Noname manuscript No.
(will be inserted by the editor)

First-Order Temporal Logic Monitoring with BDDs

Klaus Havelund · **Doron Peled** · **Dogan Ulus**

Received: date / Accepted: date

Abstract Runtime verification is aimed at analyzing execution traces stemming from a running program or system. The traditional purpose is to detect the lack of conformance with respect to a formal specification. Numerous efforts in the field have focused on monitoring parametric specifications, where events carry data, and formulas can refer to such. Since a monitor for such specifications has to store observed data, the challenge is to have an efficient representation and manipulation of Boolean operators, quantification, and lookup of data. The fundamental problem is that the actual values of the data are not necessarily bounded or provided in advance. In this work we explore the use of Binary Decision Diagrams (BDDs) for representing observed data. Our experiments show a substantial improvement in performance compared to related work.

Keywords Past time temporal logic · data · BDDs

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”.

K. Havelund
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA
E-mail: klaus.havelund@jpl.nasa.gov

D. Peled
Department of Computer Science, Bar Ilan University, Ramat Gan, Israel
E-mail: doron.peled@gmail.com

D. Ulus
Boston University, Boston, MA, USA
E-mail: doganulus@gmail.com

1 Introduction

Runtime verification (RV) allows checking whether a temporal property holds during the execution of a system. The system execution can be considered as emitting an execution trace, i.e., a sequence of events, which is then consumed and checked by a monitor. A monitor performs for each received event some incremental computation that is aimed at detecting and warning as soon as the temporal property is violated. The field of model checking has mostly focused on propositional logics [22]. Early RV systems were also based on specifications given in some form of propositional temporal logic. A propositional temporal logic formula can be translated into a finite automaton, where the incremental computation updates the automaton state based on the recent input reporting information captured from the monitored system. However, the state of the art in RV has for some time focused on monitoring parametric specifications, where events carry data, and formulas can refer to such. Since such a monitor has to store observed data, the challenge is using a representation that allows efficient manipulation of operators over sets of observed data. The field of RV has not settled on a single best solution. As is usually the case, there are compromises to be made with respect to the efficiency of algorithms and expressiveness of logics.

Temporal logics usually come in two variants: future and past (or mixtures). As future temporal properties depend on an infinite input, after a finite sequence of events one may only provide partial information about whether the property holds; namely, if it is already violated, already achieved, or undecided yet [22]. The focus of this work is past time temporal properties, which are also classified as the safety temporal properties [2, 23], and are properties for which we are capable of detecting a violation based on the monitored current prefix of the execution, as soon as it occurs. As an example, consider a predicate $open(f)$, indicating that a file f is being opened, and a predicate $close(f)$ indicating that f is being closed. We can formulate that a file cannot be closed unless it was opened before with the following first-order past time temporal logic formula:

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

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

$$\forall f\,(close(f) \longrightarrow \ominus(\neg close(f)\,S\,open(f)))$$

One problem we need to solve is the unboundedness caused by negation. For example, assume that we have only observed so far one *close* event $close(\text{"ab"})$. The subformula $close(f)$ is therefore satisfied for the value $f =$ "ab". The subformula $\neg close(f)$ is satisfied by all values from the domain of f *except* for "ab". This set contains those values that we have not seen yet in the input within a *close* event.

We need a representation of finite and infinite sets of values, upon which applying complementation is efficient.

We present a first-order past time temporal logic, named QTL (Quantified Temporal Logic), and an implementation, named DEJAVU[1] based on a BDD (Binary Decision Diagram) representation of sets of assignments of values to the free variables of subformulas. Instead of storing the values assigned to variables, we enumerate input values as soon as we see them and use Boolean encodings of this enumeration. We use BDDs to represent sets of such enumerations. For example, if the runtime verifier sees the input events *open*("a"), *open*("b"), *open*("c"), it will encode them as 000, 001 and 010 (say, we use 3 bits b_0, b_1 and b_2 to represent each enumeration, with b_2 being the most significant bit). A BDD that represents the set of values {"a","c"} would be equivalent to a Boolean function $(\neg b_0 \wedge \neg b_2)$ that returns 1 for 000 and 010 (the value of b_1 can be arbitrary). This approach has the following benefits:

- It is highly compact. With k bits used for representing enumerations, the BDD can grow to $2^{O(k)}$ nodes [10]; but BDDs usually compact the representation very well [11]. In fact, we often do not pay much in overhead for keeping surplus bits. Thus, we can start with an overestimated number of bits k such that it is unlikely to see more than 2^k different values for the domain they represent. We can also incrementally extend the BDD with additional bits when needed at runtime.
- Complementation (negation) is efficient, by just switching between the 0 and 1 leaves of the BDD. Moreover, even though at any point we may have not seen the entire set of values that will show up during the execution, we can safely (and efficiently) perform complementation: values that have not appeared yet in the execution are being accounted for and their enumerations are reserved already in the BDD before these values appear.
- Our representation of sets of assignments as BDDs allows a very simple algorithm that naturally extends the dynamic programming monitoring algorithm for propositional past time temporal logic shown in [18].

We first define a semantics of a past time first-order linear temporal logic that determines whether an assignment of values to (free) variables satisfies a formula after a given finite sequence. We then provide an equivalent semantics as a function that returns the set of assignments satisfying the property at that prefix, based on set operators. For the final algorithm, we replace the set operators by logical operators on BDDs, e.g., union is replaced by disjunction. We only have to keep values to represent the current and previous state in the execution.

The remaining part of the paper is organized as follows. Section 1 discusses related work. Section 2 presents the syntax and semantics of the QTL temporal logic. Section 3 presents the BDD-based algorithm for monitoring a trace against a QTL formula. Section 4 outlines the implementation, and Section 5 presents an evaluation of the implementation. Finally, Section 6 concludes the paper.

This paper is an extension of the conference paper [16] with the following additional contributions. In Section 3.4 we present a set theoretical characterization of sets of assignments that satisfy properties of our temporal properties after a finite sequence. This is based on a finite representation of infinite sets of assignments. This

[1] DEJAVU is available at `https://github.com/havelund/dejavu`.

presentation provides additional intuition for the BDD-based solution presented in this paper. The evaluation in Section 5 of the DEJAVU tool is extended with additional experiments showing new results. Finally, various figures of BDDs have been added to explain the approach better.

Related Work

There are several systems that allow monitoring temporal properties with data. The system closest to our presentation, in monitoring first-order temporal logic, is MONPOLY [7]. As in the current work, it monitors first-order temporal properties. In fact, it also has the additional capabilities of asserting and checking properties that involve arithmetic relations among the data elements, 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., [20] for a simple representation of sets of values). That version allows unrestricted complementation of sets of data values. Another version of MONPOLY, which is several orders of magnitude faster according to [7], is based on storing finite sets of assignments, and applying database operators to these. We compare our tool to this latter version in Section 5. In that implementation complementation is restricted, to account for finite sets. Our system is based on representing sets of enumerations of data values as BDD functions, and does not restrict negation.

An important volume of work on data centric runtime verification is the set of systems based on trace slicing. Trace slicing maps 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 [24], and QEA [25]. QEA is an attempt to increase the expressiveness of the trace slicing approach. It is based on automata, as is the ORHCIDS system [13].

Other systems include BEEPBEEP [14] and TRACECONTRACT [5], 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 [4], RULER [6], LOGFIRE [15] and LOLA [3]. The system MMT [12] represents sets of assignments as constraints solved with an SMT solver. An encoding of BDD functions over enumerations of values appears in [27], 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.

2 Syntax and Semantics

We define here the syntax and semantics for the QTL logic. 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. (In Section 3.3 it is explained how to deal with finite domains.) Let V be a finite set of *variables*, with typical instances x, y, z. We denote by $x : D$

the fact that the variable x has the domain D. An *assignment* over a set of variables V maps each variable $x \in V$ to a value from its associated domain $domain(x)$, where multiple variables (or all of them) can be related to the same domain. For example $[x \to 5, y \to \text{“abc”}]$ assigns the values 5 to x and the value “abc” to y. Let T be a set of *predicate names* with typical instances p, q, r. Each predicate name p is associated with some domain $domain(p)$. (Notice that *domain* is used both with a predicate name and with a variable.) A predicate is constructed from a predicate name and a variable or a constant of the same domain. 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 $p(x)$. Similarly, if q and y are associated with the domain of integers, then we can have the predicates $q(3)$ and $q(y)$. Predicates over constants are called *ground predicates*. A *state* is a finite set of ground predicates, also referred to as *events*, where each predicate name may appear at most once. We occasionally refer to a state containing only one ground predicate (event) as an event. An *execution* $\sigma = s_1 s_2 \ldots$ (observed at any time) is a finite sequence of *states*. For example, if $T = \{p, q, r\}$, then $\{p(\text{“xyzzy”}), q(3)\}$ is a possible state. At any point in time during monitoring we have only observed a finite trace, consisting of a prefix of an execution, but there is no a priori limit on the length of the input trace.

Syntax. The formulas of the core QTL logic are defined by the following grammar, where a is a constant representing a value in $domain(p)$. 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, including zero arguments, which correspond to propositions.

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid p(a) \mid p(x) \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid \neg\varphi \mid (\varphi \; S \; \varphi) \mid \ominus \varphi \mid \exists x \, \varphi \mid \forall x \, \varphi$$

At a given state the formula $p(\text{“a”})$ means that $p(\text{“a”})$ happened, that is, $p(\text{“a”})$ is among the ground predicates of the state. Consider now the formula $p(x)$, for a variable $x \in V$. We interpret it such that x is assigned any value “a” where $p(\text{“a”})$ appears in the current state. Thus, for interpreting $(p(x) \wedge q(y))$ in a state that has the predicates $p(\text{“a”})$ and $q(3)$, we have the assignment $[x \mapsto \text{“a”}, y \mapsto 3]$. The formula $(\varphi_1 \; S \; \varphi_2)$ (reads φ_1 *since* φ_2) means that φ_2 occurred in the past (including now) and since then (beyond that state) φ_1 has been true. This is the past dual of the commonly used future time *until* modality [23]. The property $\ominus \varphi$ means that φ is true in the previous state. This is the past dual of the future time *next* modality. We can also define the following additional temporal operators: $P \varphi = (\mathit{true} \, S \, \varphi)$ (“previously”), and $H\varphi = \neg P \neg \varphi$ (“always in the past” or “historically”). The operator $[\varphi_1, \varphi_2)$, borrowed from [21], has the same meaning as $(\neg\varphi_2 \; S \; \varphi_1)$, but reads more naturally as a semi-open interval.

We present the semantics of QTL, formulated in two alternative ways. First using predicates on variable assignments, and subsequently using sets of such assignments. In Section 3 an algorithm that encodes such sets of assignments as BDDs is introduced.

Semantics. Let γ be an assignment to the variables that appear free in a formula φ. Then $(\gamma, \sigma, i) \models \varphi$ if φ holds for the prefix $s_1 s_2 \ldots s_i$ of the trace σ with the assignment γ. This is a standard definition, agreeing, e.g., with [7]. Note that by using past operators, the semantics is not affected by states s_j for $j > i$. Let $vars(\varphi)$ be the

set of free (i.e., unquantified) variables of a subformula φ. We denote by $\gamma|_{vars(\varphi)}$ the restriction (projection) of an assignment γ to the free variables appearing in φ. Let ε be the empty assignment. In any of the following cases, $(\gamma,\sigma,i) \models \varphi$ is defined when γ is an assignment over $vars(\varphi)$, and $i \geq 1$.

- $(\varepsilon,\sigma,i) \models true$.
- $(\varepsilon,\sigma,i) \models p(a)$ if $p(a) \in \sigma[i]$.
- $([v \mapsto a],\sigma,i) \models p(v)$ if $p(a) \in \sigma[i]$.
- $(\gamma,\sigma,i) \models (\varphi \wedge \psi)$ if $(\gamma|_{vars(\varphi)},\sigma,i) \models \varphi$ and $(\gamma|_{vars(\psi)},\sigma,i) \models \psi$.
- $(\gamma,\sigma,i) \models \neg\varphi$ if not $(\gamma,\sigma,i) \models \varphi$.
- $(\gamma,\sigma,i) \models (\varphi\ S\ \psi)$ if for some $1 \leq j \leq i$, $(\gamma|_{vars(\psi)},\sigma,j) \models \psi$ and for all $j < k \leq i$, $(\gamma|_{vars(\varphi)},\sigma,k) \models \varphi$.
- $(\gamma,\sigma,i) \models \ominus\varphi$ if $i > 1$ and $(\gamma,\sigma,i-1) \models \varphi$.
- $(\gamma,\sigma,i) \models \exists x\ \varphi$ if there exists $a \in domain(x)$ such that[2] $(\gamma[x \mapsto a],\sigma,i) \models \varphi$.

The definition of the *since* operator S can be simplified in a standard way such that it refers only to the positions i and $i-1$ in the sequence σ. This is based on the fact that according to the semantics of *since*, $(\varphi S \psi) = (\psi \vee (\varphi \wedge \ominus(\varphi S \psi)))$. This will serve in the implementation to work with only two versions of the sets of assignments, for the current and previous state:

- $(\gamma,\sigma,i) \models (\varphi S \psi)$ if $(\gamma|_{vars(\psi)},\sigma,i) \models \psi$ or $i > 1$, $(\gamma|_{vars(\varphi)},\sigma,i) \models \varphi$, and $(\gamma,\sigma,i-1) \models (\varphi S \psi)$.

The rest of the operators are defined as syntactic sugar using the operators defined in the above semantic definitions: $false = \neg true$, $\forall x\ \varphi = \neg\exists x \neg\varphi$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$.

Set Semantics. We now refine the semantics of the logic. Under the new definition, $I[\varphi,\sigma,i]$ is a function that returns a set of assignments such that $\gamma \in I[\varphi,\sigma,i]$ iff $(\gamma,\sigma,i) \models \varphi$. This redefinition will later lead to a simple implementation using BDDs, where each set of assignments will be represented as a BDD, and the Boolean operators will correspond directly to Boolean operators on BDDs.

In order to deal with subformulas with different sets of free variables (hence, different domains for assignments), we apply a projection and an extension operator to assignments over a subset of the variables. Let Γ be a set of assignments over the variables W, and $U \subseteq W$. Then $hide(\Gamma,U)$ (for "projecting *out*" or "hiding" the variables U) is the largest set of assignments over $W \setminus U$, each agreeing with some assignment of Γ on all the variables in $W \setminus U$. Let $U \cap W = \emptyset$, then $ext(\Gamma,U)$ is the largest set of assignments over $W \cup U$, where each such assignment agrees with some assignment in Γ on the values assigned to the variables W. This means that we extend Γ by adding arbitrary values to the variables in U from their domains. Then $hide(ext(\Gamma,U),U) = \Gamma$ holds. 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, if Γ is a set of assignments over W and Γ' is a set of assignments over W', then $\Gamma \bigcup \Gamma'$ is defined as $ext(\Gamma,W' \setminus W) \cup ext(\Gamma',W \setminus W')$ and $\Gamma \bigcap \Gamma'$ is $ext(\Gamma,W' \setminus W) \cap ext(\Gamma',W \setminus W')$. Hence, both are defined over the set of variables $W \cup W'$.

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

We denote by $A_{vars(\varphi)}$ the set of all possible assignments of values to the variables that appear free in φ. Thus, $I[\varphi,\sigma,i] \subseteq A_{vars(\varphi)}$. To simplify definitions, we add a dummy position 0 for sequence σ (which starts with s_1), where every formula is interpreted as an empty set. Observe that the value $\emptyset$ and $\{\varepsilon\}$, behave as the Boolean constants 0 and 1, respectively. The set semantics is defined as follows, where $i \geq 1$.

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

As before, the interpretation for the rest of the operators can be obtained from the above using the connections between the operators, e.g., $I[P\varphi,\sigma,i] = I[(true\,S\,\varphi),\sigma,i]$. The correspondence between this set based semantics and the previous semantics, namely that $\gamma \in I[\varphi,\sigma,i]$ iff $(\gamma,\sigma,i) \models \varphi$ can be proved by a simple structural induction on the size of the formulas.

3 An efficient Algorithm using BDDs

3.1 Representation of Sets of Assignments as BDDs

Our last refinement is to represent sets of assignments using Ordered Binary Decision Diagrams (OBDDs, although we write simply BDDs) [9]. A BDD is a compact representation for a Boolean tree, corresponding to a Boolean function. Because of compaction, however, the BDD forms a directed acyclic graph rather than a tree, see Figure 1. Each internal node is marked with a Boolean variable, which we also call a *BDD bit* or simply a *bit*. One edge, drawn as a dotted arrow from a node, represents that this bit has the Boolean value 0, while another edge, drawn as a full arrow, represents that it has the value 1. The nodes in the tree have the same order along all paths from the root, although some of the nodes may be missing, when the result of the Boolean function does not depend on the value of the corresponding bit. The leaves have the Boolean values 0 and 1. Thus, following a path in this graph, moving along dotted or fully drawn arrows, corresponding to evaluating the values 0 or 1 respectively for the BDD bits from which the edges emanate, leads to a leaf node that is marked by either a 0 or 1. This leaf value is the Boolean value returned by the function represented by the BDD for the assignment of 0s and 1s to the Boolean bits on this path. The graph is compacted in such a way that isomorphic subtrees are "glued" together. In addition, instead of keeping a node b with both edges that lead to the same subgraph, the node and its outgoing edges are removed from graph representation of the BDD, and consequently incoming edges are redirected to the successor node. This means that for the Boolean values on the prefix of the path so

far, the BDD value does not depend on the value of b. This compaction can be quite significant. BDDs have been instrumental in achieving a tremendous improvement in the size of systems that can be automatically verified [11].

When a new value for some variable appears in a ground predicate in the current state, we add it to a list of values of that variable that were seen. In order to search efficiently if this value already appeared, in time close to linear in the length of its representation, we can use a hash structure (we use a hash table as explained below). Thus, if we see p("ab"), p("de"), p("af") and q("fg") in subsequent states, where p and q are over the domain of strings, then we obtain a list of values ["ab","de","af","fg"]. We use BDDs to represent sets of such values. Each new value that appears in the monitored sequence is enumerated as a binary number, according to the order in which they appear in the input. The BDDs are Boolean representations of sets of these binary encodings, rather than a direct representation of the actual observed values. This allows us to use less BDD bits.

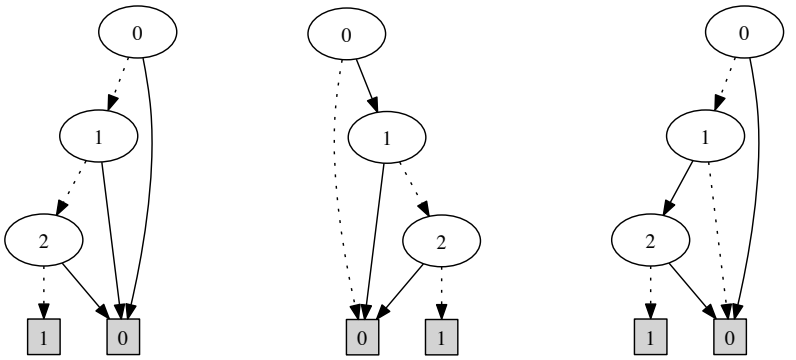
Thus, using three bits, "ab" can be represented as the bit string 000 (we start to enumerate from 000), "de" as 001, "af" as 010 and "fg" as 011. A BDD returns a 1 for each bit string representing an enumeration of a value in the set, and 0 otherwise. Then a BDD for a set containing the values "de" and "af" (2nd and 3rd values) will return 1 for 001 and 010. If the Boolean function is over b_0 (for least significant bit), b_1 and b_2 (for most significant), then this is the Boolean function $\neg b_2 \wedge ((\neg b_1 \wedge b_0) \vee (b_1 \wedge \neg b_0))$. Figure 1 shows the BDDs for each of these values as well as the BDD for the set containing the values "de" and "af". The least significant bit is denoted by BDD bit 0, and the most significant bit in this case with BDD bit 2.

We can now represent *sets of assignments to variables* as required by our set semantics. We use a partition of the BDD bits according to the variables. Say, we want to represent a set S of assignments to the variables x and y, each expected to have no more than 7 values. Then we can use the bits $y_2\,y_1\,y_0\,x_2\,x_1\,x_0$, where x_0, x_1 and x_2 represent the enumerations of values of x, and y_0, y_1 and y_2 represent the enumerations of values of y. The BDD over these 6 bits will return 1 for each pair of enumerations that represent an assignment of values to x and y in the set S. This representation is illustrated with an example in Section 4.1.

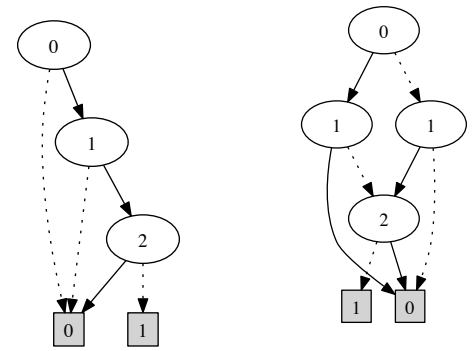
A subset of a set of k values can be represented as a function from a binary representation using $\lceil \log_2(k+1) \rceil$ bits to 0 or 1. (As explained later, we reserve one enumeration for representing values that were not seen so far.) It can be represented as a Boolean tree of size $O(k)$. If we have m variables, $z^1, \ldots z^m$, where the number of values from the domain of the variable z^i is of size k_i, then we can represent any encoding of an assignment to the m variables with $\Sigma_{i=1..m} \lceil \log_2(k_i+1) \rceil$ bits. With this number of bits, the BDD graph can grow up to size $O(\Pi_{i=1..m} k_i)$. However, representing this function as a BDD can often be quite more compact.

3.2 The Algorithm

Given some value a observed in the trace as an argument to a ground predicate, let **lookup**(a) return a bit string that represents the enumeration assigned to a. First **lookup**(a) will check in the hash table whether a already appeared in the input. If so,



(a) BDD for “ab”: 000 (b) BDD for “de”: 001 (c) BDD for “af”: 010



(d) BDD for “fg”: 011 (e) BDD for the set {“de”, “af”}: 001 or 010

Fig. 1: BDDs for the observed values “ab”, “de”, “af”, “fg” in the trace: p(“ab”).p(“de”).p(“af”).q(“fg”).

the hash table will be used to return its assigned enumeration. If a appears in this state for the first time, **lookup**(a) will assign to it a new enumeration, and add to the hash table a link from a to this enumeration value. We can use a counter (per variable) to enumerate in ascending order, incrementing the counter each time a new value for that variable is seen, and use the binary representation of the counter value as the new enumeration.

The function **build**(x,A) returns a BDD that represents the set of assignments where x is mapped to (the enumeration of) v for $v \in A$. This BDD is independent of the values assigned to any variable other than x, i.e., they can have any value. For example, assume that we use three bits x_0, x_1 and x_2 for representing enumer-

ations 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 $\neg x_2 \wedge x_0$.

Union and intersection of sets of assignments are translated simply into disjunction and conjunction of their BDDs 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, as in the interpretation of $\exists x\ \varphi$, we use the BDD existential (universal, respectively) operator over the bits that represent (the enumeration of) values of x. Thus, we translate $\exists x$, where x is represented using the bits x_0, $x_1, \ldots x_{k-1}$ into $\exists x_0 \ldots \exists x_{k-1}$. We denote by $\mathbf{exists}(\langle x_0, \ldots, x_{k-1}\rangle, bdd)$ the BDD function to perform existential quantification over the bits $x_0 \ldots x_{k-1}$. Finally, BDD(0) and BDD(1) are the BDDs that always return 0 or 1, respectively.

The algorithm uses standard BDD operators, and is almost a direct translation of the semantics using sets of assignments. The structure of the algorithm is similar to that of [18]. Namely, there are only two vectors (arrays) of values indexed by subformulas: for the current state (now) and for the previous state (pre). However, while in [18] the vectors contain Boolean values, here the vectors contain BDDs. The algorithm follows.

1. Initially, for each subformula φ, $\mathsf{now}(\varphi) = \text{BDD}(0)$.
2. Observe a new state (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 $\mathsf{now}(\psi)$.
   - $\mathsf{now}(\mathit{true}) = \text{BDD}(1)$.
   - $\mathsf{now}(p(a)) =$ if $p(a) \in s$ then BDD(1) else BDD(0).
   - $\mathsf{now}(p(x)) = \mathbf{build}(x,A)$ where $A = \{a \mid p(a) \in s\}$.
   - $\mathsf{now}((\varphi \wedge \psi)) = \mathbf{and}(\mathsf{now}(\varphi), \mathsf{now}(\psi))$.
   - $\mathsf{now}(\neg\varphi) = \mathbf{not}(\mathsf{now}(\varphi))$.
   - $\mathsf{now}((\varphi\ S\ \psi)) = \mathbf{or}(\mathsf{now}(\psi), \mathbf{and}(\mathsf{now}(\varphi), \mathsf{pre}((\varphi\ S\ \psi))))$.
   - $\mathsf{now}(\ominus\ \varphi) = \mathsf{pre}(\varphi)$.
   - $\mathsf{now}(\exists x\ \varphi) = \mathbf{exists}(\langle x_0, \ldots, x_{k-1}\rangle, \mathsf{now}(\varphi))$.
5. Goto step 2.

We select the number of BDD bits k per variable to be large enough such that no more than 2^k different values are anticipated. For example, if $k = 20$, this will allow more than a million different values.

For infinite domains, we maintain an enumeration that represents all the values that were not seen so far in the input. To see why this is necessary, consider the case where *all* 2^k enumerations are used (i.e., they were seen in the execution so far) for predicate $g(x)$. Then $Pg(x)$ will be represented as BDD(1), returning constantly a 1. Thus, $\neg Pg(x)$ will be calculated to BDD(0) (*false*). Now, $\exists x \neg Pg(x)$ will be translated into $\exists x_0 \exists x_1 \ldots \exists x_{k-1}BDD(0)$, and will also return BDD(0). However, checking $\exists x \neg Pg(x)$ should have returned BDD(1) (*true*), since it states that there are values that did not occur so far within a g predicate; indeed, for an infinite domain we could have never seen all the possible values during a finite execution.

We achieve that there is at least one value representing all the values not seen so far by reserving for this purpose the largest possible enumeration $11\ldots11$. Thus, if we use k bits, we do not allow enumerating beyond 2^k-1. The algorithm presented above will preserve the following invariant: if we did not use the enumerations between some m and 2^k, then all these enumerations are related to the other enumerations in all BDDs representing subformulas as representing values not seen before. This can be shown by a simple induction on the length of temporal formulas and the input sequence. This trivially holds for the subformulas *true*, $p(a)$ and $p(x)$, and is preserved by the application of the logical operations, quantification and also the updates performed upon the arrival of new states.

3.3 Extensions

Dynamic Expansion of the BDDs. In case we did not allocate in advance enough bits, it is possible to extend the number of bits we use for representing values for a variable. As explained above, the enumeration $11\ldots11$ of length k represents for every variable "all the values not seen so far in input the 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 $11\ldots11$.

Suppose we have 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, and 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 may depend on the other bits, representing the x and z variables. Thus, to expand the BDD, we generate a new one as follows:

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

The generalization of this formula to any number of variables is clear.

Finite Domains. We now show how to deal with the case of variables that are defined over finite domains. Say we have a BDD over enumerations of variables x, y and z, where y has a domain of size m. Then we need $k = \lceil \log_2(m) \rceil$ bits, $y_0, \ldots y_{k-1}$, for representing y. We need to relativize the use of existential quantifier to m. For finite domains we do not need to reserve a special enumeration for "all other values". We can encode a fixed BDD function $smaller(y,t)$ that expresses that the bits that represent y have a binary value that is *smaller* than the binary representation of t. Assuming that we start enumerating from $00\ldots00$, we check *smaller* rather than *smaller or equal*). For example, if we use two BDD bits y_0 and y_1 then

$smaller(y,3) = \neg(y_0 \wedge y_1)$, as any binary number smaller than 3 will have at least $y_0 = 0$ or $y_1 = 0$. Now, we need to replace each subformula of the form $\exists x\, \varphi$ where x appears free in φ by $\exists x\, (smaller(x,m) \wedge \varphi)$. This limits the quantification on the bits that represent x to values that are in the finite domain with m values. We implement universal quantification $\forall x\, \varphi$ using negation (twice) and existential quantification, obtaining $\forall x\, (smaller(x,m) \rightarrow \varphi)$.

Quantifying over Values Seen so Far. We can also extend our logic with a construct $seen(y)$ for a variable y. This construct will be translated, in a similar way as explained in the previous paragraph, into a BDD that encodes the binary values of the bits representing y that are no bigger than the maximal enumeration used (seen) so far for the variable y. We saw earlier that $\exists x\, \neg P\, g(x)$ should always return *true* in an infinite domain, as it says that there is a value in the domain of x that did not appear within a g predicate name. However, we may intend to mean that the existential quantification is restricted to be only over the values that were seen. In this case, we can write $\exists x\, (seen(x) \wedge \neg P\, g(x))$. This can be *true* if there is some value for the variable x that appeared in the execution so far within a predicate name other than g, but not within g.

3.4 A Characterization of Sets of Assignments for QTL

We describe a mathematical representation of the sets of assignments that satisfy (sub)formulas of QTL. It provides an additional intuition and motivation for the BDD representation. This subsection can be safely skipped without affecting readability of the rest of the paper.

Consider as an example the QTL property $\eta = \mathbf{P}\, (q(x) \vee r(y))$, where the domain of both x and y are the naturals $\mathbb{N}$, after observing a sequence of states $\rho = \{q(1)\}.\{r(3)\}$. The infinite set of satisfying assignments is denoted by $\{[x \mapsto u, y \mapsto v] \mid u = 1 \vee v = 3\}$.

Definition 1 Let Δ_i, for $1 \leq i \leq n$ be a finite subset of the domain D_i. Then $D_{\Delta_1,\ldots,\Delta_n} = \Delta_1 \cup \{\infty\} \times \ldots \times \Delta_n \cup \{\infty\}$. A *bounded* semi-finite set $\mathcal{S}_{\Delta_1,\ldots,\Delta_n}$ is a subset of $D_{\Delta_1,\ldots,\Delta_n}$. The symbol ∞ in the ith position of a vector in $D_{\Delta_1,\ldots,\Delta_n}$ corresponds to any elements in $D_i \setminus \Delta_i$. Accordingly, define $elem(\mathcal{S}_{\Delta_1,\ldots,\Delta_n})$ as $\{[v_1,\ldots,v_n] \mid \exists [w_1,\ldots,w_n] \in \mathcal{S}_{\Delta_1,\ldots,\Delta_n}\ s.t.\ for\ 1 \leq i \leq n, (w_i \neq \infty \rightarrow v_i = w_i) \wedge (w_i = \infty \rightarrow v_i \in D_i \setminus \Delta_i)\}$; this is the set of vectors over $D_1 \times \ldots \times D_n$ that is represented by $\mathcal{S}_{\Delta_1,\ldots,\Delta_n}$.

We can represent the set of assignments satisfying η after the input ρ as the bounded semi-infinite set $\{[1,\infty],[1,3],[\infty,3]\}_{\{1\},\{3\}}$ and the assignments satisfying $\neg\eta$ after ρ as $\{[\infty,\infty]\}_{\{1\},\{3\}}$.

Lemma 1 *For each* $\sigma_1, \sigma_2 \in D_{\Delta_1,\ldots,\Delta_n} = \Delta_1 \cup \{\infty\} \times \ldots \times \Delta_n \cup \{\infty\}$, *either* $elem(\{\sigma_1\}) = elem(\{\sigma_2\})$ *or* $elem(\{\sigma_1\}) \cap elem(\{\sigma_2\}) = \emptyset$.

Proof. Because for bounded semi finite sets the values represented by ∞ from D_i and the values in Δ_i are disjoint. □

Theorem 1 *Applying the set operators $\cup$ (union), $\cap$ (intersection) and complementation on Bounded semi-finite sets correspond to applying the same operators on their sets of elements.*

Proof. It follows from Lemma 1 that $elem(S_{\Delta_1,\ldots,\Delta_n} \cap S'_{\Delta_1,\ldots,\Delta_n}) = elem(S_{\Delta_1,\ldots,\Delta_n}) \cap elem(S'_{\Delta_1,\ldots,\Delta_n})$ and $elem(S_{\Delta_1,\ldots,\Delta_n} \cup S'_{\Delta_1,\ldots,\Delta_n}) = elem(S_{\Delta_1,\ldots,\Delta_n}) \cup elem(S'_{\Delta_1,\ldots,\Delta_n})$. For complementation, $\overline{\{[v_1,\ldots,v_n]\}}$ is the finite set $\{[w_1,\ldots,w_n] \mid \exists i\, 1 \leq i \leq n, \quad w_i \in ((\Delta_i \cup \{\infty\}) \setminus \{v_i\})\}$, with both sets representing the same set of elements from $D_{\Delta_1,\ldots,\Delta_n}$. Then use the fact that $\overline{\{\sigma_1,\ldots\sigma_n\}} = \overline{\{\sigma_1\}} \cup \ldots \cup \overline{\{\sigma_n\}} = \overline{\{\sigma_1\}} \cap \ldots \cap \overline{\{\sigma_n\}}$. □

The bounds for the semi-finite sets used to interpret a QTL formula over a given input is a set of values observed in the input so far per each variable. With each new state, one of the sets Δ_i can grow, while the other sets remain the same.

Lemma 2 *Bounded semi-finite sets are closed under extending any set Δ_i by an element of $D_i \setminus \Delta_i$.*

Proof. Let $w \notin \Delta_i$. Then $\mathcal{S}_{\Delta_1,\ldots,\Delta_{i-1},\Delta_i \cup \{w\},\Delta_{i+1},\ldots\Delta_n} = \mathcal{S}_{\Delta_1,\ldots,\Delta_{i-1},\Delta_i,\Delta_{i+1},\ldots,\Delta_n} \cup \{[v_1,\ldots,v_{i-1},w,v_{i+1},\ldots v_n] \mid \{[v_1,\ldots,v_{i-1},\infty,v_{i+1},\ldots v_n] \in \mathcal{S}_{\Delta_1,\ldots,\Delta_{i-1},\Delta_i,\Delta_{i+1},\ldots,\Delta_n}\}$. □

Note that if we extend Δ_i with a new value w, then ∞ appearing in the ith component of a vector does not represent anymore the value w.

For the above sequence $\rho = \{q(1)\}.\{r(3)\}$, we start with $\Delta_1 = \Delta_2 = \emptyset$. Then, after observing the value 1 in $q(1)$ (for the variable x), we have $\Delta_1 = \{1\}$ and $\Delta_2 = \emptyset$, and after observing the value 3 in $r(3)$ for y, we have $\Delta_1 = \{1\}$ and $\Delta_2 = \{3\}$.

Theorem 2 *The set of assignments that satisfy a* QTL *formula φ over the variables $x_1 : D_1, \ldots x_n : D_n$ after a finite sequence of states can be represented as a bounded semi-finite set $\mathcal{S}_{\Delta_1,\ldots,\Delta_n}$.*

Proof. Using induction on the length of the input and on the size of the formula. For an empty input, all bounded semi-infinite sets are empty, and the bounds are also empty. Consider an event $p(a)$ that appears in the input (this can be easily generalized to an event with multiple arguments). We show how to calculate the bounded semi-finite sets representing the assignments for a given subformula in several cases:

- For $p(x_i)$, where $x_i : D_i$, the set of assignments is represented using the following bounded semi-finite set: $\{[v_1,\ldots,v_{i-1},a,v_{u+1},\ldots v_n] \mid \textit{ for } 1 \leq j \leq n \textit{ s.t. } j \neq i : v_j \in \Delta_j \cup \{\infty\}\}$. If a appears for the first time in a ground predicate, then Δ_i is expanded first to include a.
- For $\ominus\varphi$, the assignments after the current nonempty input is the same as the assignments for φ before the last event.
- For $(\varphi \vee \psi)$ we take the union of the corresponding semi-finite sets for φ and ψ after the current input. Intersection is handled similarly.
- For $\neg\varphi$, we take the complementation of the bounded semi-finite set representing φ.
- For $\exists x\varphi$, $x : D_i$, where $\mathcal{S}_{\Delta_1,\ldots,\Delta_n}$ represents the set of assignments satisfying φ, we obtain $\{[v_1,\ldots,v_{i-1},w,v_{i+1},\ldots v_n] \mid [v_1,\ldots,v_i,\ldots v_n] \in \mathcal{S}_{\Delta_1,\ldots,\Delta_n} \wedge w \in \Delta_i \cup \{\infty\}\}$.

□

To represent bounded semi-finite sets using BDDs, we can assign a fixed number of bits per each component domain D_i in the product domain. Encoding values using a fixed number of bits requires to give an a priori bound on the size of values. We enumerate the different values, and keep a hash table that maps values to enumerations. This is in particular appealing for runtime verification, where we may not be given in advance the exact set of values that will appear in the monitored event stream, but perhaps a rough bound on the number of different values. Using enumerations instead of actual values for the BDD may also contribute to minimizing the BDD representation. The Boolean representation needs to reserve, per each component domain D_i, at least one (enumeration of a) value that represents ∞, i.e., all values in $D_i \setminus \Delta_i$. For that we use $11\ldots11$, as explained in section 3.2.

4 Implementation

We implemented a monitoring tool for the QTL logic, called DEJAVU. We assume that each state contains one[3] ground predicate, called an *event*. Let $\mathbb{E}$ be the type of events, and the $\mathbb{B}$ be the type of Boolean values. The implementation of the monitoring algorithm presented in Section 3 consists of a program *translate* : $Spec \rightarrow (\mathbb{E}^* \rightarrow \mathbb{B}^*)$, which, when provided a specification generates a *monitor* program; the monitor takes as input a trace, and returns a verdict, effectively a Boolean value per each new event in the trace. In the following we outline the format of the generated monitor program. The tool is implemented in SCALA, using the standard approach where a parser parses the specification and produces an abstract syntax tree, which is then traversed and translated into the monitor program. The parser is written using SCALA parser combinators. The generated monitor program uses the JavaBDD package [19] for generating and operating BDDs. Log files in CSV format are parsed using the Apache Commons CSV (Comma Separated Value format) parser. The tool can be used for online monitoring (observing a program as it executes) as well as for offline monitoring (analyzing log files). We shall illustrate the monitor generation using an example. Consider the following variation of the first property from Section 1 (using syntax supported by the implementation):

```
prop p: forall f . close(f) → exists m . P open(f,m)
```

It states that if a file f is closed, it should have been opened in the past with some access mode (*read*, *write*, ...). The generated monitor relies on an enumeration of the subformulas of the original formula in order to evaluate the subformulas bottom up for each new event. Figure 2 (right) shows the decomposition of the original formula into subformulas (an Abstract Syntax Tree - AST), indexed by numbers from 0 to 5, satisfying the invariant that if a formula φ_1 is a subformula of a formula φ_2 then φ_1's index is bigger than φ_2's index. The monitor generated from the property is shown in Figure 2 (left). Specifically two arrays are declared, indexed by subformula indexes: pre for the previous state and now for the current state. As previously explained, a

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

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

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

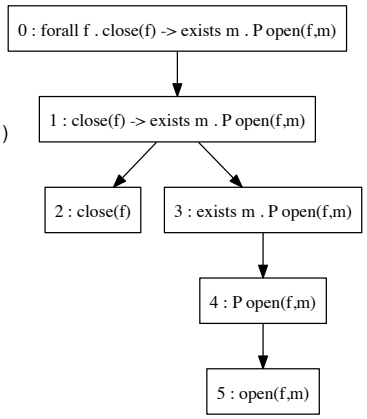


Fig. 2: Monitor (left) and AST (right) for the property.

BDD represents a predicate on bit strings, effectively representing a set of bit strings (those for which the BDD evaluates to true). Actual values in the trace are uniquely mapped to such bit strings, and the BDD therefore indirectly represents a set (set membership function) of the actual values.

In each step the evaluate function re-computes the now array from highest to lowest index, and returns true (ok) iff now(0) is not BDD(0). Assume for example that an event *close*(out) is observed. At the leaf node 2 representing the close(f) event, the function call build("close")(V("f")) builds a new BDD for out unless one has previously been computed, in which case that is used. At composite subformula nodes, BDD operators are applied. For example for subformula 4, the new value is now(5).or(pre(4)), which is the interpretation of the formula **P** open(f,m) according to the relation $\mathbf{P}\,\varphi = (\varphi \vee \ominus\, \mathbf{P}\,\varphi)$. Quantification is solved by performing quantification over the relevant BDD bits corresponding to the variable in question.

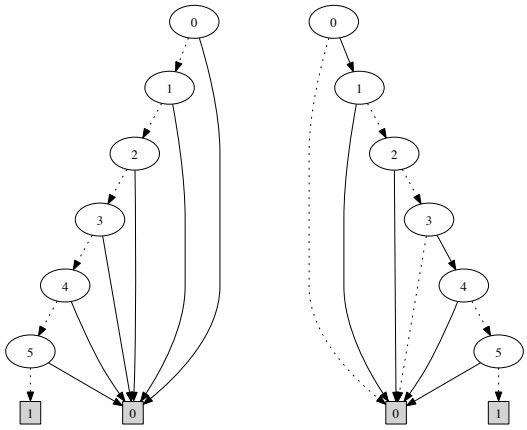
4.1 Example Monitor Execution

We shall briefly evaluate the example formula on a trace. Assume that each variable f and m is represented by three BDD bits. Consider the input trace, consisting of three events[4]:

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

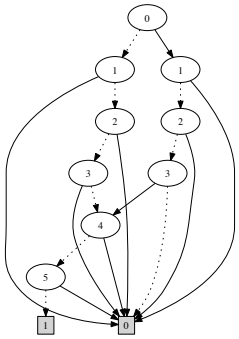
When the monitor evaluates subformula 5 (subformulas here are numbered according to Figure 2) on the first event $\mathit{open}(\mathit{input}, \mathit{read})$, it will create a bit string composed

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

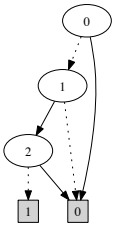


(a) B_1 @ 5 and 4
after *open*(input,read)

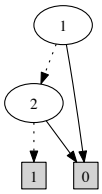
(b) B_2 @ 5
after *open*(output,write)



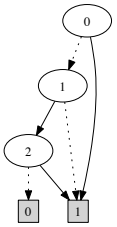
(c) B_3 @ 4
after *open*(output,write)



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



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



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

Fig. 3: Selected BDDs, named $B_1, \ldots, B_6$, computed after each event at various subformula nodes, indicated by BDD B_i @ *node* (see Figure 2), during processing of the trace: *open*(input,read).*open*(output,write).*close*(out).

of a bit string for each variable f and m. As previously explained, bit strings for each variable are allocated in increasing order: 000, 001, 010,..., hence the first bit string representing the assignment [f $\mapsto$ input, m $\mapsto$ read] becomes 000000 where the three rightmost bits represent the assignment of input to f, and the three leftmost bits represent the assignment of read to m. Figure 3a shows the corresponding BDD B_1. Recall that most significant bits are implemented lower in the BDD, and that for each bit (node) in the BDD, the dotted arrow corresponds to this bit being 0 and the full drawn arrow corresponds to this bit being 1. In this BDD all bits have to be zero in order to be accepted by the function represented by the BDD. We will not show how all the tree nodes evaluate, except observe that node 5 assumes the same BDD value

as node 4 (all the seen values in the past), and conclude that since no *close*(...) event has been observed, the top-level formula (node 0) is true at this position in the trace.

Upon the second *open*(output,write) event, new values (output,write) are observed as arguments to the *open* event. Hence a new bit string for each variable f and m is allocated, in both cases 001 (the next unused bit string for each variable). The new combined bit string for the assignments satisfying subformula 5 then becomes 001001, forming a BDD representing the assignment [f $\mapsto$ output, m $\mapsto$ write], and appearing in Figure 3b as B_2. The computation of the BDD for node 4 is computed by now(4) = now(5).or(pre(4)), which results in the BDD B_3, representing the set of the two so far observed assignments (B_3 = **or**(B_1,B_2)).

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

5 Evaluation

DejaVu's performance is evaluated by comparing against MonPoly, the tool that seems to have most similarities to DejaVu as previously discussed. We specifically evaluated six temporal properties, formalized in QTL in Figure 4, on different sizes of traces, while for DejaVu varying the number of bits allocated to represent variables in BDDs. The properties were encoded in MonPoly in a 1-1 manner. These properties have the following intuitive meaning. The Access property states that if a file f is accessed by a user u, then the user should have logged in and not yet logged out, and the file should have been opened and not yet closed. The File property states that if a file is closed, then it must have been opened (and not yet closed) with some mode m (e.g. read or write). The Fifo property is a conjunction of four subformulas about data entering and exiting a queue. The first two subformulas state that a datum can at most enter and exit once. The third subformula states that a datum can only exit if it has previously been entered. The last subformula states the Fifo principle of queues.

The next three properties concern the acquisition and release of locks by concurrently executing threads. The Locking property concerns the safe use of locks and is composed of three subformulas. The first subformula states that a thread going to sleep must have released all acquired locks before then. The second subformula states that if a thread acquires a lock, no thread may prior have acquired the lock and not yet released it. The third subformula states that a thread cannot release a lock without having acquired it and not yet released it. The Deadlock property states that any two threads are not allowed to acquire any two locks in opposite order. That is, if a thread t_1 acquires a lock l_1, and then before releasing it, acquires a lock l_2, then an-

```
prop access :
  forall u . forall f . access(u,f) → [login(u),logout(u)) & [open(f),close(f))

prop file :
  forall f . close(f) → exists m . @ [open(f,m),close(f))

prop fifo :
  forall x .
    (enter(x) → ! @ P enter(x)) &
    (exit(x) → ! @ P exit(x)) &
    (exit(x) → @ P enter(x)) &
    (forall y . (exit(y) & P(enter(y) & @ P enter(x))) →
      @ P exit(x))

prop locking :
  forall t . forall l .
    (
      (sleep(t) → ![acq(t,l),rel(t,l))) &
      (acq(t,l) → ! exists s . @ [acq(s,l),rel(s,l))) &
      (rel(t,l) → @ [acq(t,l),rel(t,l)))
    )

prop deadlock :
  forall t1 . forall t2 . forall l1 . forall l2 .
    (@ [acq(t1,l1),rel(t1,l1)) & acq(t1,l2)) →
      (! @ P (@ [acq(t2,l2),rel(t2,l2)) & acq(t2,l1)))

prop datarace :
  forall t1 . forall t2 . forall x .
    ((P (read(t1,x) | write(t1,x))) & (P write(t2,x))) →
      exists l .
        (H ((read(t1,x) | write(t1,x)) → [acq(t1,l),rel(t1,l))) &
         H ((read(t2,x) | write(t2,x)) → [acq(t2,l),rel(t2,l))))
```

Fig. 4: Evaluation properties in DEJAVU's QTL logic.

other thread t_2 is not allowed to first acquire l_2 and then, before releasing it, acquire l_1. Following this discipline prevents cyclic deadlocks. Note that a violation of this property in a trace only indicates that the monitored application has a *potential* for deadlocking, not necessarily an actual deadlock. See [8] for a graph algorithm that detects such deadlock potentials in the general case of N threads and N locks. Finally, the DATARACE property similarly captures data race potentials. A data race occurs when two threads access (read or write) the same shared variable simultaneously, and at least one of the threads writes to the variable. The property states that in this case there must exist a lock, which both threads hold whenever they access the variable. See [26] for an algorithm that detects potential data races.

Table 1 shows the results of the evaluation, which in part were performed on a Mac laptop, with the Mac OS X 10.10.5 operating system, on a 2.8 GHz Intel Core

Property	Trace length	MONPOLY	DEJAVU			
			min bits	20 bits	40 bits	60 bits
ACCESS	11,006	1.2s	**0.9s** [13]	**0.9s**	**1.0s**	**1.0s**
	110,006	6m12.9s	**2.1s** [16]	**2.0s**	**2.8s**	**3.1s**
	1,100,006	16h14m16s★	**15.5s** [19]	**14.1s**	**22.5s**	**31.8s**
FILE	11,004	35.1s	**0.9s** [13]	**0.9s**	**0.9s**	**0.9s**
	110,004	85m42.4s	**1.9s** [17]	**1.9s**	**1.9s**	**2.1s**
	1,100,004	DNF★	**8.6s** [20]	**8.6s**	**9.6s**	**10.8s**
FIFO	5,051	2m9.4s	**22.7s** [13]	2m22.7s	OOM	-
	10,101	14m28.8s	**4m22.9s** [14]	OOM	-	-
LOCKING	10,401	2.1s	**1.2s** [10]	**1.6s**	3.5s	23.5s
	105,001	**0.4s**	2.1s [13]	2.0s	2.1s	2.6s
	1,050,126	29.6s	**7.7s** [08]	**10.2s**	**14.8s**	**20.9s**
DEADLOCK	9,608	**1.1s**	1.8s [10]	7.0s	27.3s	54.3s
	100,008	6.9s	**1.5s** [13]	**1.6s**	**1.8s**	**1.7s**
	1,050,008	23.7s	**14.2s** [07]	47.7s	1m55.0s	4m1.1s
DATARACE	10,005	NMO	**1.0s** [06]	**1.2s**	**1.5s**	**1.8s**
	100,005	NMO	**2.1s** [10]	**2.6s**	**3.0s**	**4.5s**
	1,050,005	NMO	**6.7s** [09]	**6.8s**	**7.8s**	**8.5s**

Table 1: Evaluation of DEJAVU and MONPOLY. Time values are provided in minutes and seconds. For DEJAVU four time values are provided for each trace: the time using the minimal number of bits possible (number of bits in square brackets [...]), and for 20, 40 and 60 bits respectively. Numbers in bold indicate a tool being superior on the particular trace, and bit count in the case of DEJAVU.

```
FORALL t1 . FORALL t2 . FORALL x .
  ((( ONCE (read(t1,x) OR write(t1 ,x)))  AND (ONCE write(t2,x))) IMPLIES
    EXISTS l . (
      (PAST_ALWAYS ((read(t1,x) OR write(t1,x)) IMPLIES (NOT rel(t1,l) SINCE acq(t1,l))))
        AND
      (PAST_ALWAYS ((read(t2,x) OR write(t2,x)) IMPLIES (NOT rel(t2,l) SINCE acq(t2,l))))
    )
  )
```

Fig. 5: The DATARACE property in MONPOLY's logic.

i7 with 16 GB of memory. Two of the MONPOLY verifications, marked with ★, were performed on a desktop UNIX machine in order to allow for long uninterrupted execution times. The properties were evaluated with each tool on traces of sizes spanning from (approximately) 5 thousand to 1 million events (see table for exact numbers). Traces have the general form that initially numerous opening events (login, open, enter, acq) occur, in order to accumulate a large amount of data stored in the monitor, after which a smaller number of corresponding closing events (logout, close, exit, rel) occur. In addition, for each trace we applied DEJAVU with four different sizes of bit vectors: the minimal number of bits possible, 20 bits, 40 bits, and 60 bits, the latter three corresponding to the ability to store respectively approximately a million, a trillion, and a quintillion different values for each variable (the latter two are not needed for these traces). The following abbreviations are used: OOM = Out of Mem-

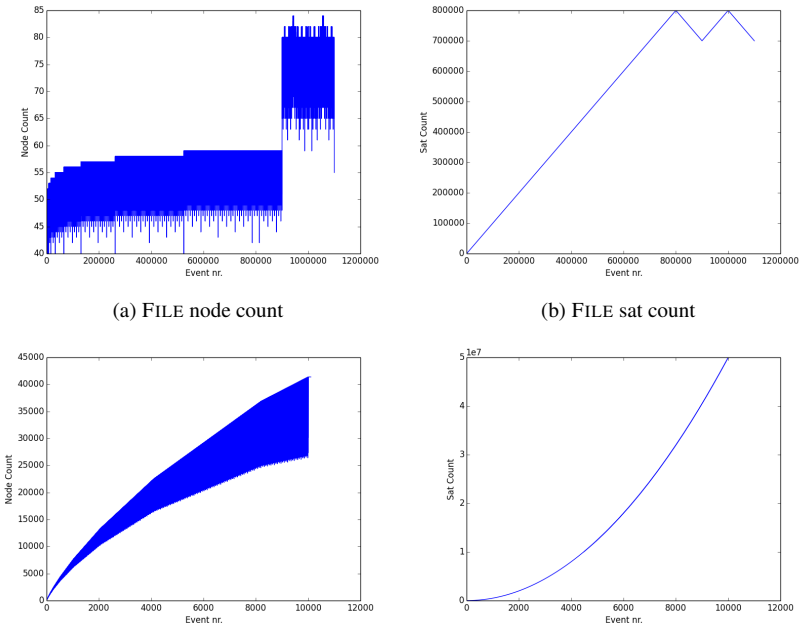
ory, DNF = Did Not Finish (during 16 hours), and NMO = Not Monitorable in the logic. Any number in bold font in the table indicates a monitor execution time for a tool that is less than the opponent tool's execution time for the same property and trace.

From Table 1 one can observe the following. For the first two properties ACCESS and FILE, DEJAVU clearly outperforms MONPOLY. For the ACCESS property on the shortest trace, the tools are comparable. However, for the two larger traces of the ACCESS property, DEJAVU is faster by factors 178 and 3,771 respectively, and for the two smaller traces of the FILE property, DEJAVU is faster by factors 39 and 2,707. For the FILE property the MONPOLY tool had not finished on the largest trace after 16 hours. For the FIFO property the picture depends on how many bits are allocated in DEJAVU for variables. When running DEJAVU with the minimum number of bits, DEJAVU outperforms MONPOLY with factors 3 to 6. However, when running DEJAVU with 20 bits, the results are similar for the shortest trace and on the largest trace DEJAVU runs out of memory, whereas MONPOLY finishes in over 14 minutes. This property is the most challenging of the properties examined. The complexity lies in the last of the four subformulas in the conjunction, the actual FIFO property. For the LOCKING and DEADLOCK properties, the results are mixed, with wins on each side with factors in the smaller end of the scale.

The DATARACE property, with its MONPOLY syntax version shown in Figure 5, stands out in the sense that it is not monitorable in MONPOLY due to need for this formula to compute infinite sets of assignments, caused by the occurrence of certain combinations of operators, such as e.g. **NOT** and **OR**. See [7,17] for an elaboration of monitorability of MONPOLY formulas. This illustrates the additional expressive power of DEJAVU caused by the use of BDDs as internal data structure, and in particular the use of the bit pattern $11\ldots11$ (all 1's) to represent the infinite set of all values not yet seen in the trace.

Increasing the number of bits allocated per variable in DEJAVU can have impact on execution times, in some cases more than in others. The average slow-down factor between using lowest number of bits and 60 bits is 5.7. However, in most cases the factor is 2 or less. The properties FIFO, LOCKING, and DEADLOCK show some higher slow-down factors up to 30. In the case of the FIFO property increasing the bit numbers cause out of memory exceptions. It is somewhat unfair to compare MONPOLY against DEJAVU using minimal bits. The minimal number of bits required by DEJAVU for analyzing a trace is unknown up front. However, even if one compares with 20+ bit results from DEJAVU, the overall result is largely the same, except for the FIFO property and the largest trace for the DEADLOCK property. In both cases the minimal number of bits are required in order to perform better than MONPOLY.

We further investigated our monitoring procedure by profiling various BDD parameters for the six properties. We specifically focused on two BDD parameters: the *node count* and the *sat count*. The node count denotes the number of nodes in a BDD and is the most important performance parameter as the complexity of operations on BDDs directly depends on the number of BDD nodes of the operand. On the other hand, the sat-count denotes the cardinality of the set of all satisfying assignments for the Boolean function that the BDD represents. The ratio of sat count and node count is often used to measure how much the BDD representation is compressing the



(a) FILE node count (b) FILE sat count

(c) FIFO node count (d) FIFO sat count

Fig. 6: BDD profiling (node and sat counts) for the FILE and FIFO properties.

underlying Boolean function. The charts in Figure 6 illustrate the evolution of these parameters during monitoring of the properties FILE and FIFO, for selected subformulas. These properties are cases where we performed well (FILE) and not-so-well (FIFO).

For the FILE property, the node count for the subformula [open(f,m),close(f)), which is equivalent to (!close(f) **S** open(f,m)), grows logarithmically as new open-events continuously arrive up to the $800,000^{th}$ event and stays constant around 60 nodes until the $900,000^{th}$ event. This suggests that the standard encoding and our enumeration scheme leads to very compact BDDs for one of the most common types of formulas even under a data-intensive input trace. For the FIFO property, however, we see that the sat count for the subformula (**P** (enter(y) & @ **P** enter(x))) quadratically grows as this subformula needs to store a set of assignments that contains a conjunction of each enter event seen so far and every enter event seen in its past. In other words, we need to represent a set isomorphic to the set $\{(i,j) \mid i < j \text{ and } i,j \in 0..N\}$ where N is the number of distinct enter events. Note that the number of satisfying assignments reaches 50 million $(10000^2/2)$ for $N = 10000$ in the plot. Although the BDD representation manages to compress such sets considerably, the amount of compression is simply not enough and the node count continues to grow in a linear fashion, reaching 40,000 nodes. This fact increasingly slows down the monitoring process and we will eventually run out the resources.

Overall, however, the results clearly demonstrate that BDDs are promising for representing observed data in runtime verification, augmenting efficiency of the monitoring algorithm as well as expressiveness of logic at the same time.

6 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 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 (free) in the monitored property. Each value observed in the trace is represented by a BDD that encodes the value's enumeration in appearance order. While the size of the BDD 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 redundantly large number of bits for representing the encoding of values, so that even extremely long executions can be monitorable. For example, if the encoding for each variable uses 64 bits, the BDD can hold up to 2^{64} different values for each variable. Alternatively, we showed how to dynamically expand the BDD when the number of values exhausts the allocated size.

Our experiments provide an optimistic view on the benefit of using BDDs. The implementation was written in SCALA, an object-oriented and functional programming language with active garbage collection. We expect that using a programming language such as C will result in even faster runtime verification monitors.

Acknowledgements We would like to thank Oded Maler for a discussion on representing sets using BDDs. We also thank Eugen Zălinescu for discussions concerning monitorability of MONPOLY formulas. Finally, we thank the reviewers of this article for their useful comments.

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 2005, 345-364.
2. B. Alpern, F. B. Schneider, Recognizing Safety and Liveness. Distributed Computing 2(3), 117-126, 1987.
3. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, Z. Manna: LOLA: Runtime Monitoring of Synchronous Systems, TIME 2005, 166-174.
4. H. Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-Based Runtime Verification, VMCAI, LNCS Volume 2937, Springer, 2004.
5. H. Barringer, K. Havelund, TraceContract: A Scala DSL for Trace Analysis, Proc. of the 17th International Symposium on Formal Methods (FM'11), LNCS Volume 6664, Springer, 2011.
6. H. Barringer, D. Rydeheard, K. Havelund, Rule Systems for Run-Time Monitoring: from Eagle to RuleR, Proc. of the 7th Int. Workshop on Runtime Verification (RV'07), LNCS Volume 4839, Springer, 2007.

7. D. A. Basin, F. Klaedtke, S. Müller, E. Zalinescu, Monitoring Metric First-Order Temporal Properties, Journal of the ACM 62(2), 45, 2015
8. S. Bensalem, K. Havelund, Dynamic Deadlock Analysis of Multi-threaded Programs, Haifa Verification Conference, Haifa, Israel, LNCS Volume 3875, Springer, 2006.
9. R. E. Bryant, Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, ACM Computing Survety 24(3), 293-318 (1992).
10. R. E. Bryant, On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication, IEEE Transactions on Computers 40(2): 205-213 (1991).
11. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, Symbolic Model Checking: 10^{20} States and Beyond, LICS 1990, 428-439.
12. N. Decker, M. Leucker, D. Thoma, Monitoring Modulo Theories, Journal of Software Tools for Technology Transfer, Volume 18, Number 2, 2016.
13. J. Goubault-Larrecq, J. Olivain, A Smell of ORCHIDS, Proc. of the 8th Int. Workshop on Runtime Verification (RV'08), LNCS Volume 5289, Springer, 2008.
14. S. Hallé, R. Villemaire, Runtime Enforcement of Web Service Message Contracts with Data, IEEE Transactions on Services Computing, Volume 5 Number 2, 2012.
15. K. Havelund, Rule-based Runtime Verification Revisited, Journal of Software Tools for Technology Transfer, Volume 17 Number 2, Springer, 2015.
16. K. Havelund, D. Peled, D. Ulus, First-Order Temporal Logic Monitoring with BDDs, 17th Conference on Formal Methods in Computer-Aided Design (FMCAD 2017), 2-6 October, 2017, Vienna, Austria, IEEE.
17. K. Havelund, G. Reger, D. Thoma, E. Zălinescu, Monitoring Events that Carry Data, book chapter in: Lectures on Runtime Verification - Introductory and Advanced Topics, book editors: Ezio Bartocci and Yliès Falcone, LNCS Volume 10457, Springer, 2018.
18. K. Havelund, G. Rosu, Synthesizing Monitors for Safety Properties, TACAS 2002, 342-356.
19. JavaBDD, http://javabdd.sourceforge.net.
20. J. G. Henriksen, J. L. Jensen, M. E. Jorgensen, N. Klarlund, R. Paige, T. Rauhe, A. Sandholm, Mona: Monadic Second-Order Logic in Practice, TACAS 1995, 89-110.
21. M. Kim, S. Kannan, I. Lee, O. Sokolsky, Java-MaC: a Run-time Assurance Tool for Java, Proc. of the 1st Int. Workshop on Runtime Verification (RV'01), Elsevier, ENTCS 55(2), 2001.
22. O. Kupferman, M. Y. Vardi, Model Checking of Safety Properties, Formal Methods in System Design 19(3): 291-314, 2001.
23. Z. Manna, A. Pnueli, Completing the Temporal Picture, Theoretical Computer Science 83, 91-130, 1991.
24. P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Rosu, An Overview of the MOP Runtime Verification Framework, J. Software Tools for Technology Transfer, Springer, 2011.
25. G. Reger, H. Cruz, D. Rydeheard, MarQ: Monitoring at Runtime with QEA, Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015), Springer, 2015.
26. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, Eraser: A Dynamic Data Race Detector for Multithreaded Programs, ACM Transactions on Computer Systems 15(4), 1997.
27. J. Whaley, D. Avots, M. Carbin, M. S. Lam, Using Datalog with Binary Decision Diagrams for Program Analysis, APLAS 2005, 97-118.