) S dis("MOB_PRM")))

prop armraces : suc("ARM_PRM") → (P[<=5] dis("ARM_PRM") &
  @ (!( suc("ARM_PRM") | Exists msg . tr_err(msg)) S dis("ARM_PRM")))
```

Fig. 4: Evaluation properties.

Table 1 shows the results of the evaluation, performed on a Mac Pro laptop, running the Mac OS X 10.14.6 operating system, with a 2.9 GHz Intel Core i9 processor and 32 GB of memory. Each property is evaluated on one or more traces, numbered 1-15. Six of these traces are taken from [19] (traces nr. 1, 2, 3 and 7, 8, 9), and which are very data heavy, requiring lots of data to be stored by the monitor. The remaining traces require storing less information (and perhaps are more realistic). Traces 1-13 were generated for the experiment and are artificial, stress testing DEJAVU. Traces 14 and 15 are real logs of events reported by the Mars Curiosity rover, transmitted to JPL's ground operations (trace 14 is a prefix of the longer trace 15). For each trace is shown length in number of events, depth in terms of how many data values must be stored by the monitor, and whether it was verified without time constraints (no constr.) or with time constraints. A depth for the ACCESS property of e.g. 5,000 can mean that there at some point has been 5,000 users that have logged in and not yet logged out. Events in the logs 1-12 have consecutive clock values 1, 2, 3, ... Resulting trace analysis times are provided in minutes and seconds. In addition the factor of slowdown is shown for verifying with time constraints compared to verification without time constraints (execution time with constraints divided by execution time without constraints).

The interpretation of the results is as follows. By observing the factor numbers in the rightmost column, it is clear that there is a cost to monitoring timed properties compared

Property	Trace nr.	Trace length	Depth	Time constraint	Time	Factor
COMMANDS	1	11,004	8,000	no constr.	1.0s	
				50	1.8s	1.8
	2	110,004	80,000	no constr.	1.7s	
				50	13.2s	7.8
	3	1,100,004	800,000	no constr.	9.3s	
				50	2m5.8s	13.5
	4	10,050	25	no constr.	0.7s	
				50	1.0s	1.4
	5	100.050	25	no constr.	1.1s	
				50	1.8	1.6
	6	1,000,050	25	no constr.	2.6s	
				50	5.9s	2.3
ACCESS	7	11,006	5000	no constr.	0.9s	
				50	3.7s	4.1
	8	110,006	50,000	no constr.	2.2s	
				50	16.7s	7.6
	9	1,100,006	500,000	no constr.	15.2s	
				50	3m53.9s	15.4
	10	10.100	25	no constr.	0.8s	
				50	1.7s	2.1
	11	100,100	25	no constr.	1.1s	
				50	8.4s	7.6
	12	1,000,100	25	no constr.	2.6s	
				50	1m15.9s	29.2
BOOTS	13	10,012	low	no constr.	0.2s	
				2	0.4s	2.0
				20	0.8s	4.0
				50	5.1s	25.5
				60	7.2s	36.0
MOB + ARM RACES	14	50,000	low	no constr.	0.3s	
				10	0.7s	2.3
				60	1.0s	3.3
	15	96,795	low	no constr.	0.5s	
				10	1.0s	2.0
				60	1.6s	3.2

Table 1: Evaluation data. The factors (rightmost column) show how much slower verification of formulas with time constraints are compared to the untimed version of those formulas.

to monitoring properties without time constraints. This holds for all traces. Furthermore, the larger the time constraints, the more calculations the monitor has to perform on bit strings representing time values. The performance of DEJAVU is acceptable for time constraints that require no more than 7 bits of storage. We observed, however, that going beyond 7 bits causes the monitor execution to become considerably slower. This corresponds to time constraints beyond 63.5 (note that for a time constraint of δ one

needs $log_2(2\delta+1)$ bits, see page 13). The reason for this is not understood at the time of writing, and remains to be explored.

7 Conclusions

We extended the theory and implementation of runtime verification for first-order past (i.e., safety) temporal logic from [19] to include timing constraints. The untimed algorithm was based on representing relations over data values using BDDs. The use of BDDs over enumerations of the data values as integers, and subsequently, bit vectors, allowed an efficient representation that was shown, through an implementation and experiments, to allow the monitoring of large execution traces.

This was extended here to allow timing constraints, as in $(\varphi \mathcal{S}_{\leq\delta} \varphi)$, $(\varphi \mathcal{Z}_{\leq\delta} \varphi)$ and $(\varphi \mathcal{S}_{>\delta} \varphi)$, with each event in the input trace including an integer clock value. The addition of timing constraints was done by extending the BDDs to represent relations over both enumeration of data and timer values. This required the use of nontrivial operations over BDDs that allow updating relations while performing arithmetic operations on the timer values. We extended the tool DEJAVU, and reported on some of the experimental results performed with time constraints.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, Adding Trace Matching with Free Variables to AspectJ, OOPSLA'05, IEEE, 345-364, 2005.
2. B. Alpern, F. B. Schneider, Recognizing Safety and Liveness. Distributed Computing 2(3), 117-126, 1987.
3. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, Z. Manna: LOLA: Runtime Monitoring of Synchronous Systems, TIME'05, 166-174, 2005.
4. H. Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-Based Runtime Verification, VMCAI, LNCS Volume 2937, Springer, 44-57, 2004.
5. H. Barringer, K. Havelund, TraceContract: A Scala DSL for Trace Analysis, Proc. of the 17th International Symposium on Formal Methods (FM'11), LNCS Volume 6664, Springer, 57-72, 2011.
6. H. Barringer, D. Rydeheard, K. Havelund, Rule Systems for Run-Time Monitoring: from Eagle to RuleR, Proc. of the 7th Int. Workshop on Runtime Verification (RV'07), LNCS Volume 4839, Springer, 111-125, 2007.
7. D. A. Basin, F. Klaedtke, S. Müller, E. Zalinescu, Monitoring Metric First-Order Temporal Properties, Journal of the ACM 62(2), 1-45, 2015.
8. D. A. Basin, F. Klaedtke, E. Zalinescu, Algorithms for Monitoring Real-time Properties. Acta Informatica 55(4):, 309-338 (2018).
9. A. Bauer, M. Leucker, C. Schallhart, The Good, the Bad, and the Ugly, But How Ugly is Ugly?, RV'07, LNCS Volume 4839, Springer, 126-138, 2007.
10. A. Bauer, M. Leucker, C. Schallhart, Runtime verification for LTL and TLTL. ACM Trans. Software Engineering Methodologies, 20(4), 14:1-14:64, 2011.
11. R. E. Bryant, Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, ACM Comput. Surv. 24(3), 293-318, 1992.

12. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, Symbolic Model Checking: 10^{20} States and Beyond, LICS'90, 428-439, 1990.
13. N. Decker, M. Leucker, D. Thoma, Monitoring Modulo Theories, Journal of Software Tools for Technology Transfer, Volume 18, Number 2, 205-225, 2016.
14. E. M. Clarke, K. L. McMillan, X.g Zhao, M. Fujita, Jerry C.-Y. Yang, Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping. Formal Methods in System Design 10(2/3): 137-148 (1997).
15. Y. Falcone, J.-C. Fernandez, L. Mounier, Runtime Verification of Safety/Progress Properties, RV'09, LNCS Volume 5779, Springer, 40-59, 2009.
16. P. Faymonville, B. Finkbeiner, D. A. Peled, Monitoring Parametric Temporal Logic. VMCAI 2014, 357-375.
17. S. Hallé, R. Villemaire, Runtime Enforcement of Web Service Message Contracts with Data, IEEE Transactions on Services Computing, Volume 5 Number 2, 192-206, 2012.
18. K. Havelund, Rule-based Runtime Verification Revisited, Journal of Software Tools for Technology Transfer, Volume 17 Number 2, Springer, 2015.
19. K. Havelund, D. Peled, D. Ulus, First-order Temporal Logic Monitoring with BDDs, FM-CAD'17, IEEE, 116-123, 2017.
20. K. Havelund, D. Peled, Efficient Runtime Verification of First-Order Temporal SPIN 2018: 26-47.
21. K. Havelund, D. Peled, BDDs on the Run. ISoLA (4) 2018, Lymassol, Cyprus, 58-69.
22. K. Havelund, G. Reger, D. Thoma, E. Zălinescu, Monitoring Events that Carry Data, book chapter in: Lectures on Runtime Verification - Introductory and Advanced Topics, book editors: Ezio Bartocci and Yliès Falcone, LNCS Volume 10457, Springer, 61-102, 2018.
23. K. Havelund, G. Rosu, Synthesizing Monitors for Safety Properties, TACAS'02, LNCS Volume 2280, Springer, 342-356, 2002.
24. JavaBDD, http://javabdd.sourceforge.net.
25. J. G. Henriksen, J. L. Jensen, M. E. Jorgensen, N. Klarlund, R. Paige, T. Rauhe, A. Sandholm, Mona: Monadic Second-Order Logic in Practice, TACAS'95, LNCS Volume 1019, Springer, 89-110, 1995.
26. B. Könighofer, M. Alshiekh, R. Bloem, L. R. Humphrey, R. Könighofer, U. Topcu, C. Wang, Shield Synthesis, Formal Methods in System Design 51(2): 332-361 (2017).
27. Z. Manna, A. Pnueli, Completing the Temporal Picture, Theoretical Computer Science 83, 91-130, 1991.
28. Mars Science Laboratory (MSL) mission website: http://mars.jpl.nasa.gov/msl.
29. P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Rosu, An Overview of the MOP Runtime Verification Framework, STTT 14(3), Springer, 249-289, 2012.
30. D. Peled, K. Havelund, Refining the Safety–liveness Classification of Temporal Properties According to Monitorability, Submitted for publication, LNCS, Sept. 2018.
31. G. Reger, H. Cruz, D. Rydeheard, MarQ: Monitoring at Runtime with QEA, Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15), LNCS Volume 9035, Springer, 595-610, 2015.
32. G. Rosu, S. Bensalem, Allen Linear (Interval) Temporal Logic - Translation to LTL and Monitor Synthesis, CAV 2006, 263-277.
33. K. Y. Rozier, J. Schumann, R2U2: Tool Overview. RV-CuBES 2017, 138-156.