Efficient Runtime Verification of First-Order Temporal Properties

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

[1] Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA
klaus.havelund@jpl.nasa.gov

[2] Department of Computer Science, Bar Ilan University, Ramat Gan, Israel
doron.peled@gmail.com

Abstract. Runtime verification allows monitoring the execution of a system against a temporal property, raising an alarm if the property is violated. In this paper we present a theory and system for runtime verification of a first-order past time linear temporal logic. The first-order nature of the logic allows a monitor to reason about events with data elements. While runtime verification of propositional temporal logic requires only a fixed amount of memory, the first-order variant has to deal with a number of data values potentially growing unbounded in the length of the execution trace. This requires special compactness considerations in order to allow checking very long executions. In previous work we presented an efficient use of BDDs for such first-order runtime verification, implemented in the tool DejaVu. We first summarize this previous work. Subsequently, we look at the new problem of dynamically identifying when data observed in the past are no longer needed, allowing to reclaim the data elements used to represent them. We also study the problem of adding relations over data values. Finally, we present parts of the implementation, including a new concept of user defined property macros.

1 Introduction

Runtime verification (RV) is used to check the execution of a system against a temporal property, expressed, e.g., in Linear Temporal Logic (LTL), alarming when it is violated, so that aversive action can be taken. To inspect an execution, the monitored system is instrumented to report on occurrences of events. The monitor performs incremental computation, updating its internal memory. It is important that this operation is efficient in terms of time and space, in order to be able to keep up with rapid occurrence of events in very long executions.

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

M. M. Gallardo and P. Merino (Eds.): SPIN 2018, LNCS 10869, pp. 26–47, 2018.
https://doi.org/10.1007/978-3-319-94111-0_2

Even if monitoring is performed offline, i.e. on log files, performance is an issue when logs are large. For each consumed event, a monitor has to decide whether the property is violated based on the finite part of the execution trace that it has viewed so far. Thus, the checked properties are often limited to *safety* properties [24]. For a safety property, any violating execution has a prefix that cannot be completed into an execution that satisfies it [2]. Hence, by definition, safety properties are those that are finitely refutable. In LTL, safety properties are those that can be written in the form $\Box\varphi$ (for *always* φ), where φ uses past operators: $\ominus$ for *previous-time* and $\mathcal{S}$ for *since* [25]. While it is sufficient to find one prefix that violates φ to deduce that $\Box\varphi$ does not hold, RV often keeps monitoring the system and reporting on further prefixes that fail to satisfy φ. We shall henceforth not prefix properties with $\Box$.

Two central challenges in RV are to increase the expressiveness of the properties that can be monitored, and to improve the efficiency of monitoring. Unfortunately, there is often a tradeoff between these goals. Therefore, the combined goal is to achieve a good balance that would allow checking the property we want to monitor with a reasonable efficiency. While propositional LTL is useful for describing some properties, in many cases we want to monitor executions with events that contain data values that need to be related to each other. Such properties can be expressed e.g., using first-order temporal logic or parametric automata. As monitoring may be done without assuming a bound on the length of the execution or the cardinality of the data elements, remembering an unbounded amount of elements may be unavoidable. Consider the property that asserts that each file that is closed was opened before. This can be expressed in a first-order temporal logic as follows (where $\mathbf{P}\,\varphi$ means: *sometime in the past* φ):

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

If we do not remember for this property *all* the files that were opened, then we will not be able to check when a file is closed whether it was opened before. This calls, in the first place, for using an algorithm and data structure where memory growth is often quite moderate, allowing to check large executions. In previous work [18] we presented an algorithm based on the use of BDDs, and its implementation in the DejaVu runtime verification tool. In this paper we go a step further in increasing efficiency (and hence the magnitude of executions that can be monitored) by presenting 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 referred to as *dynamic data reclamation.* As mentioned, the temporal formula (1) forces a monitor to store information about all the files that were ever opened so that it can check that no file is closed without being opened. Consider now a more refined specification, requiring that a file can be closed only if (in the previous step) it was opened before, and has not been closed since:

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

We can observe that if a file was opened and subsequently closed, then if it is closed again before opening, the property would be invalidated just as in the case where it was not opened at all. This means that we can "forget" that a file was opened when it is closed without affecting our ability to monitor the formula. Assume that at any time during the execution there are no more than N files opened simultaneously. Then, in the approach to be presented here, we need space for only N file names for monitoring the property. This is in contrast to our original algorithm, where space for all new file names must be allocated.

The contributions of the paper are the following. We present an elegant algorithm, and its implementation in DejaVu for dynamically reclaiming data elements that can not affect the value of the property anymore. Our solution is based on using a non-trivial combination of BDD operations to automatically detect values that are not further needed for the rest of the monitoring. Note that the approach does not involve static analysis of formulas. We furthermore introduce relations between variables (such as $x > y$) and a distinction between two forms of quantification: quantification over infinite domains and, as a new concept in DejaVu, quantification over values *seen* in the trace.

The remaining part of the paper is organized as follows. Section 2 presents the syntax and semantics of the logic, while Sect. 3 presents the BDD-based algorithm. Section 4 introduces the new dynamic data reclaiming algorithm. Section 5 introduces relations and the new forms of quantification over seen values. Section 6 outlines the implementation. Section 7 presents an evaluation of the dynamic data reclamation implementation. Section 8 describes related work, and finally Sect. 9 concludes the paper.

2 Syntax and Semantics

In this section we present briefly the syntax and semantics of the logic used by the DejaVu tool. Assume a finite set of domains $D_1, D_2, \ldots$. Assume further that the domains are infinite, e.g., they can be the integers or strings[1]. Let V be a finite set of *variables*, with typical instances x, y, z. An *assignment* over a set of variables W maps each variable $x \in W$ to a value from its associated domain $domain(x)$. For example $[x \rightarrow 5, y \rightarrow \text{"abc"}]$ maps x to 5 and y to "abc". Let T a set of *predicate names* with typical instances p, q, r. Each predicate name p is associated with some domain $domain(p)$. A predicate is constructed from a predicate name, and a variable or a constant of the same type. Thus, if the predicate name p and the variable x are associated with the domain of strings, we have predicates like $p(\text{"gaga"}), p(\text{"baba"})$ and $p(x)$. Predicates over constants are called *ground predicates*. An *event* is a finite set of ground predicates. (Some restrictions may be applied by the implementation, e.g., DejaVu allows only a single ground predicate in an event.) For example, if $T = \{p, q, r\}$, then $\{p(\text{"xyzzy"}), q(3)\}$ is a possible event. An *execution* $\sigma = s_1 s_2 \ldots$ is a finite sequence of events.

For runtime verification, a property φ is interpreted on prefixes of a monitored sequence. We check whether φ holds for every such prefix, hence, conceptually, check whether $\Box\varphi$ holds, where $\Box$ is the "always in the future" linear temporal

[1] For dealing with finite domains see [18].

logic operator. The formulas of the core logic, referred to as QTL (Quantified Temporal Logic) are defined by the following grammar. For simplicity of the presentation, we define here the logic with unary predicates, but this is not due to any principle limitation, and, in fact, our implementation supports predicates with multiple arguments.

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

The formula $p(a)$, where a is a constant in $\mathit{domain}(p)$, means that the ground predicate $p(a)$ occurs in the most recent event. The formula $p(x)$, for a variable $x \in V$, holds with a binding of x to the value a if a ground predicate $p(a)$ appears in the most recent event. The formula $(\varphi_1\ \mathcal{S}\ \varphi_2)$ means that φ_2 held in the past (possibly now) and since then φ_1 has been true. The property $\ominus\ \varphi$ means that φ was true in the previous event. We can also define the following additional operators: $\mathit{false} = \neg \mathit{true}$, $(\varphi \lor \psi) = \neg(\neg\varphi \land \neg\psi)$, $(\varphi \longrightarrow \psi) = (\neg\varphi \lor \psi)$, $\mathbf{P}\ \varphi = (\mathit{true}\ \mathcal{S}\ \varphi)$ (*previously* φ), $\mathbf{H}\ \varphi = \neg\mathbf{P}\ \neg\varphi$ (*historically* φ, or φ *always in the past*), and $\forall x\ \varphi = \neg\exists x \neg\varphi$. The operator $[\varphi, \psi)$, borrowed from [23], has the same meaning as $(\neg\psi\ \mathcal{S}\ \varphi)$, but reads more naturally as an interval.

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

- $(\epsilon, \sigma, i) \models \mathit{true}$.
- $(\epsilon, \sigma, i) \models p(a)$ if $p(a) \in \sigma[i]$.
- $([x \mapsto a], \sigma, i) \models p(x)$ if $p(a) \in \sigma[i]$.
- $(\gamma, \sigma, i) \models (\varphi \land \psi)$ if $(\gamma|_{\mathit{free}(\varphi)}, \sigma, i) \models \varphi$ and $(\gamma|_{\mathit{free}(\psi)}, \sigma, i) \models \psi$.
- $(\gamma, \sigma, i) \models \neg\varphi$ if not $(\gamma, \sigma, i) \models \varphi$.
- $(\gamma, \sigma, i) \models (\varphi\, \mathcal{S}\, \psi)$ if $(\gamma|_{\mathit{free}(\psi)}, \sigma, i) \models \psi$ or the following hold: $i > 1$, $(\gamma|_{\mathit{free}(\varphi)}, \sigma, i) \models \varphi$, and $(\gamma, \sigma, i-1) \models (\varphi\, \mathcal{S}\, \psi)$.
- $(\gamma, \sigma, i) \models \ominus\varphi$ if $i > 1$ and $(\gamma, \sigma, i-1) \models \varphi$.
- $(\gamma, \sigma, i) \models \exists x\ \varphi$ if there exists $a \in \mathit{domain}(x)$ such that[2] $(\gamma\,[x \mapsto a], \sigma, i) \models \varphi$.

Set Semantics. It helps to present the BDD-based algorithm by first refining the semantics of the logic in terms of sets of assignments satisfying a formula. Let $I[\varphi, \sigma, i]$ be the semantic function, defined below, that returns a set of assignments such that $\gamma \in I[\varphi, \sigma, i]$ iff $(\gamma, \sigma, i) \models \varphi$. The empty set of assignments $\emptyset$ behaves as the Boolean constant 0 and the singleton set that contains an assignment over an empty set of variables $\{\epsilon\}$ behaves as the Boolean constant 1. We define the union and intersection operators on sets of assignments, even if they are defined over non identical sets of variables. In this case, the assignments are extended over the union of the variables. Thus intersection between two sets of assignments A_1 and A_2 is defined like database “join” operator; i.e., it consists

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

of the assignments whose projection on the *common* variables agrees with an assignment in A_1 and with an assignment in A_2. Union is defined as the dual operator of intersection. Let A be a set of assignments over the set of variables W; we denote by $hide(A, x)$ (for "hiding" the variable x) the set of assignments obtained from A after removing from each assignment the mapping from x to a value. In particular, if A is a set of assignments over only the variable x, then $hide(A, x)$ is $\{\epsilon\}$ when A 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 $i > 0$.

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

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

3 An Efficient Algorithm Using BDDs

We describe here briefly an algorithm for monitoring first order past time LTL properties, first presented in [18] 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) [10]. A BDD is a compact representation for a Boolean valued function of type $\mathbb{B}^k \rightarrow \mathbb{B}$ for some $k > 0$ (where $\mathbb{B}$ is the Boolean domain $\{0, 1\}$), as a directed acyclic graph (DAG). A BDD is essentially a compact representation of a Boolean tree, where compaction glues together isomorphic subtrees. Each non-leaf node is labeled with one of the Boolean variables $b_0, \ldots, b_{k-1}$. A non-leaf node b_i is the source of two arrows leading to other nodes. A dotted-line arrow represents that b_i has the Boolean value 0, while a thick-line arrow represents that it has the value 1. The nodes in the DAG have the same order along all paths from the root. However, some of the nodes may be absent along some paths, when the result of the Boolean function does not depend on the value of the corresponding Boolean variable. Each path leads to a leaf node that is marked by either a 0 or a 1, representing the Boolean value returned by the function for the Boolean values on the path. Figure 1 contains five BDDs (a)–(e), over three Boolean variables b_0, b_1, and b_2 (referred to by their subscripts 0, 1, and 2), as explained below.

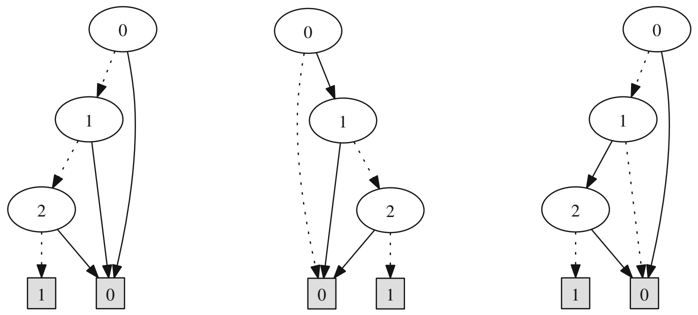
Mapping Data to BDDs. Assume that we see $p(\text{“ab”}), p(\text{“de”}), p(\text{“af”})$ and $q(\text{“fg”})$ in subsequent states in a trace, where p and q are predicates over the domain of strings. When a value associated with a variable appears for the first time in the current event (in a ground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

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

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

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

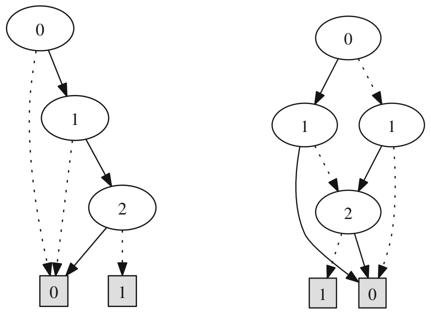
The function $\mathbf{build}(x, A)$ returns a BDD that represents the set of assignments where x is mapped to (the enumeration of) v for $v \in A$. This BDD is independent of the values assigned to any variable other than x, i.e., they can have any value. For example, assume that we use three Boolean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



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

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

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



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

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

Fig. 1. BDDs for the trace: $p(\text{"ab"}).p(\text{"de"}).p(\text{"af"}).q(\text{"fg"})$

Intersection and union of sets of assignments are translated simply to conjunction and disjunction of their BDD representation, respectively, and complementation becomes negation. We will denote the Boolean BDD operators as **and**, **or** and **not**. To implement the existential (universal, respectively) operators, we use the BDD existential (universal, respectively) operators over the 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**$(\langle x_0, \ldots, x_{k-1}\rangle, B_\varphi)$ is the BDD that represents the assignments satisfying $\exists x \varphi$ in the current state. Finally, BDD$(\bot)$ and BDD$(\top)$ are the BDDs that return always 0 or 1, respectively.

The algorithm, shown below, operates on two vectors (arrays) of values indexed by subformulas (as in [20]): 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 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 $\text{BDD}(\top)$ else $\text{BDD}(\bot)$.
   - $\mathsf{now}(p(x)) := \mathbf{build}(x, A)$ where $A = \{a | p(a) \in s\}$.
   - $\mathsf{now}((\varphi \wedge \psi)) := \mathbf{and}(\mathsf{now}(\varphi), \mathsf{now}(\psi))$.
   - $\mathsf{now}(\neg\varphi) := \mathbf{not}(\mathsf{now}(\varphi))$.
   - $\mathsf{now}((\varphi\ \mathcal{S}\ \psi)) := \mathbf{or}(\mathsf{now}(\psi), \mathbf{and}(\mathsf{now}(\varphi), \mathsf{pre}((\varphi \mathcal{S} \psi))))$.
   - $\mathsf{now}(\ominus\ \varphi) := \mathsf{pre}(\varphi)$.
   - $\mathsf{now}(\exists x\ \varphi) := \mathbf{exists}(\langle x_0, \ldots, x_{k-1}\rangle, \mathsf{now}(\varphi))$.
5. Goto step 2.

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

Dynamic Expansion of the BDDs. We can sometimes define the number k of Boolean variables per domain to be a large enough such that we anticipate no more than $2^k - 1$ different values. However, if the number of bits used for representing enumerations becomes insufficient, we can dynamically expand it during RV [18]. As explained above, the enumeration $11\ldots11$ of length k represents for every variable "all the values not seen so far in the input sequence". Consider the following two cases:

- When the added (most significant) bit has the value 0, the enumeration still represents the same value. Thus, the updated BDD needs to return the same values that the original BDD returned without the additional 0.
- When the added bit has the value 1, we obtain enumerations for values that were not seen so far in the input. Thus, the updated BDD needs to return the same values that the original BDD gave to $11\ldots11$.

An increase in one bit allows doubling the number of enumerations, hence, this, relatively expensive operation does not need to take place frequently (if at all). We demonstrate the expansion of the enumerations by a single bit on formulas with three variables, x, y and z, represented using three BDD bits each, i.e., x_0, x_1, x_2, y_0, y_1, y_2, z_0, z_1, z_2. We want to add a new most significant bit y_{new} for representing y. Let B be the BDD before the expansion. The case where the value of y_{new} is 0 is the same as for a single variable. For the case where y_{new} is 1, the new BDD needs to represent a function that behaves like B when all the y bits are set to 1. Denote this by $B[y_0 \setminus 1, y_1 \setminus 1, y_2 \setminus 1]$. This function returns the same Boolean values independent of any value of the y bits, but it depends on the other bits, representing the x and z variables. Thus, to expand the BDD, we generate a new one as follows:

$$((B \wedge \neg y_{new}) \vee (B[y_0 \setminus 1, y_1 \setminus 1, y_2 \setminus 1] \wedge y_{new}))$$

Expanding the number of bits allowed for enumerations, when needed, has the disadvantage that memory can grow unbounded. The next section suggests a method for identifying enumerations of values that can no longer affect the checked property, and therefore can be reclaimed.

4 Dynamic Data Reclamation

In the previous section, we described an algorithm that implements runtime verification with data, based on a BDD representation. The algorithm generates a new enumeration for each value that appears in the input and uses a hash table to map from a value to its enumeration. It is possible that the set of enumerations for some variable eventually exceeds the number of bits allocated for the BDD. In this case, the BDD can be expanded dynamically, as shown in the previous section. However, this can be very costly, and we may eventually run out of memory. In this section we study 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.

We saw in the introduction an example of a property where values that already occurred cannot be reclaimed (1), and a similar property, where there are values that are not useful any more from some point in the execution (2). Consider a more complicated example:

$$\forall z\, (r(z) \rightarrow \exists y\, (q(y) \mathcal{S} p(z))) \tag{3}$$

It asserts that when a ground predicate $r(a)$ appears with some value a, then there should be at least one value b for which $q(b)$ appeared ever since the most recent appearance of $p(a)$ (an appearance of $p(a)$ is required). Consider an execution σ with some prefix ρ that does not contain $r(a)$ since the most recent appearance of $p(a)$. Furthermore, no ground predicate $q(b)$ commonly appears since the last occurrence of $p(a)$. In this case, when $r(a)$ later occurs in σ, the property (3) will be violated. This is indistinguishable from the case where $p(a)$ never occurred. Thus, after seeing ρ, we can "forget" the value a.

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. Since these BDDs sufficiently summarize the information that will be used about the execution monitored so far, reclaiming data can be performed fully automatic, 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 includes 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 $hide(A[x = a], x) = hide(A[x = b], x)$. 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 and then, their corresponding values are removed from the hash table, and these 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, as explained in Sect. 3. 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 between the enumeration of a and the enumeration $11\ldots11$ on the pre BDDs. 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.

To simplify the presentation and prevent using many indexes, 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 all the enumerations of x values that are analogous to $11\ldots11$ in $\mathsf{pre}(\psi)$.

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

The available enumerations for the variable x are represented then by the conjunction of $I_{\psi,x}$ over all subformulas ψ of the specification φ. (Note that this will also include enumerations that are not used, as they are also analogous to $11\ldots11$ in all subformulas.)

To take advantage of reclaimed enumerations, we cannot generate them in successive order using a counter anymore. Thus we need to keep a set of available enumerations. This can be represented using a BDD. Let $avail(x)$ be the BDD that represents the enumerations (given as a Binary encoding $x_0, \ldots, x_{k-1}$) that are available for values of 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 the number of available enumerations becomes short, and 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, 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. Data reclaiming may sometimes be time consuming and also result in expanding the BDD. This is based on the observation that a BDD representing just the Binary enumerations from 1 to n is much more compact than a BDD representing some n random enumerations. On the other hand, as we observe in the evaluation section, the ability to use less Boolean variables for enumerations due to reclaiming data may require less memory and time.

As the BDD-based algorithm detects an enumeration e that 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* [11]: 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. 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 Relations and Quantification over Seen Values

Quantification over Seen Values. The QTL logic allows assertions over infinite domains (for handling finite domains, see [18]). The quantification defined in Sect. 2 is over all possible domain values, whether they appeared already in the monitored sequence or not. It is sometimes more natural to quantify only over values that already appeared in events. Consider the property that asserts that there exists a session s, such that any user u that has ever logged into any session (s') so far, is currently logged into this session s and not logged out yet. This can be expressed in the following slightly inconvenient manner:

$$\exists s \forall u ((\exists s' \, \mathbf{P} \, login(u, s')) \rightarrow (\neg logout(u, s) \, \mathcal{S} \, login(u, s))) \tag{4}$$

One may be tempted to use the following shorter formula with the naive intent that we quantify over all users u seen in the trace so far:

$$\exists s \forall u (\neg logout(u, s) \, \mathcal{S} \, login(u, s)) \tag{5}$$

Unfortunately, this formula does not have the intended semantics because it asserts that *all potential users* have not logged out since they logged into s. For an unbounded domain, this property can never hold, since at any time, only a finite number of users could have been observed to log in. Property (5) can, however, be corrected to conveniently quantify only over values u that were seen so far in the monitored sequence. We extend the logic QTL with the bounded quantifiers $\tilde{\exists}$ and $\tilde{\forall}$, quantifying over only seen values, hence we can now express property (4) as:

$$\exists s \tilde{\forall} u (\neg logout(u,s)\, \mathcal{S}\, login(u,s)) \tag{6}$$

The new kind of quantifiers do not extend the expressive power of the logic, as one can always use the form of property (4) to limit the quantification to seen values. However, it allows writing shorter formulas, and is also supported by an efficient implementation.

In order to implement the quantifiers $\tilde{\exists}$ and $\tilde{\forall}$, we keep, for each variable x that is quantified in this way, a BDD $seen(x)$. $seen(x)$ is initialized to the empty BDD (BDD($\bot$)). Upon seeing an event with a new value a for x, we update $seen(x)$ such that for the BDD bits representing the new enumeration e for a it will also return 1. That is, $seen(x) := \mathbf{or}(seen(x), \mathrm{B}DD(e))$. We augment our algorithm with $\mathsf{now}(\tilde{\exists} x\, \varphi) := \mathbf{exists}(\langle x_0, \ldots, x_{k-1}\rangle, \mathbf{and}(seen(x), \mathsf{now}(\varphi)))$. For implementing $\tilde{\forall} x$, note that $\tilde{\forall} x\, \varphi = \neg \tilde{\exists} x\, \neg \varphi$.

Arithmetic Relations. Another extension of the QTL logic is the ability to use arithmetic relations. This allows comparing between values that occurred, as in the following property:

$$\forall x\, (p(x) \rightarrow \exists y \ominus (\mathbf{P}\, q(y) \wedge x > y)) \tag{7}$$

It asserts that if $p(x)$ is seen with some value of x, then there exists a smaller value y such that $q(y)$ was seen in the past. In order to implement this comparison along the same lines of the set semantics BDD-based solution, we can represent a BDD $\mathsf{now}(x > y)$ over the enumerations of the variables x and y. Suppose that x and y are represented using the BDD bits $x_0, \ldots, x_{k-1}, y_0, \ldots y_{k-1}$. Then, $\mathsf{now}(x > y)$ returns a 1 when $x_0, \ldots, x_{k-1}$ represents the enumeration for some seen value a, and $y_0, \ldots, y_{k-1}$ represents the enumeration of some seen value b, where $b > a$.

The BDD $\mathsf{now}(x > y)$ is updated incrementally, when a new value for x or for y is seen. For property (7), that would be an occurrence of an event that contains a ground predicate of the form $p(a)$ or $q(b)$. Suppose that a is a new value for the variable x. We build at this point a temporary BDD $B_{a>x}$ that represents the set of assignments $\{[x \mapsto a, y \mapsto b] \mid b \in seen(y) \wedge a > b\}$. Then we set $\mathsf{now}(x > y) := \mathbf{or}(\mathsf{pre}(x > y), B_{a>y})$.

Property (7) guarantees (due to the subformula $\mathbf{P}\, q(y)$) that the values compared using $x > y$ were already seen. The following property, however, appears more ambiguous since the domain of y is not completely clear:

$$\forall x\, (p(x) \rightarrow \exists y\, x > y) \tag{8}$$

For example, assuming standard mathematical reasoning, if y ranges over the integers, then this property should always hold; if y ranges over the naturals, then this should hold if x is bigger than 0, although some definitions of the naturals do not contain 0, in which case this should hold if x is bigger than 1. To solve ambiguity, we chose an alternative implementation; we analyze the formulas, and if a relation contains an occurrence of a variable x in the scope of a quantification $\forall x$ or $\exists x$, we change the quantification into $\tilde{\forall} x$ or $\tilde{\exists} x$, respectively.

6 Implementation

Basic Algorithm. DejaVu is implemented in Scala. The current version, which supports data reclamation, is an augmentation of the tool previously described in [18]. DejaVu takes as input a specification file containing one or more properties, and generates a self-contained Scala program (a text file) - the monitor. This program (which first must be compiled) takes as input the trace file and analyzes it. The tool uses the following libraries: Scala's parser combinator library for parsing [28], the Apache Commons CSV (Comma Separated Value format) parser for parsing log files [4], and the JavaBDD library for BDD manipulations [21]. We shall illustrate the monitor generation using an example. Consider the property Close in Fig. 4, which corresponds to property 1 on page 2, but in the input format for the tool. The property-specific part[3] of the generated monitor, shown in Fig. 2 (left), relies on an enumeration of the subformulas, shown in Fig. 2 (right). Specifically, two arrays are declared,

```
...
class Formula_p extends Formula(monitor) {
  var pre  : Array[BDD] = Array. fill (5)(False)
  var now : Array[BDD] = Array. fill (5)(False)
  var tmp : Array[BDD] = null
  val var_f :: Nil = declareVariables(("f",false))

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

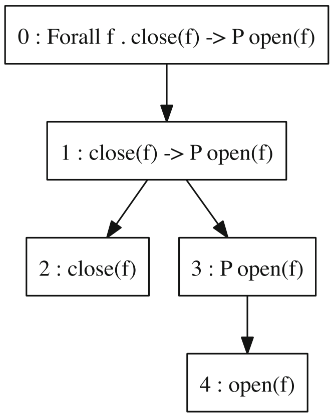

0 : Forall f . close(f) -> P open(f)
1 : close(f) -> P open(f)
2 : close(f)
3 : P open(f)
4 : open(f)

Fig. 2. Monitor (left) and subformula enumeration (right) for property Close

[3] An additional 530 lines of property independent boilerplate code is generated.

indexed by subformula indexes: pre for the previous state and now for the current state. For each observed event, the function evaluate() computes the now array from highest to lowest index, and returns true (property is satisfied in this position of the trace) iff now(0) is not *false*, i.e., not BDD($\bot$). At composite subformula nodes, BDD operators are applied. For example for subformula 3, the new value is now(4).or(pre(3)), which is the interpretation of the formula P open(f) corresponding to the law: $\mathbf{P}\,\varphi = (\varphi \vee \ominus\, \mathbf{P}\,\varphi)$. As can be seen, for each new event, the evaluation of a formula results in the computation of a BDD for each subformula. It turns out that this process, linear in the size of the formula, is rather efficient.

Dynamic Memory Reclamation. The implementation of dynamic data reclamation is illustrated with the code snippets in Fig. 3. The method build(...)(...):BDD (lines 1–3) is called in Fig. 2 (left) for subformulas 2 and 4. It in turn calls the method getBddOf(v: Any):BDD (line 2) on the Variable object (the class of which is defined on lines 5–34) denoted by the variable name 'name' (looked up in varMap), and with the value v occurring in the current event. This method (lines 6–17) returns the BDD corresponding to the newly observed value v. In case the value has previously been encountered (line 7–9), the previously allocated BDD is returned. Otherwise (line 10), if the variable avail of available BDDs is False (none available), the data reclamation is activated (explained below). An Out Of Memory error is thrown in case there still are no available BDDs (line 11). Otherwise (line 12), we select an available BDD from avail using JavaBDD's SAT solver (satOne() which when called on a BDD returns some bit enumeration for which the BDD returns 1). The selected BDD (result) is then "removed" from the avail BDD (line 13), and the hash table from values to the BDDs that represent them is finally updated (line 14). Note that the hash table maps each data value directly to a BDD representing its enumeration. The method reclaimData() (lines 19–26) starts with an avail BDD allowing any assignment different from $11\ldots11$ (line 20), and then refines it by repeatedly (line 22) computing formula $I_{\psi,x}$ from Sect. 4 for each temporal subformula (the method getFreeBDDOf(...):BDD computes $I_{\psi,x}$.), and and-ing the result to avail (line 23), corresponding to down-selecting with set intersection. The method removeReclaimedData() (lines 36–42) finally removes those values that map to a BDD that is included in avail. The test bdd.imp(avail).isOne (line 31) is the logic formulation of "the BDD avail contains the BDD bdd".

Relations and Quantification over Seen Values. Relations and quantification over seen values are implemented in a straight forward manner as explained in Sect. 5. We shall explain here just the concrete syntax chosen for relations and quantifiers. Relations are written exactly as in Sect. 5, e.g. y $<$ x. It is also possible to compare variables to constants, as in z $<$ 10. Concerning the two forms of quantifiers (quantification over infinite domains and quantification over seen values) we use **Exists** (for $\exists$) and **Forall** (for $\forall$) to quantify over infinite domains, and **exists** (for $\tilde{\exists}$) and **forall** (for $\tilde{\forall}$) to quantify over seen values. For

```
def build(name: String)(patterns: Pattern∗): BDD = {
  ... varMap(name).getBDDOf(v) ...
}

class Variable(name: String) {
  def getBDDOf(v: Any): BDD = {
    if (bdds.contains(v)) {
      bdds(v)
    } else {
      if (avail.isZero) reclaimData()
      if (avail.isZero) error()
      val result = avail.satOne(...)
      avail = avail.and(result.not())
      bdds = bdds + (v → result)
      result
    }
  }

  def reclaimData(): Unit = {
    avail = allOnes.not
    for (i ← indicesOfTemporalOps) {
      val bdd_i = getFreeBDDOf(name, pre(i))
      avail = avail.and(bdd_i)
    }
    removeReclaimedData()
  }

  def removeReclaimedData(): Unit = {
    for (v ← bdds.keySet) {
      val bdd = bdds(v)
      if (bdd.imp(avail).isOne) bdds = bdds − v
    }
  }
}

def getFreeBDDOf(varName: String, formula: BDD): BDD = {
  val variable = varMap(varName)
  val formulaWithOnes = formula.restrict(variable.allOnes)
  var result = formulaWithOnes.biimp(formula)
  for (quantVar ← otherQuantVars(varName)) result = result.forAll(quantVar)
  result
}
```

Fig. 3. Implementation of dynamic data reclamation

example, property (6) in Sect. 5, reading $\exists s \tilde{\forall} u(\neg logout(u, s)\, \mathcal{S}\, login(u, s))$ (*there exists a session s such that any user u that has ever logged into a session so far, is currently logged into session s - and not logged out yet*), is expressed as follows in DejaVu's concrete syntax: **Exists** s . **forall** u . !logout(u,s) **S** login (u,s).

7 Evaluation of Dynamic Data Reclamation

In this section we evaluate the implementation of DejaVu's dynamic data reclamation algorithm. We specifically evaluate the four temporal properties shown in Fig. 4, written in DejaVu's input notation, on different sizes and shapes of traces (auto-generated specifically for stress testing the algorithm), while varying the number of bits allocated to represent variables in BDDs. The properties come in two variants: those that do not trigger data reclamation and therefore cause accumulation of data in the monitor, and those (who's names have suffix 'DR') that do trigger data reclamation, and therefore save memory. The first two properties model variants of the requirement that a file can only be closed if has been opened. Property Close corresponds to formula (1), and will not trigger data reclamation as explained previously. Property CloseDR corresponds to formula (2) and will trigger data reclamation. The next two properties model variants of the requirement that a file cannot be opened if it has already been opened. Property Open states that if a file is opened then either (in the previous step) it must have been closed in the past and not opened since, or it must not have been opened at all in the past. This latter disjunct causes the formula not to trigger data reclamation, essentially for the same reason as for Close. Finally, property OpenDR states this more elegantly by requiring that if a file was opened in the past, and not closed since then, it cannot be opened. This property will trigger data reclamation.

```
// A file can only be closed if has been opened:
prop close   : Forall f . close(f) → P open(f)
prop closeDR : Forall f . close(f) → @ (! close(f) S open(f))

// A file cannot be opened if it has already been opened:
prop open    : Forall f . open(f) → @ (((! open(f) S close(f))) | ! P open(f))
prop openDR  : Forall f . @ (! close(f) S open(f)) → ! open(f)
```

Fig. 4. Four evaluation properties

Table 1 shows the result of the evaluation, which was performed on a Mac OS X 10.10.5 (Yosemite) operating system with a 2.8 GHz Intel Core i7, and 16 GB of memory. The properties were evaluated on traces of sizes spanning from (approximately) 2 million to 3 million events, with approximately 1–1.5 million files opened in each log (see table for exact numbers). Traces have the following general form: initially O files, where O ranges from 0 to 50,000, are opened in order to accumulate a large amount of data stored in the monitor. This is followed by a repetitive pattern of closing C files and opening C **new** files. This pattern is repeated R times. The shape of each log is shown in the table as $O{:}C{:}R$. For example, for Log 1 we have that: $O{:}C{:}R = 50,000 : 1,000 : 1,000$.

For each combination of property and log file, we experimented with different number of bits used to represent observed file values in the traces: 21, 20, 16,

Table 1. Evaluation. For each property, the performance against each log is shown. For each log, its size in number of events, and number of files opened are shown. Furthermore the pattern of the log is shown as three numbers $O : C : R$ where O is the number of events opened initially, C is the number of close events and new open events in each iteration, and R indicates how many times the C pattern is repeated. For each experiment is shown how many bits (followed by a 'b') per variable, how many seconds ('s') the trace analysis took, and whether there was an out of memory (**OOM**) or whether the presented data reclamation was invoked (**dr**).

Property	Log 1	Log 2	Log 3	Log 4
	2,052,003 events 1,051,000 files 50,000:1,000:1,000	3,007,003 events 1,504,000 files 1,000:500:3,000	2,400,009 events 1,200,006 files 6:5:200,000	2,000,004 events 1,000,001 files 1:1:1,000,000
Close	21b : 10.2s 20b : 12.5s **OOM**	21b : 14.3s 20b : 12.5s **OOM**	21b : 10.6s 20b : 12.8s **OOM**	20b : 9.5s 2b : 0.6s **OOM**
CloseDR	20b : 13.3s **dr** 15b : 14.8s **dr**	21b : 17.0s 20b : 21.2s **dr** 10b : 10.6s **dr**	21b : 12.5s 20b : 14.0s **dr** 10b : 7.6s **dr** 3b : 5.4s **dr**	20b : 9.0s 2b : 4.7s **dr**
Open	21b : 15.2s 20b : 17.5s **OOM**	21b : 25.3s 20b : 18.9s **OOM**	21b : 17.1s 20b : 17.7s **OOM**	20b : 11.7s 2b : 0.6s **OOM**
OpenDR	20b : 15.2s **dr** 16b : 16.1s **dr**	20b : 27.4s **dr** 10b : 10.6s **dr**	20b : 13.8s **dr** 10b : 8.2s **dr** 3b : 5.5s **dr**	20b : 9.1s 2b : 5.6s **dr**

15, 10, 3, and 2 bits, corresponding to the ability to store respectively 2097151, 1048575, 65535, 32767, 1023, 7, and 3 different values for each variable (having subtracted the 11...11 pattern reserved for representing "all other values"). The following abbreviations are used: **OOM** = Out of Memory (the number of bits allocated for a variable are not sufficient), and **dr** = data reclamation according to the algorithm presented in this paper has occurred. Typically when data reclamation occurs, approximately 1–1.5 million data values are reclaimed.

Table 1 demonstrates that properties Close and Open run out of memory (bits) if the allocated number of bits is not large enough to capture all opened files. For these properties, 21 bits are enough, while 20 bits are insufficient in most cases. On the other hand, the properties CloseDR and OpenDR can be monitored on all logs without **OOM** errors, by invoking the data reclamation, and without substantial differences in elapsed trace analysis time. In fact, we observe for these latter two properties, that as we reduce the number of bits, and thereby force the data reclamation to occur more often, the lower are the elapsed trace analysis times. As a side remark, for logs with an initially large amount of file openings, specifically logs 1 and 2, a certain minimum amount of bits are required to store these files. E.g. we cannot go below 15 bits for the CloseDR property on log 1. In contrast, we can go down to 2 bits for log 4 for the same property, even though logs 1 and 4 have approximately the same length and the same number of files being opened. In summary, for certain data reclamation friendly properties, data reclamation can allow monitoring of traces that would otherwise not be monitorable. In addition, data reclamation combined with reducing the number of bits representing variables seems to reduce execution time, a surprisingly positive result. We had in fact expected the opposite result.

It is well known that efficiency on BDD-based techniques are sensitive to the ordering of variables in the BDDs. Currently, as already indicated, the variable corresponding to the least significant bit always occurs first (at the top), and the variable corresponding to the most significant bit appears last (at the bottom), in the BDD. One may consider alternative orderings, either determined statically from the formula or dynamically as monitoring progresses. We have not explored such alternatives at the time of writing. Another factor potentially influencing efficiency may be the structure of monitored data. Consider e.g. the monitoring of data structures in a program, such as *sets*, *lists*, or, generally, *objects* in an object-oriented programming language. It is here important to stress that the shape of data monitored is not reflected in the BDDs themselves, but only concerns the mapping from data to BDDs using a hash table, which indeed supports complex data keys. However, as we have only experimented with offline log file analysis, we have not explored this online monitoring problem.

Macros. A new addition to DejaVu is the possibility to define data parameterized macros representing subformulas, which can be called in properties, without having any performance consequences for the evaluation. Macros are expanded at the call site. Macros can call macros in a nested manner, supporting a compositional way of building more complex properties. Figure 5 illustrates the use of macros to define the properties from Fig. 4, and should be self explanatory. Also, events can be declared up front in order to ensure that properties refer to the correct events, and with the correct number of arguments (not shown here).

```
pred isOpen(f) = !close(f) S open(f)
pred isClosed(f) = !open(f) S close(f)
pred wasOpened(f) = P open(f)

// A file can only be closed if has been opened:
prop close   : Forall f . close(f) → wasOpened(f)
prop closeDR : Forall f . close(f) → @ isOpen(f)

// A file cannot be opened if it has already been opened:
prop open   : Forall f . open(f) → @ (isClosed(f) | ! wasOpened(f))
prop openDR : Forall f . @ isOpen(f) → ! open(f)
```

Fig. 5. our evaluation properties using macros

8 Related Work

There are several systems that allow monitoring temporal properties with data. The systems closest to our presentation, in monitoring first-order temporal logic, are MonPoly [8] and Ltl$^{\text{FO}}$ [9]. As in the current work, MonPoly monitors first-order temporal properties. In fact, it also has the additional capabilities of asserting and checking properties that involve progress of time and a limited capability of reasoning about the future. The main difference between our system and MonPoly is in the way in which data are represented and manipulated. MonPoly exists in two versions. The first one models unbounded sets of values using regular expressions (see, e.g., [22] for a simple representation of sets of values). This version allows unrestricted complementation of sets of data values. Another version of MonPoly, which is several orders of magnitude faster according to [8], is based on storing finite sets of assignments, and applying database operators to these. In that implementation complementation, and some of the uses of logical and modal operators is restricted, due to the explicit finite set-representation used. This has as consequence that not all formulas are monitorable, see [19] for details. The BDD representation in DejaVu provides full expressiveness, allowing for any arbitrary combination of Boolean operators, including negation, temporal operators, and quantifiers, and with a fully compositional interpretation of formulas. In [18] we compared DejaVu to this latter version of MonPoly and showed performance advantages of DejaVu.

Ltl$^{\text{FO}}$ [9] supports first-order future time LTL, where quantification is restricted to those elements that appear in the current position of the trace. The monitoring algorithm is based on spawning automata. Monitoring first-order specifications has also been explored in the database community [12] in the context of so-called temporal triggers, which are first-order temporal logic specifications that are evaluated w.r.t. a sequence of database updates.

An important volume of work on data centric runtime verification is the set of systems based on trace slicing. Trace slicing is based on the idea of mapping variable bindings to propositional automata relevant for those particular bindings. This results in very efficient monitoring algorithms, although with limitations w.r.t. expressiveness. Systems based on trace slicing include TraceMatches [1], Mop [26], and Qea [27]. Qea is an attempt to increase the expressiveness of the trace slicing approach. It is based on automata, as is the Orhcids system [15]. Other systems include BeepBeep [16] and TraceContract [6], which are based on future time temporal logic using formula rewriting. Very different kinds of specification formalisms can be found in systems such as Eagle [5], Ruler [7], LogFire [17] and Lola [3]. The system Mmt [14] represents sets of assignments as constraints solved with an SMT solver. An encoding of enumerations of values as BDDs appears in [29], where BDDs are used to represent large relations in order to efficiently perform program analysis expressed as Datalog programs. However, that work does not deal with unbounded domains.

Concerning reclamation of data values no longer needed in a monitor we are aware of the following alternative approaches. MonPoly is interesting since it is part of the monitoring algorithm to get rid of such unnecessary values, and as such data reclamation is not an orthogonal concept. This is possible due to the explicit representation of sets of assignments. However, as already mentioned, the explicit representation has as consequence that some formulas are not monitorable. In Larva [13] it is up to the user to indicate that an entire monitor can be garbage collected by using acceptance states: when the monitor enters such an acceptance state it can be discarded, and released for normal garbage collection. In systems such as Ruler [7], LogFire [17] and TraceContract [6], monitor states get garbage collected the normal way when data structures are no longer needed. The last variant occurs in Mop [26], where monitored data values can be structured objects in the monitored program (such as a set, a list, an iterator). When such a monitored object is no longer used in the monitored program, a garbage collector would normally collect it. However, if the monitor keeps a reference to it, this is not possible. To circumvent this, Mop monitors use Java's so-called *weak references* to refer to objects in the monitored program. An object referenced only by weak references is considered to be garbage by the garbage collector. Hence the object is garbage collected when nothing or only monitors refer to it.

9 Conclusion

We described a BDD-based runtime verification algorithm for checking the execution of a system against a first-order past time temporal logic property. The propositional version of such a logic is independent of the length of the prefix seen so far. The first-order version may need to represent an amount of values that can grow linearly with the number of data values observed so far. The challenge is to provide a compact representation that will grow slowly and can be updated quickly with each incremental calculation that is performed per each new monitored event, even for very long executions.

We used a BDD representation of sets of assignments for the variables that appear in the monitored property. While the size of such BDDs can grow linearly with the number of represented values, it is often much more compact, and the BDD functions of a standard BDD package are optimized for speed. Our representation allows assigning a large number of bits for representing the encoding of values, so that even extremely long executions can be monitorable. However, a lower number of bits is still preferable to a larger number of bits. We presented an algorithm and its implementation for dynamically reclaiming data no longer used, as a function of all current subformula BDDs, representing sets of assignments. That is, the specification is not statically analyzed to achieve this reclamation. Experiments demonstrated that even frequent activation of data reclamation is not necessarily costly, and in fact in combination with a lower number of bits needed can reduce the trace analysis time compared to using more bits and no data reclamation.

We also presented support for numerical relations between variables and constants, and a new form of quantification over values seen in the trace. Future work includes support for functions applied to data values seen in the trace, and real-time constraints. Other future work includes comparison with slicing-based algorithms, as found in e.g. Mop [26] and Qea [27], which are very efficient, however at the price of some limited expressiveness.

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L.J., Kuzins, S., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: OOPSLA 2005, pp. 345–364. IEEE (2005)
2. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distrib. Comput. **2**(3), 117–126 (1987)
3. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: TIME 2005, pp. 166–174 (2005)
4. Apache Commons CSV parser. https://commons.apache.org/proper/commons-csv
5. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_5
6. Barringer, H., Havelund, K.: TraceContract: a Scala DSL for trace analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 57–72. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_7
7. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from Eagle to RuleR. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 111–125. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77395-5_10
8. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 45 (2015)
9. Bauer, A., Küster, J.-C., Vegliach, G.: From propositional to first-order monitoring. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 59–75. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1_4
10. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
11. Cormen, Th.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. The MIT Press and McGraw-Hill Book Company (1989)
12. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. ACM Trans. Database Syst. **20**(2), 149–186 (1995)
13. Colombo, C., Pace, G.J., Schneider, G.: LARVA - safer monitoring of real-time Java programs (tool paper), 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM), Hanoi, Vietnam, pp. 33–37. IEEE Computer Society (2009)
14. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. J. Softw. Tools Technol. Transf. **18**(2), 205–225 (2016)
15. Goubault-Larrecq, J., Olivain, J.: A smell of ORCHIDS. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 1–20. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89247-2_1
16. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Serv. Comput. **5**(2), 192–206 (2012)
17. Havelund, K.: Rule-based runtime verification revisited. J. Softw. Tools Technol. Transf. **17**(2), 143–170 (2015)

18. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with BDDs. In: FMCAD 2017, pp. 116–123. IEEE (2017)
19. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring events that carry data. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 61–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_3
20. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_24
21. JavaBDD. http://javabdd.sourceforge.net
22. Henriksen, J.G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: monadic second-order logic in practice. In: Brinksma, E., Cleaveland, W.R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60630-0_5
23. Kim, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: a run-time assurance tool for Java. In: Proceedings of the 1st International Workshop on Runtime Verification (RV 2001). ENTCS, vol. 55(2). Elsevier (2001)
24. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **3**(2), 125–143 (1977)
25. Manna, Z., Pnueli, A.: Completing the temporal picture. Theor. Comput. Sci. **83**, 91–130 (1991)
26. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. J. Softw. Tools Technol. Transf. **14**, 249–289 (2012)
27. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 596–610. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_55
28. Scala Parser Combinators. https://github.com/scala/scala-parser-combinators
29. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog with binary decision diagrams for program analysis. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 97–118. Springer, Heidelberg (2005). https://doi.org/10.1007/11575467_8