BDDs on the Run*

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

Abstract. Runtime verification (RV) of first-order temporal logic must handle a potentially large amount of data, accumulated during the monitoring of an execution. The DEJAVU RV system represents data elements and relations using BDDs. This achieves a compact representation, which allows monitoring long executions. However, the potentially unbounded, and frequently very large amounts of data values can, ultimately, limit the executions that can be monitored. We present an automatic method for "forgetting" data values when they no longer affect the RV verdict on an observed execution. We describe the algorithm and illustrate its operation through an example.

1 Introduction

Runtime verification (RV) can be used to check the execution (run) of a system against a temporal property, yielding an alarm when the property is violated, so that aversive action can be taken. For each consumed event the monitor performs incremental computation, updating its internal memory, and has to decide whether the property is violated based on the finite part of the execution trace that it has viewed so far. To inspect an execution, the monitored system is instrumented to report on occurrences of events.

Runtime verification is often applied to executions that consist of events that contain data values [1–7, 9–13, 16–18]. A large amount of different observed data values can pose a challenge to the efficiency of RV systems, in terms of time and space, since it is essential to keep up with rapid occurrence of events in very long executions. We present the DEJAVU system and its logic QTL (Quantified Temporal Logic), which in its core supports *past* temporal properties, including existential quantification, predicates with data values and variables, the Boolean operators *and*, *not*, and the modal operators $\ominus$ for previous-time and $\mathcal{S}$ for since. Several standard operators are derived from these.

In [14] we presented an early version of DEJAVU and its algorithm based on the use of BDDs. We describe here furthermore an approach for detecting when data elements that were seen so far do not affect the rest of the execution and can be discarded, also

* 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 ISF grant 2239/15: "Runtime Measuring and Checking of Cyber Physical Systems".

referred to as *dynamic data reclamation*. As an example, consider the following formula, asserting that data can be written to a file only if it has been opened in the past, and not closed since then.

$$\forall f\,((\exists d\, write(f,d)) \longrightarrow (\neg close(f)\,\mathcal{S}\, open(f))) \tag{1}$$

We can observe that if a file was opened and subsequently closed, then the property would be invalidated if a value is written to that file, just as in the case where the file was never opened. This means that we can "forget" that a file was opened when it is closed, without affecting our ability to monitor the formula. If there are no more than N files simultaneously opened at any time, then we need space for only N files for monitoring the property. This is in contrast to [14], where space for all new file names must be allocated. We present the algorithm for storing data as BDDs, and the automatic detection of data values that are not required anymore, reclaiming the space used for storing them.

The contents of the paper is as follows. Section 2 presents the syntax and semantics of QTL. Section 3 describes the basic BDD-based algorithm. Section 4 outlines the dynamic data reclamation approach. Section 5 illustrates the extended algorithm by executing a monitor on an example trace. Finally Section 6 concludes the paper.

2 Syntax and Semantics

Let X be a finite set of *variables*. An *assignment* over a set of variables $W \subseteq X$ maps each variable $x \in W$ to a value from its associated domain $domain(x)$. We assume that the domains (e.g., integers, strings) are infinite (see [14] for dealing with finite domains). For example $[x \to 5, y \to \text{"abc"}]$ maps x to 5 and y to "abc". Let T be a set of *predicate names*, where 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 $p(\text{"gaga"})$, and $p(x)$. Predicates over constants are called *ground predicates*. An *event* is a finite set of ground predicates. For example, if $T = \{p,q,r\}$, then $\{p(\text{"xyzzy"}), q(3)\}$ is a possible event. An *execution trace* $\sigma = s_1, s_2, \ldots$ is a finite sequence of events.

The formulas of the 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.

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

The formula $p(a)$, where a is a constant in $domain(p)$, means that the ground predicate $p(a)$ occurs in the most recent event. The formula $p(x)$, for a variable $x \in X$, holds with a binding of x to the value a if a ground predicate $p(a)$ appears in the most recent event. The formula $(\varphi\ \mathcal{S}\ \psi)$ means that ψ held in the past (possibly now) and since then φ has been true. The formula $\ominus\, \varphi$ means that φ is true in the previous event. We can furthermore define the following derived operators: $false = \neg true$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \to \psi) = (\neg\varphi \vee \psi)$, $\mathbf{P}\,\varphi = (true\,\mathcal{S}\,\varphi)$, $\mathbf{H}\,\varphi = \neg\mathbf{P}\,\neg\varphi$, and $\forall x\,\varphi = \neg\exists x\,\neg\varphi$.

Let $free(\varphi)$ be the set of free (i.e., unquantified) variables of a subformula φ. Let $I[\varphi,\sigma,i]$ be the semantic function, defined below. It returns the set of assignments that satisfy φ after the ith event of the execution σ. 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 $\{\varepsilon\}$ 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 assignments whose projection on the *common* variables agree with an assignment in A_1 and with an assignment in A_2. Union is defined as the operator dual of intersection. Let Γ be a set of assignments over a set of variables W; we denote by $hide(\Gamma,x)$ the set of assignments over $W \setminus \{x\}$, obtained from Γ by removing the assignment to x for each element of Γ. In particular, if Γ is a set of assignments over just the variable x, then $hide(\Gamma,x)$ is $\{\varepsilon\}$ when Γ is nonempty, and $\emptyset$ otherwise. $A_{free(\varphi)}$ 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 $i > 0$.

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

3 An Efficient Algorithm using BDDs

We describe here an algorithm for monitoring QTL, first presented in [14], and implemented as the first version of the tool DEJAVU. We shall represent a set of assignments as an Ordered Binary Decision Diagram (OBDD, although we write simply BDD) [8].

Recall that a BDD is a directed acyclic graph (DAG), where the non-leaf nodes represent Boolean variables. Figures 2 and 3 (to be explained) show BDDs over the BDD variables b_0, b_1, b_2, and b_3. A BDD is a compact representation of a Boolean formula over these variables, and can be used to determine, for a given assignment to the variables, whether the formula is true or not. Each non-leaf node is the source of two arrows leading to other nodes. A dotted-line arrow represents that the Boolean variable has the value 0 (false), while a thick-line arrow represents that it has the value 1 (true). 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 does not depend on the value of the corresponding Boolean variable. Each path leads to a leaf node that is marked by either 0 (false) or 1 (true), representing the Boolean value returned by the formula for the variable assignment corresponding to the followed path.

Assume that we see p("ab"), p("de"), p("af") and q("fg") in subsequent events in an execution 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. The least significant bit in an enumeration is represented by BDD variable b_0, and the most significant bit by the BDD variable with highest index. Using a three-bit enumeration $b_2b_1b_0$, the first encountered value "ab" can be represented[1] 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 $\neg b_2 \wedge (b_1 \leftrightarrow \neg b_0)$.

When representing a set of assignments for two variables x and y with k bits each, we use Boolean BDD variables $x_{k-1},\ldots,x_0,y_{k-1},\ldots y_0$. A BDD will return a 1 for each bit string consisting of the concatenation of enumerations that correspond to the represented assignments, and 0 otherwise. For example, to represent the assignment $[x \mapsto$ "de", $y \mapsto$ "af"], where "de" is enumerated as 001 and "af" with 010, the BDD will return a 1 for 001010. The BDD that returns always 0 is denoted by BDD($\bot$), and the BDD that returns always 1 is denoted by BDD($\top$).

Given a ground predicate $p(a)$, observed in the currently monitored event of the execution, then when 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. The function **build**(x,V), where V is a set of values, returns a BDD that represents the set of assignments where x is mapped to (the enumeration of) a for $a \in V$. This BDD is independent of the values assigned to any variable other than x, i.e., they can have any value.

The algorithm, shown below, operates on two vectors (arrays) of values indexed by subformulas (as in [15]): pre for the state before that event, and now for the current state (after the last seen event).

1. Initially, for each subformula φ, $\mathsf{now}(\varphi) := \text{BDD}(\bot)$.
2. Observe a new event (as 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,V) where $V = \{a \mid p(a) \in s\}$.
   - $\mathsf{now}((\varphi \wedge \psi)) :=$ **and**$(\mathsf{now}(\varphi), \mathsf{now}(\psi))$.
   - $\mathsf{now}(\neg\varphi) :=$ **not**$(\mathsf{now}(\varphi))$.
   - $\mathsf{now}((\varphi \mathcal{S} \psi)) :=$ **or**$(\mathsf{now}(\psi),$ **and**$(\mathsf{now}(\varphi), \mathsf{pre}((\varphi \mathcal{S} \psi))))$.

[1] Enumerations are here selected using a counter initialized to 0, as in [14]. The data reclamation solution in Sect. 4 instead uses a SAT solver.

- $\mathsf{now}(\ominus\,\varphi) := \mathsf{pre}(\varphi)$.
- $\mathsf{now}(\exists x\,\varphi) := \mathbf{exists}(\langle x_0,\ldots,x_{k-1}\rangle, \mathsf{now}(\varphi))$.

5. Goto step 2.

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. In particular, we save for that purpose the highest valued enumeration $11\ldots11$, which we denote by $\mathrm{BDD}(11\ldots11)$. This allows us to use a finite representation and quantify existentially and universally over *all* values in infinite domains.

4 Dynamic Data Reclamation

We now describe the possibility of reusing enumerations of data values, when this does not affect the decision whether the property holds or not. When a value a is *reclaimed*, its enumeration e can be reused for representing another value that appears later in the execution.

Recall that upon the occurrence of a new event, the basic algorithm uses the BDD $\mathsf{pre}(\psi)$, for any subformula ψ, representing assignments satisfying this subformula calculated based on the sequence monitored so far before the new event. Since these BDDs sufficiently summarize the information that will be used about the execution monitored so far, reclaiming data can be automated without user guidance or static formula analysis, solely based on the information the BDDs contain.

We are seeking a condition for reclaiming values of a variable x. 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.

As the pre BDDs use enumerations to represent values, we find the enumerations that can be reclaimed. Then, their corresponding values are removed from the hash table, and the enumerations can later be reused to represent new values. Recall that the enumeration $11\ldots11$ 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 betwen the enumeration of a and the enumeration $11\ldots11$. 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., $x_0,\ldots,x_{k-1}$, $y_0,\ldots,y_{k-1}$ and $z_0,\ldots,z_{k-1}$. 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 $11\ldots11$.

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

To take advantage of reclaimed enumerations, we represent a set of available enumerations for a variable x using a BDD $\mathit{avail}(x)$. Initially at the start of monitoring, we set

$avail(x) := \neg\text{BDD}(11\ldots11)$. Let $sub(\varphi)$ be the set of subformulas of the property φ. When we are short in the number of available enumerations and thus we want to perform data reclamation, we calculate $I_{\psi,x}$ for all the subformulas $\psi \in sub(\varphi)$ that contain x as a free variable, and set:

$$avail(x) := (\bigwedge_{\psi \in sub(\varphi),\ x \in free(\psi)} I_{\psi,x}) \wedge \neg\text{BDD}(11\ldots11)$$

This updates $avail(x)$ to denote all available enumerations, including reclaimed enumerations. When we need a new enumeration for variable x, we just pick some enumeration e that satisfies $avail(x)$. Let $BDD(e)$ denote a BDD that represents only the enumeration e. To remove that enumeration from $avail(x)$, we update $avail(x)$ as follows:

$$avail(x) := avail(x) \wedge \neg\text{BDD}(e)$$

The formula $I_{\psi,x}$ includes multiple quantifications (over the bits used to represent the free variables other than x). Therefore, it may not be efficient to reclaim enumerations too frequently. We can reclaim enumerations either periodically or when $avail(x)$ becomes empty or close to empty.

As the BDD-based algorithm detects which enumerations e can be reclaimed, we need to identify the related data value a and update the hash table, so that a will not point to e. In particular, we need to be able to find the data that is represented by a given enumeration. To do that, one can use a *trie*: in our case this will be a trie with at most two edges from each node, marked with either 0 or 1. Traversing the trie from its root node on edges labeled according to the enumeration e reaches a node that contains the value a that is enumerated as e. The traversing and updating the trie is linear per each enumeration. The current implementation, however, uses the simpler straightforward strategy of walking though all values and removing those which point to reclaimed enumerations.

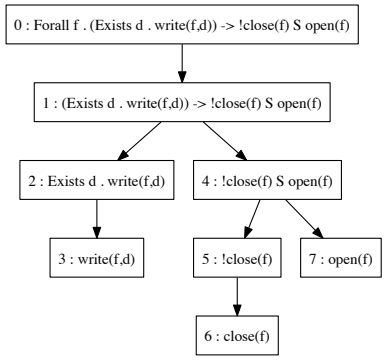
5 Example Monitor Execution

In this section we illustrate the working of the algorithm with a minimal, and yet complete, example. Specifically we execute the algorithm on formula (1) and the following trace consisting of nine events:

$\{open(\text{f1})\}, \{open(\text{f2})\}, \{open(\text{f3})\},$
$\{close(\text{f1})\}, \{close(\text{f2})\}, \{close(\text{f3})\},$
$\{open(\text{f1})\}, \{open(\text{f4})\}, \{write(\text{f4,2})\}$

The formula contains two variables f and d, and we use just two bits to represent each of these, yielding four possible bit combinations per variable: 00, 01, 10, and 11. The enumeration 11, however, as has been explained, is devoted to represent *values not seen yet* in the trace during monitoring, hence with two bits we can represent three values observed in the trace at a time.

To recall previous material, the algorithm in Sect. 3 updates for each new event the now array, updating entries for innermost formulas first. References are made to




```
override def evaluate(): Boolean = {
  now(7) = build("open")(V("f"))
  now(6) = build("close")(V("f"))
  now(5) = now(6).not()
  now(4) = now(7).or(now(5).and(pre(4)))
  now(3) = build("write")(V("f"),V("d"))
  now(2) = now(3).exist(var_d.quantvar)
  now(1) = now(2).not().or(now(4))
  now(0) = now(1).forAll(var_f.quantvar)
  val error = now(0).isZero
  tmp = now; now = pre; pre = tmp
  !error
}
```

Fig. 1: Numbering of subformulas and generated monitor for property (1).

the pre array when computing BDDs for subformulas containing a temporal operator at the outermost level, such as in this case the subformula $(\neg close(f)\,\mathcal{S}\,open(f))$. The *subformulas-first* principle is achieved by enumerating subformulas as shown in Fig. 1 (left) and use this enumeration to update the now array in the generated monitor code[2], as shown in Fig. 1 (right).

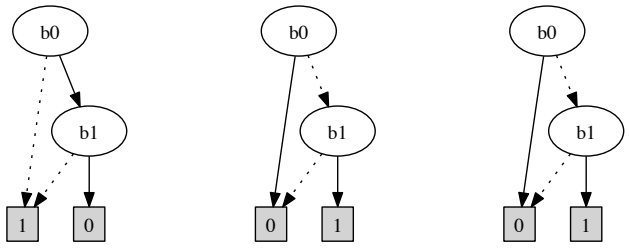
We illustrate now the BDDs generated for selected positions in the now array as the events in the above trace are submitted to the monitor. Figure 2 shows selected BDDs from monitoring the first six events, whereas Fig. 3 shows selected BDDs from monitoring the remaining three events. A BDD is either the denotation of $avail(f)$ (Sect. 4), or the contents of the now array at an index corresponding to a position in the subformula tree in Fig. 1 (left). The caption for each BDD identifies either $avail(f)$ or a subformula index, an @-sign, and the event that caused the computation of this BDD.

Recall that upon analysis of a new event, a data value in the event for a variable is mapped to one of the bit enumerations 00, 01, or 10 (in a hash table for that variable). The BDD denoted by a subformula (and stored in the now array at the appropriate index) for a single variable will represent a subset of these three enumerations, representing the set of values making the subformula true. The BDD for a variable has a unique *BDD variable* for each bit. In our case BDD variables b_0 and b_1 are used to represent the variable f, and BDD variables b_2 and b_3 are used to represent variable d. The monitoring of the trace above proceeds as follows.

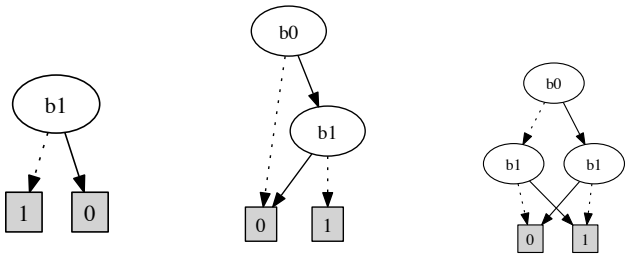
Initially: Figure 2a shows the BDD representing initially available enumerations for variable f ($avail(f)$). These are all enumerations different from 11 (namely 00, 01, and 10). The enumeration 11 is the reserved enumeration representing all values not yet seen, and is the only assignment leading to leaf-node 0 (follow the fully drawn arrows).

After event *open*(f1): Figure 2b shows the generated BDD for the enumeration 10 (note that the least significant rightmost bit in 10 corresponds to the BDD variable b_0 at

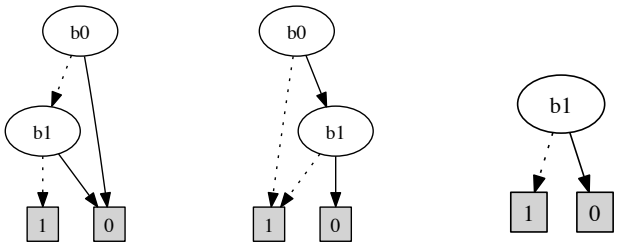
[2] An additional 600+ lines of, mostly property-independent, code is generated.



(a) *avail*(f) initially (b) node 7 @ *open*(f1) (c) node 4 @ *open*(f1)

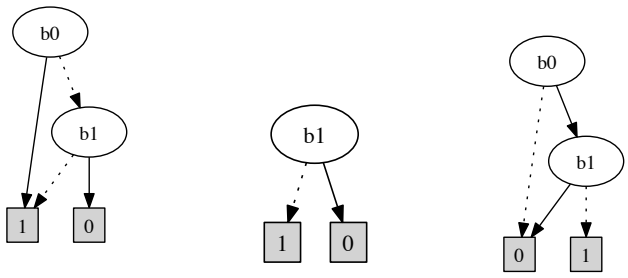


(d) *avail*(f) @ *open*(f1) (e) node 7 @ *open*(f2) (f) node 4: @ *open*(f2)

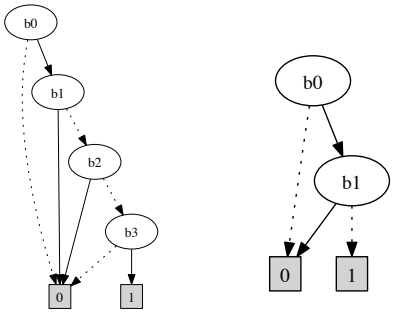


(g) node 7 @ *open*(f3) (h) node 4: @ *open*(f3) (i) node 4 @ *close*(f1)

Fig. 2: Selection of BDDs computed during monitoring of first six events.



(a) $I_{\psi,f}$ for node 4 @ *open*(f4) (b) $avail(f)$ @ *open*(f4) (c) node 7 @ *open*(f4)



(d) node 3 @ *write*(f4,2) (e) node 2 @ *write*(f4,2)

Fig. 3: Selection of BDDs computed during monitoring of remaining three events.

the top of the BDD). This enumeration represents file f1, and is picked from $avail(f)$ using a SAT solver. Figure 2c shows the BDD for subformula 4 $(\neg close(f) \mathcal{S} open(f))$. It represents the enumeration 10 for file f1, since this is the only file that has been opened so far and not closed yet, and this BDD is therefore the same as the one in Fig. 2b. Figure 2d shows the BDD denoted by $avail(f)$ thereafter, representing the set $\{00, 01\}$. Note that these are the enumerations where the leftmost (most signifiant) bit is 0, shown in the BDD as BDD variable b_1 having value 0 (the dashed line).

After event *open*(f2): Figure 2e shows the enumeration 01 allocated for file f2. $avail(f)$ is updated accordingly, subtracting 01 (now shown). Figure 2f shows the BDD for subformula 4, which now represents the set containing the two enumerations $\{10, 01\}$. This illustrates the core principle of representing a set of assignments as a BDD. This BDD is obtained by performing a BDD **or** (corresponding to a set union) on the BDDs for 10 respectively 01.

After event *open*(f3): Figure 2g shows the last available enumeration 00 allocated for file f3. $avail(f)$ is updated accordingly, subtracting 00, and now becomes BDD($\bot$)

(not shown), that returns 0 for all enumerations. Figure 2h shows the BDD for subformula 4, which now represents the set containing the three enumerations $\{10, 01, 00\}$, in other words: any enumeration except 11, which is the only enumeration leading to 0.

After event *close*(f1): Figure 2i shows the BDD for node 4 after removal of the enumeration 10 representing file f1, resulting in the set: $\{01, 00\}$, which contains all enumerations where BDD variable b_1 (representing the most significant bit) is 0.

After events *close*(f2) and *close*(f3): The subsequent closing of files f2 and f3 results in a situation where node 4 is BDD($\bot$), since all files now have been closed. Furthermore, $avail(f)$ is also still BDD($\bot$), meaning that the opening of a new file not yet seen will cause reclamation to be initiated.

After event *open*(f1): The re-opening of file f1 is possible without data reclamation (even though $avail(f)$ is BDD($\bot$)) since the former enumeration 10 associated with f1 is still recorded in the hash table and is therefore reused. This leads to a BDD for node 4 that is the same as in Fig. 2c. $avail(f)$ remains BDD($\bot$).

After event *open*(f4): The opening of file f4 causes data reclamation since there are no more available enumerations: $avail(f)$ is BDD($\bot$). Recall from Sect. 4 that the new value of $avail(f)$ is computed by computing the BDD $I_{\psi,f}$ of available enumerations for variable f for each subformula ψ, and **and**-ing them together with **not**(BDD(11)). We only need to compute these contributions for temporal formulas. Figure 3a shows the BDD $I_{\psi,f}$ for $\psi = (\neg close(f) \mathcal{S} open(f))$. Since file f1 was re-opened, and thereby its enumeration 10 reused, the irrelevant enumerations $I_{\psi,f}$ stemming from this subformula are all the enumerations (including the special value 11, which will be subtracted later) that are different from 10, which here is the only enumeration leading to a 0-leaf. Figure 3b shows the value of $avail(f)$ after these computations have been performed, resulting in the BDD representing all enumerations different from 10 and 11. These are the enumerations $\{01, 00\}$ (BDD variable b_1, representing the most significant bit, is 0), which now can be allocated again for new data. Specifically, Fig. 3c shows the BDD for the allocation of enumeration 01 for the new file f4.

After event *write*(f4,2): Above we have seen examples of how a set of assignments to a single variable (f) is represented as a BDD. The writing of the datum 2 to file f4 illustrates how assignments to multiple variables, in this case f and d, are represented. Writing 2 to file f4 invokes the just allocated enumeration 01 for f4 and a new enumeration 10 for d (same procedure as for f). Figure 3d shows the BDD representing the juxtaposition of these two enumerations. BDD variables b_0 and b_1 (representing f) denote the enumeration 01, and BDD variables b_2 and b_3 (representing d) denote the enumeration 10. This BDD therefore represents the assignment $[f \mapsto \text{f4}, d \mapsto 2]$. Finally, Fig. 3e shows the BDD in node 2 of Fig. 1 after performing existential quantification over d on the 4-variable BDD in Figure 3d. The result is obtained by removing BDD variables b_2 and b_3, and repointing BDD variable b_2's incoming arrow to leaf-node 1.

6 Conclusion

We described a BDD-based algorithm for monitoring executions of a system against first-order past time temporal logic properties. The algorithm supports automated dynamic data reclamation, removing data values when they no longer affect the verdict.

The BDD data structure appears to offer advantages for runtime verification w.r.t. efficiency of monitoring, but also w.r.t. expressiveness of the logic. Although not discussed in this paper, DEJAVU supports numerical relations between variables, and, in addition to quantification over all possible values in an infinite domain, also quantification only over values seen in the trace. Future work includes support for real-time constraints, and functions applied to data values.

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, SIGPLAN Not. 40(10), 345-364, ACM, 2005.
2. 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, IEEE Computer Society, 166-174, 2005.
3. H. Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-Based Runtime Verification, VMCAI'04, LNCS 2937, 44-57, Springer, 2004.
4. H. Barringer, K. Havelund, TraceContract: A Scala DSL for Trace Analysis, FM'11, LNCS 6664, 57-72, Springer, 2011.
5. H. Barringer, D. Rydeheard, K. Havelund, Rule Systems for Run-Time Monitoring: from Eagle to RuleR, RV'07, LNCS 4839, 111-125, Springer, 2007.
6. D. A. Basin, F. Klaedtke, S. Müller, E. Zalinescu, Monitoring Metric First-Order Temporal Properties, Journal of the ACM 62(2), 1-45, 2015.
7. A. Bauer, J.C. Küster, G. Vegliach, From Propositional to First-Order Monitoring, RV'13, LNCS 8174, 59-75, Springer, 2013.
8. R. E. Bryant, Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, ACM Comput. Surv. 24(3), 293-318, 1992.
9. C. Colombo, G.J. Pace, G. Schneider, LARVA – Safer Monitoring of Real-Time Java Programs (Tool Paper), SEFM'09, IEEE Computer Society, 33-37, 2009.
10. N. Decker, M. Leucker, D. Thoma, Monitoring Modulo Theories, Int. Journal on Software Tools for Technology Transfer 18(2), 205-225, Springer, 2016.
11. J. Goubault-Larrecq, J. Olivain, A Smell of ORCHIDS, RV'08, LNCS 5289, 1-20, Springer, 2008.
12. S. Hallé, R. Villemaire, Runtime Enforcement of Web Service Message Contracts with Data, IEEE Transactions on Services Computing 5(2), 192-206, 2012.
13. K. Havelund, Rule-based Runtime Verification Revisited, Int. Journal on Software Tools for Technology Transfer 17(2), 143-170, Springer, 2015.
14. K. Havelund, D. Peled, D. Ulus, First-order Temporal Logic Monitoring with BDDs, FMCAD'17, IEEE, 116-123, 2017.
15. K. Havelund, G. Rosu, Synthesizing Monitors for Safety Properties, TACAS'02, LNCS 2280, 342-356, Springer, 2002.
16. M. Kim, S. Kannan, I. Lee, O. Sokolsky, Java-MaC, A Run-time Assurance Tool for Java, RV'01, ENTCS 55(2), 218-235, Elsevier, 2001.
17. P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Rosu, An Overview of the MOP Runtime Verification Framework, Int. Journal on Software Tools for Technology Transfer 14(3), 249-289, Springer, 2012.
18. G. Reger, H. Cruz, D. Rydeheard, MarQ: Monitoring at Runtime with QEA, TACAS'15, LNCS 9035, 596-610, Springer, 2015.