First Order Temporal Logic Monitoring with BDDs

Klaus Havelund
Jet Propulsion Laboratory,
California Inst. of Technology, USA

Doron Peled
Department of Computer Science
Bar Ilan University, Israel

Dogan Ulus
Verimag/Université Grenoble-Alpes
Grenoble, France

***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 so-called 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.**

I. 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, 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 [18]. Very early RV systems, were also based on specifications given in some form of propositional temporal logic. A propositional temporal logic formula can for example 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 so-called parametric specifications, where events carry data, and formulas can refer to such. Since such a monitor has to store observed data, the challenge is efficient representation; manipulation such as negation, conjunction, disjunction, quantification, and lookup. The field 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, one may only provide partial information about whether the property holds; namely, if it is already violated, already achieved, or undecided yet [18]. The focus of this work is past temporal properties, which are also classified as the safety temporal properties [2], [19], 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 [18]. 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 $\mathbf{P}$ is the "sometimes in the past" temporal operator. This property must be checked for every monitored event. Already in this very simple example we see that we need to store *all* the names of files that were previously opened so we can compare to the files that are being closed. A more refined specification would be the following, requiring that a file can be closed only if it was opened before, and has not been closed since. Here, we use the temporal operators $\ominus$ ("at previous step") and $\mathcal{S}$ ("since"):

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

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 $f = \text{“ab”}$. 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 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 $open(\text{“a”})$, $open(\text{“b”})$, $open(\text{“c”})$, 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_0 being the most significant bit). A BDD that represents the set of values $\{\text{“a”}, \text{“c”}\}$ 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:

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".

- It is highly compact. With k bits we can represent 2^k values. The BDD can grow up to a maximal number of 2^k+1 nodes; but BDDs usually compact the representation very well [9]. In fact, we are expected to pay very little for keeping surplus bits, as the BDD will compact away most of their effect. 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.
- Complementation (negation) is efficient, by just switching the 0 and 1 leaves of the BDD. Moreover, even though at any point we may have not seen the entire set of values that will show up during the execution, we can safely (and efficiently) perform complementation: values that have not appeared yet in the execution are being accounted for and their enumerations are reserved already in the BDD before these values appear.
- Our representation of sets of assignments as BDDs allows a very simple algorithm that naturally extends the dynamic programming monitoring algorithm for propositional past time temporal logic shown in [14].

We first define the semantics of a predicate linear temporal logic property for an assignment of values to its free variables after a given execution prefix. Then we redefine it as a function that returns the set of assignments satisfying the property at that prefix. There, we use the union and intersection set operators. For the final algorithm, we replace union and intersection by BDD disjunction and conjunction operators, respectively. 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 II discusses related work. Section III presents the syntax and semantics of the QTL temporal logic. Section IV presents the BDD-based algorithm for monitoring a trace against a QTL formula. Section V outlines the implementation, and Section VI presents an evaluation of the implementation. Finally, Section VII concludes the paper.

II. 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 the MonPoly system [7]. As in the current work, it monitors first order temporal properties. In fact, it is 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., [16] for a simple representation of sets of values). This version allows unrestricted complementation of sets of data values. Another version of MonPoly is based on representing finite sets of assignments. This is based on using algebraic database operators. For example, intersecting between two sets of assignments that are possibly over non identical sets of variables is done using the *join* operator. 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. These include TraceMatches [1], Mop [20], and Qea [21]. 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. Qea is an attempt to increase the expressiveness of the trace slicing approach. It is based on automata, as is the Orhcids system [11].

Other systems include BeepBeep [12] 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 [13] and Lola [3]. The system Mmt [10] represents assignments as constraints solved with an SMT solver. An encoding of BDD functions over enumerations of values appears in [22] in the context of datalog programs. However, that work does not deal with unbounded domains.

III. 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 for now that the domains are infinite, e.g., they can be the integers or strings. (In Section IV it is explained how to deal with finite domains.) Let V be a finite set of *variables*, with typical instances x, y, z. In an *assignment*, each variable x can be assigned a value from its associated domain $domain(x)$, where multiple variables (or all of them) can be related to the same domain. For example $[x \to 5, y \to \text{“abc”}]$ is an assignment of the values 5 and “abc” to the variables x and y, respectively. Let T 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 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 $p(x)$. Similarly, if q and y are associated with the domain of integers, then we can have the predicates $q(3)$ and $q(y)$. We refer to predicates over constants as *ground predicates*. A *state* is a finite set of ground predicates, where each predicate name may appear at most once. 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. Although during monitoring we always at any point in time only have observed a finite trace so far, the trace can grow unbounded, as the system being monitored keeps executing. In a monitoring context such as this, however, we will never observe an infinite trace.

Syntax. The formulas of the core QTL logic are defined by the following grammar, where a is a constant 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.)

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

At a given state the formula $p(\text{“a”})$ means that $p(\text{“a”})$ happened, more formally, that $p(\text{“a”})$ 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 $(\varphi_1\, \mathcal{S}\, \varphi_2)$ (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 common future time *until* modality [19]. The property $\ominus\, \varphi$ means that φ is true in the previous state. This is the past dual of the common future time *next* modality. We can also define the following additional temporal operators: $P\, \varphi = (true\, \mathcal{S}\, \varphi)$ (“previously”), and $H\varphi = \neg P \neg\varphi$ (“always in the past”). The operator $[\varphi_1, \varphi_2)$, borrowed from [17], has the same meaning as $(\neg\varphi_2\, \mathcal{S}\, \varphi_1)$, but reads more naturally as an interval.

In the following 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 IV the algorithm is introduced which encodes such sets of assignments as BDDs.

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 an 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 $(\gamma, \sigma, i) \models \varphi$.
- $(\gamma, \sigma, i) \models (\varphi\, \mathcal{S}\, \psi)$ 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[1] $(\gamma[x \mapsto a], \sigma, i) \models \varphi$.

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

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*, $(\varphi \mathcal{S} \psi) = (\psi \vee (\varphi \wedge \ominus(\varphi \mathcal{S} \psi)))$. This will serve in the implementation to work with only two versions of the sets of assignments, for the current and previous state:

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

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. We have that $hide(ext(\Gamma, U), U) = \Gamma$. 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 $W \cup W'$.

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

As before, the interpretation for the rest of the operators can be obtained from the above using the connections between the operators. For example, $I[P\,\varphi,\sigma,i] = I[(true\,\mathcal{S}\,\varphi),\sigma,i]$. 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.

IV. An efficient Algorithm using BDDs

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) [8]. A BDD is a compact representation for a Boolean tree representing a Boolean function. Because of compaction, however, the BDD forms a directed acyclic graph rather than a tree. Each internal node is marked with a Boolean variable. The left edge from a node represents that this variable has the Boolean value 0, while the right edge 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, where the result of the Boolean function does not depend on the value of the corresponding variable. The leaves have the Boolean values 0 and 1. Thus, following a path in this graph, moving left or right corresponding to choosing 0s or 1s, respectively, leads to a leaf node that is marked by either a 0 or 1, representing the Boolean value returned by the function for the Boolean values on the path. The graph is compacted in such a way that isomorphic subtrees are "glued" together. Instead of keeping a node b with left or right edges that lead to the same subgraph, the node and its outgoing edges are removed from graph representation of the BDD. (previous edges point directly to a 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 [9].

When a new value of some domain D_i appears in a predicate in the current state, we add it to a list of values of that domain that were seen. In order to search efficiently if this value already appeared, in time linear with its representation, we can use e.g. a hash table. 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"].

Each new value that appears in the monitored sequence is enumerated as a binary number. We use BDDs to represent sets of values. The BDDs are over Boolean representations of *enumerations* of the observed values, according to the order in which they appear in the input, rather than a direct representation of the actual domain values. Thus, using two bits, "ab" can be represented as the bit string 00 (we start to enumerate from 00), "de" as 01, "af" as 10 and "fg" as 11. 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 01 and 10. If the Boolean function is over b_0 (for most significant bit) and b_1 (for least significant), then this is the Boolean function $(\neg b_0 \wedge b_1) \vee (b_0 \wedge \neg b_1)$.

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 assume no more than 8 values. Then we can use the bits $y_0 y_1 y_2 x_0 x_1 x_2$, 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.

A subset of a set of k values can therefore be represented as function on $\lceil \log_2(k) \rceil$ bits. 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 assignments to the m variables with $\Sigma_{i=1..m} \lceil \log_2(k_i) \rceil$ 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.

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 occurrence order of appearance of a (among other values of the same domain) in the trace. Thus, if a is the first value occurring for that domain, **lookup**(a) will return $00\ldots00$. If it is the 2nd value occurring, $00\ldots01$, and so forth. We update this representation for each new state that appears.

We use a function called **build**(x,a) for building a BDD function that represents an assignment of a to the variable x, independent of the other variables. For example, if **lookup**$(a) = 011$, assuming we use only 3 bits, b_0, b_1 and b_2 to represent values, then **build**(x,a) will obtain a BDD representation of the function $\neg b_0 \wedge b_1 \wedge b_2$. There may be other bits, representing other variables, but the BDD function is independent of them (which leads to a large compaction).

Union and intersection of sets of assignments are translated simply to 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 use the following BDD function to perform existential quantification over bits: **exists**$(\langle x_0, \ldots, x_{k-1} \rangle, bdd)$. Finally, BDD$(0)$ and BDD(1) are the BDDs that return always 0 or 1, respectively.

The algorithm uses these 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 [14]. 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 [14] the vectors contain Boolean values, here the values are BDD functions. The algorithm follows.

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

We can define the number of bits per domain to a large enough number k such that we anticipate no more than 2^k different values. For example, if $k = 20$, this will allow more than a million different values. In fact, large part of the BDD that is related to bits that are not used is mostly compacted away. To see this, recall that the BDD functions, obtained during the runtime verification for representing the sets of assignments for the subformulas, are functions from the enumeration of values, according to the order in which they appear in the input. We start enumerating from $00\ldots00$, and then continue with $00\ldots01$, $00\ldots10$, etc. The Boolean operators, including the negation and applying quantification, maintain invariantly, that as long as only $m < 2^k$ values appeared, then the values for the binary representation of the $m+1st$ to the $2^k th$ enumerations of values are the same (for any combination of values of the other variables). In fact, it can be shown by induction on the length of temporal formulas and the input sequence that these enumerations, and in particular the enumeration $11\ldots11$, correctly represent all the values that were not seen so far in the input[2].

Suppose now that only $l < k$ bits are needed for storing the current set of enumerations, where the other (most significant bits) are 0. Then the maximal enumeration that is assigned to input values is no larger than binary $00\ldots0011\ldots11$, with $k-l$ times 0s, and l times 1s. The BDD function will return the same Boolean values for larger enumerations of the same domain (these will be enumerations that have at least a single 1 in the most significant $k-l$ digits). This by itself leads to a significant compaction.

It is important to maintain that for infinite domains there is at least one such unused enumeration for the allocated k bits. In particular, the largest possible enumeration $11\ldots11$ would play this role (but as discussed above, possibly also some smaller enumerations). To see why this is important, 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 $P\,g(x)$ will be represented as $\mathrm{BDD}(1)$, returning constantly a 1. Thus, $\neg P\,g(x)$ will be calculated to $\mathrm{BDD}(0)$, returning constantly 0. Now, $\exists x \neg P\,g(x)$ will be translated into $\exists x_0 \exists x_1 \ldots \exists x_{k-1} \mathrm{BDD}(0)$, and will return $\mathrm{BDD}(0)$ (*false*). However, checking $\exists x \neg P\,g(x)$ should have returned $\mathrm{BDD}(1)$ (*true*), since it claims 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.

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 3 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. We can encode a fixed BDD function $\mathit{smaller}(y,t)$ that expresses that the bits that

[2]Formally, let ψ be a subformula, for which a BDD B_ψ was constructed so far. Then B_ψ will return 1 for exactly the following bit strings. Let γ be some assignment satisfying ψ after the current input. Construct the following concatenation of bit strings, according to the given order on variables: for each variable, if its value under γ has appeared in the input, concatenate its binary enumeration, otherwise, concatenate some enumeration larger than the number of its domain values seen so far.

represent y have a binary value that is *smaller* than the binary representation of t. (Note that we start enumerating from $00\ldots00$, hence we check *smaller* rather than *smaller or equal*). For example, if we use 2 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; effectively, it is translated into $\forall x\, (smaller(x,m) \rightarrow \varphi)$.

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 domain of y. We saw earlier that $\exists x\, \neg P\, g(x)$ should always return a *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 in the domain of x that appeared in the execution so far within a predicate name other than g, but not within g.

Comparing variables. Another important extension is to be able to compare different variables, i.e., $x = y$ (or $x \neq y$). This can also be coded as a fixed BDD over the bit representation of enumerations of values of x and y. This is encoded as a BDD representing $x_0 = y_0 \wedge x_1 = y_1 \wedge \ldots x_{k-1} = y_{k-1}$.

V. IMPLEMENTATION

We implemented a monitoring tool for the QTL logic, called DEJAVU. Let $\mathbb{E}$ be the type of n-ary predicate symbols, and the $\mathbb{B}$ be the type of Boolean values. The implementation of the monitoring algorithm presented in Section IV 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 (a sequence of events), and returns a verdict, effectively a Boolean value for each 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 [15] 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 (observing a program as it executes) as well as offline (analyzing log files) monitoring. We shall illustrate the monitor generation using an example. Consider the following variation of the first property from Section I (using syntax supported by the implementation):

```
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
  }
}
```

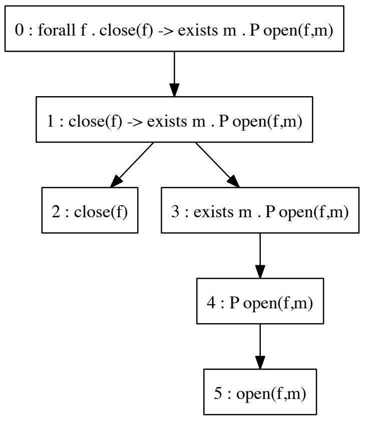
0 : forall f . close(f) -> exists m . P open(f,m)
1 : close(f) -> exists m . P open(f,m)
2 : close(f)
3 : exists m . P open(f,m)
4 : P open(f,m)
5 : open(f,m)

Fig. 1: Monitor (top) and AST (bottom) for the property

```
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 1 (bottom) 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 1 (top). Specifically two arrays are declared, indexed by subformula indexes: pre for the previous state and now for the current state. A BDD here 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 the zero-BDD. 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). Quantification is solved by performing quantification over the relevant bits of the BDD corresponding to the variable in question.

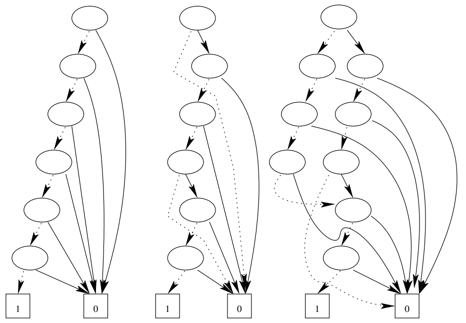

Fig. 2: BDDs for (left) subformula 5 on the first event, (mid) subformula 5 on the second event, (right) subformula 4 on the second event

Example

Assume that each variable f and m is represented by three bits. Consider the input trace, consisting of three events: $\langle open(input, read), open(output, write), close(out)\rangle$. When the monitor processes the first event for subformula number 5 (all subformulas here are numbered according ot Figure 1), it will create a bit string composed of a bit string for each parameter f and m. As previously explained, bit strings for each variable are allocated in increasing order: 000, 001, 010,..., hence the first bit string representing the assignment [f↦*input*,m↦*read*] becomes 000000 where the three rightmost bits represent *input* and the three leftmost bits represent *read*. Figure 2 (left) shows the corresponding BDD. (Note that most significant bits are implemented lower in the BDD.) 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.

Upon the second event, new values $(output, write)$ are observed as argument to the *open* event. Hence a new bit string for each variable f and m is allocated, in both cases 001 (the next unused bit string). The new combined bit string for the assignments satisfying subformula 5 then becomes 001001, again forming a BDD representing a single assignment, appearing in Figure 2 (mid). Subformula 4 now becomes the union of the two BDDs, resulting in the BDD on Figure 2 (right). The existential quantification in subformula 3 causes the BDD to be reduced to only the first 3 bits. However since subformula 2 is still *false* the whole formula evaluates to *true*.

```
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))
```

Fig. 3: Evaluation properties in QTL

Finally, on observation of the third and last event *close*(*out*), a new value *out* for f is observed, and allocated the bit pattern 010, represented by the corresponding BDD for subformula 2. As end result, for the formula in node 1 we end up with a BDD that is neither constantly *true* nor constantly *false*, and hence universally quantifying over it yields *false* since it is not the case that for all bit assignments it yields *true*.

VI. EVALUATION

DEJAVU performance is evaluated by comparing against MONPOLY, the tool that seems to have most similarities to DEJAVU as previously discussed. We specifically evaluated the three temporal properties shown in Figure 3, on different sizes of traces, while varying the number of bits allocated to represent variables in BDDs. The properties were encoded in MONPOLY in a 1-1 manner. 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). Finally, 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 last subformula states the FIFO principle of queues.

TABLE I: Evaluation of DEJAVU and MONPOLY

Property	Trace length	MONPOLY (sec)	DEJAVU (sec) bits per var.: 20 (40, 60)
ACCESS	11,006	**1.9**	**3.1** (3.3, 3.2)
	110,006	**241.9**	**6.1** (9.1, 10.9)
	1,100,006	**58,455.8**	**36.8** (61.9, 88.8)
FILE	11,004	**61.1**	**2.8** (2.8, 3.0)
	110,004	**7,348.7**	**6.3** (6.5, 8.6)
	1,100,004	**DNF**	**30.3** (43.9, 59.5)
FIFO	5,051	**158.3**	**195.4** (OOM, -)
	10,101	**1140.0**	**ERR** (-, -)

Table I shows the result of the evaluation, which was performed in the Mac OS X 10.7.5 operating system on a 2 $\times$ 2.93 GHz 6-Core Intel Xeon with 32 GB of memory. 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) 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) occur. In addition, for each trace we experimented with three different sizes of bit vectors: 20, 40 and 60 bits, 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: DNF = Did Not Finish (during 16 hours), OOM = Out of Memory, and ERR = an error occurred, in this case an array index out of bound problem in the JavaBDD package. The important numbers to compare are in bold font.

Table I demonstrates clearly that w.r.t. the first two properties, DEJAVU outperforms MONPOLY by a factor up to 3000. Those are substantial differences, and demonstrates that BDDs may be an interesting way to refer to stored data. However, for the last FIFO property, the two systems are somewhat comparable, although MONPOLY seems to do better on this particular property. The complexity lies in the last of the four subformulas in the conjunction, the actual FIFO property. Increasing the number of bits allocated per variable, from 20 to 40 and 60, does not seem to have a substantial impact on the performance, except for the FIFO property, where it causes an OOM result for 40 bits.

VII. 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 encoding 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. Redundant bits, used pessimistically for representing encodings in the expectation that the number of values encountered during the execution will grow considerably, do not cause a large explosion in the size of the BDD. 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 an iterative programming language such as C will result an even quicker runtime verification monitor. A limitation of our approach is that the encoding of assignment sets does not blend well with using various relations between values. While we can easily compare variables to be equal or not equal, we are not capable of comparing, say, whether one value is smaller than another value in an efficient way. This remains a challenge for future work.

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] R. E. Bryant, Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, ACM Comput. Surv. 24(3), 293-318 (1992).

[9] 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.

[10] N. Decker, M. Leucker, D. Thoma, Monitoring modulo theories, Journal of Software Tools for Technology Transfer, Volume 18, Number 2, 2016.

[11] 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.

[12] S. Hallé, R. Villemaire, Runtime Enforcement of Web Service Message Contracts with Data, IEEE Transactions on Services Computing, Volume 5 Number 2, 2012.

[13] K. Havelund, Rule-based runtime verification revisited, Journal of Software Tools for Technology Transfer, Volume 17 Number 2, Springer, 2015.

[14] K. Havelund, G. Rosu, Synthesizing Monitors for Safety Properties, TACAS 2002, 342-356.

[15] JavaBDD, http://javabdd.sourceforge.net.

[16] 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.

[17] 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.

[18] O. Kupferman, M. Y. Vardi, Model Checking of Safety Properties, Formal Methods in System Design 19(3): 291-314, 2001.

[19] Z. Manna, A. Pnueli, Completing the Temporal Picture, Theoretical Computer Science 83, 91-130, 1991.

[20] 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.

[21] 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.

[22] J. Whaley, D. Avots, M. Carbin, M. S. Lam, Using Datalog with Binary Decision Diagrams for Program Analysis, APLAS 2005, 97-118.