BDDs for Representing Data in Runtime Verification*

Klaus Havelund[1] and Doron Peled[2]

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

Abstract. A BDD (Boolean Decision Diagram) is a data structure for the compact representation of a Boolean function. It is equipped with efficient algorithms for minimization and for applying Boolean operators. The use of BDDs for representing Boolean functions, combined with symbolic algorithms, facilitated a leap in the capability of model checking for the verification of systems with a huge number of states. Recently BDDs were considered as an efficient representation of data for Runtime Verification (RV). We review here the basic theory of BDDs and summarize their use in model checking and specifically in runtime verification.

1 Introduction

Boolean functions have shown to have many useful applications in computer science. E.g. Boolean functions early on resulted in lowering the production costs and footprints of digital circuits. A later use of Boolean functions is analysis of software and hardware [9], where BDDs represent sets of states as Boolean functions. For these applications, there is a need not only to achieve a compact representation, but also to have efficient procedures for applying Boolean operators. In particular, the conjunction of Boolean functions that represent sets of states returns the intersection of these sets, and the disjunction returns to their union.

More specifically, a *Boolean Decision Diagram* or BDD, is a rooted directed acyclic graph (DAG), with nonleaf nodes labeled by Boolean variables, and leafs labeled with 0 (*false*) or 1 (*true*). BDDs were already used for representing Boolean functions since the middle of the previous century [21]. However, it was only in the 80s that Bryant [6] presented their reduced ordered version (ROBDD), where the ordering between the Boolean variables are fixed along each path from the root to a leaf, and isomorphic parts are combined.

The ability to encode sets of states and relations between values and to apply Boolean operators on ROBDDs was exploited in *model checking* (see, [10]). It resulted

* The research performed by the first author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research performed by the second author was partially funded by Israeli Science Foundation grant 1464/18: "Efficient Runtime Verification for Systems with Lots of Data and its Applications".

in a huge increase in the size of systems that can be checked over previous techniques. Recently ROBDDs have been used to support *runtime verification* of execution traces containing large amounts of data, e.g., in monitoring of sequences with data-carrying events against first-order past-time LTL formulas [15]. In this paper we survey these applications of BDDs with emphasis on runtime verification.

The paper is organized as follows. Section 2 provides a brief introduction to BDDs. Section 3 outlines how to represent sets and relations with BDDs. Section 4 gives a brief introduction to their use in symbolic model checking. Section 5 presents the use of BDDs in runtime verification of first-order past-time LTL for representing the data quantified over in traces. Section 6 expands on this framework and illustrates how BDDs can be used to monitor timed events against first-order past-time LTL with time constraints.

2 Introduction to OBDDs

A Boolean function $f : \{0,1\}^k \mapsto \{0,1\}$ maps k-tuples of Boolean values 1 and 0 (for *true* and *false*, respectively) to Boolean values. Each k-tuple can be considered as an assignment $\Gamma : \mathcal{V} \mapsto \{0,1\}$ from variables in a fixed set $\mathcal{V}$ to a Boolean value. A Boolean function can be expressed using *literals*, which denote the Boolean variables, and Boolean operators: conjunction (*and*), disjunction (*or*) and negation (*not*). Conjunction is denoted here, in a standard way, using concatenation, disjunction is denoted with $+$ and negation is denoted by putting a line over the negated part; conjunction has priority over disjunction. A *minterm* is a conjunction of literals, e.g., $x_1\overline{x_3}x_4$ (standing for $x_1 \wedge \neg x_3 \wedge x_4$). Each Boolean function can be written in *disjunctive normal form* as a sum (disjunction) of minterms.

An OBDD $G = ((Q, v_0, E), \mathcal{V}, <, L)$ consists of the following components:

- (Q, v_0, E) is a rooted directed acyclic graph with
  - Q is finite set of *nodes*. Each non-leaf node has two distinguished *successor* nodes $l(v)$ and $h(v)$.
  - $v_0 \in Q$ is the *root* node.
  - $E \subseteq Q \times Q$ is a finite set of directed edges. Each non-leaf node has exactly two outgoing edges to its successors: the *low* edge $(v, l(v)) \in E$ and the *high* edge $(v, h(v)) \in E$.
- $\mathcal{V}$ is a finite set of *Boolean variables* (or *BDD variables* or simply *bits*).
- $<$ is a total order on $\mathcal{V}$, extended with two *maximal* values: 0 and 1.
- $L : Q \mapsto \mathcal{V} \cup \{0, 1\}$ is a mapping that satisfies the following conditions:
  - The leafs are mapped to $\{0,1\}$ and the non-leaf nodes are mapped to $\mathcal{V}$.
  - If $(v, v') \in E$, then $L(v) < L(v')$, i.e., variables that label nodes on any path of the graph appear according to the order $<$, hence the name *Ordered BDD*.

An OBDD G represents a Boolean function (expression) over the variables $\mathcal{V}$. The interpretation of a BDD as a formula is based on the *Shannon expansion*

$$f = \overline{x} f[0/x] + x f[1/x] \tag{1}$$

where $f[0/x]$ ($f[1/x]$, respectively) denotes the function f when fixing x as 0 (1, respectively); thus, equation (1) separates the function f into two components, according to the truth value of the variable x. Each node $v \in Q$ in the OBDD represents the formula

$$f_v = \begin{cases} L(v), & \text{for a leaf v} \\ \overline{L(v)} f_{l(v)} + L(v) f_{h(v)}, & \text{for a non-leaf v.} \end{cases} \tag{2}$$

An OBDD G represents the formula f_{v_0} of the root node v_0. Another way to interpret an OBDD is that a path $F \subset E$ that starts at the root and ends at a leaf w corresponds to an assignment Γ, where for each nonleaf node $v \in Q$,

$$\Gamma(L(v)) = \begin{cases} 0, & \text{if } (v, l(v)) \in F \\ 1, & \text{if } (v, r(v)) \in F \end{cases} \tag{3}$$

and for each variable $x \in \mathcal{V}$ that does not label any node on the path $\Gamma(x)$ can be either 0 or 1. The Boolean function returns for the assignment Γ the truth value $L(w)$. We will, from now on, use the convention of calling OBDDs simply BDDs.

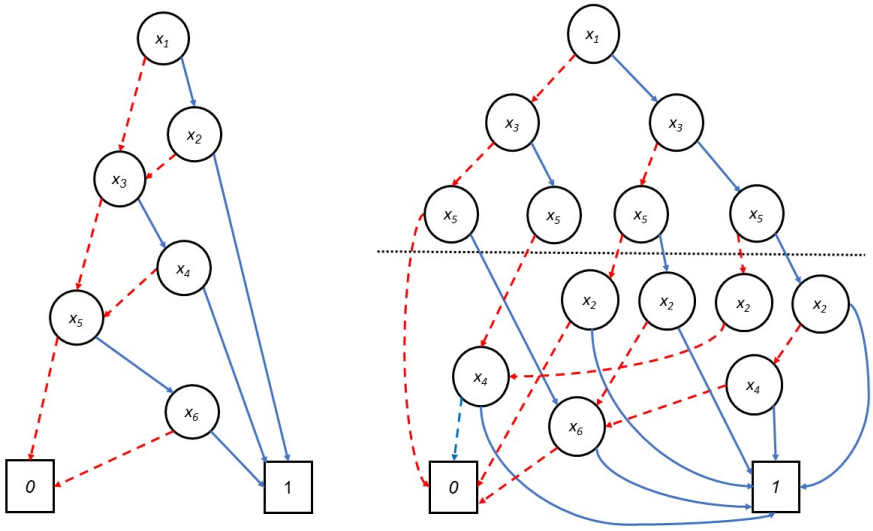


Fig. 1: The effect of variables order.

BDDs are typically depicted as in Figure 1 (both left and right), where each nonleaf node is denoted with a circle, and the leafs are denoted with rectangles. Edges of the form $(v, l(v'))$ are dashed, while edges of the form $(v, h(v))$ are full lines.

A benefit of using BDDs is the ability to *minimize* them, often producing a representation that is considerably smaller than other representations. The minimization

allows combining isomorphic subgraphs, using the following rules, applied from the leafs upwards.

1. Combine all the leaf nodes that are labeled 1 and all the leaf nodes that are labeled 0. Redirect incoming edges to the resulting leafs.
2. When $l(f) = l(g)$ and $h(f) = h(g)$, combine the nodes f and g and redirect incoming edges to the single copy.
3. For a node f such that $l(f) = h(f)$, remove f and redirect its incoming edges to $l(f)$.

The term *minimal* BDD is sometimes used to emphasize that it is a *Reduced* Ordered Boolean Decision Diagram. The algorithm is linear in the size of the BDD[1].

As an example, consider the Boolean function f over the variables x_1, x_2 and x_3 that returns the parity (the sum modulo 2) of $x_1 + x_2 + x_3$. The full binary tree for this function, where $x_1 < x_2 < x_3$, appears in Figure 2**A**. We obtain 2**B** by combining leafs with the same Boolean values, then 2**C** by combining the middle two x_3 nodes, and finally 2**D** by combining the exterior two x_3 nodes. No further minimization steps are available.

The order of variables in the BDD can greatly impact the size of the BDD. A classical example is the expression $x_1x_2 + x_3x_4 + \ldots x_{n-1}x_n$. Given the variable order $x_1 < x_2 < \ldots < x_{n-1} < x_n$, the BDD grows linearly with n. But for the order $x_1 < x_3 < \ldots x_{n-1} < x_2 < x_4 < \ldots < x_{n-2} < x_n$, the BDD grows exponentially with n. The two BDDs for $n = 6$ appear in Figure 1. Note that in the right BDD, the part above the dotted line, with nodes labeled with $x_1, x_3, \ldots, x_{n-1}$, is a full binary tree. For each Boolean function, there is exactly one (minimized) BDD per each variable ordering [6]. However, in some pathological cases, for example, when describing bit-vector multiplication circuits, the size of the minimal BDD grows exponentially with the number Boolean variables for any variables ordering.

Another benefit of using BDDs, in addition to achieving compact representation of Boolean functions, is the availability of efficient algorithms for applying Boolean operators. This makes BDDs useful for applications that process sets of data elements, such as model checking and runtime verification, as will be shown in Secions 4 and 5.

The **restrict** operator computes from a BDD representing a function f a BDD representing the function $f[0/x_i]$ ($f[1/x_i]$, respectively). It replaces any edge that leads to a node labeled with x_i with an edge (from the same source node) into x_i's low (high, respectively), as follows:

1. If, for the root node v, $L(v) = x_i$, then $f[0/x_i]$ ($f[1/x_i]$, respectively) is the BDD rooted at $l(v)$ ($h(v)$, respectively).
2. Replace any edge $(v,w) \in E$, where $L(w) = x_i$, by an edge $(v, l(w))$ ($(v, h(w))$, respectively), and remove w.
3. Minimize the BDD.

In Figure 3, the left BDD represents some function f, and the right BDD is $f[1/x_2]$.

[1] To achieve linearity, for rule 2, bucket sort is applied to cluster together the nodes with the same variable and the same outgoing l edge. Then within each bucket, bucket sort is applied again according to the outgoing h edge.

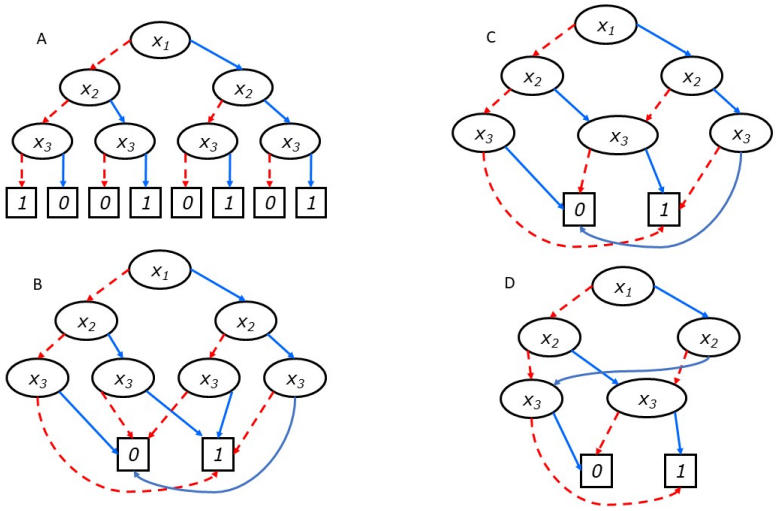


Fig. 2: A: original, B: reduce leafs, C: combine middle x_3's, D: combine other x_3's.

The operator **apply#** applies an arbitrary Boolean operator # (e.g., *and*, *or*) on BDDs. It is based on the fact that restriction distributes over function decomposition, i.e.,

$$(f\#g)[a/x] = f[a/x]\#g[a/x]$$

for $a \in \{0, 1\}$ and $x \in \mathcal{V}$. Then, using Shannon's expansion, see Equation (1), we have:

$$(f\#g) = \bar{x}(f\#g)[0/x] + x(f\#g)[1/x] \;=\; \bar{x}(f[0/x]\#g[0/x]) + x(f[1/x]\#g[1/x])$$

We calculate the BDD for $f\#g$, for BDDs f and g, using the following recursive procedure $apply\#(v_f, v_g)$, called initially with the roots of the two BDDs.

1. If v_f and v_g are leafs, then return a leaf v with $L(v) = L(v_f)\#L(v_g)$. Otherwise,
2. if $L(v_f) = L(v_g)$ (both parameters are labeled with the same variable), then return a node v with $l(v) = apply\#(l(v_f), l(v_g))$ and $h(v) = apply\#(h(v_f), h(v_g))$,
3. if $L(v_f) < L(v_g)$ (there is no node labeled with $L(v_f)$ in the current path of recursive calls in the BDD g) then return a node v with $l(v) = apply\#(l(v_f), v_g)$ and $h(v) = apply\#(r(v_f), v_g)$.
   The symmetric case is handled similarly.

A naive application of this procedure can repeatedly recalculate subgraphs starting from the same pair of BDD nodes. This is avoided by using a dynamic programming principle, where the results of the recursive calls are hashed according to the call parameters (v_f, v_g). This is demonstrated in Figure 4, where some (arbitrary) Boolean operator # is applied to the two BDDs that appear at the left. The tree in the middle of Figure 4 is obtained by using the recursive procedure without using dynamic programming. The

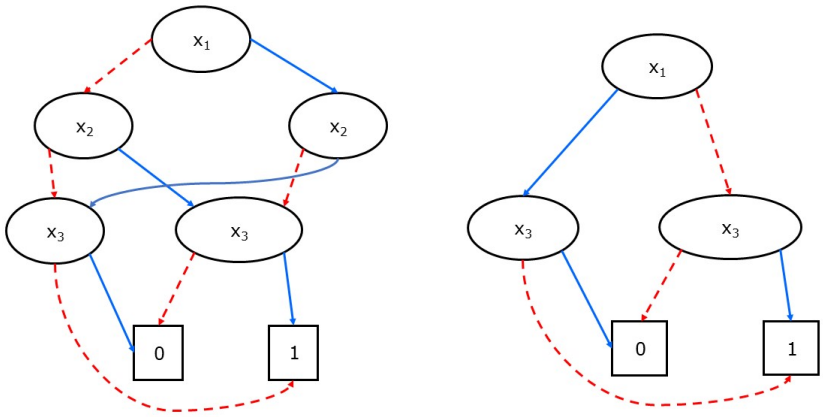


Fig. 3: A BDD f on the left, and $f[1/x_2]$ on the right.

DAG on the right is obtained using dynamic programming. Further reduction may be possible. Note that the leafs in the middle appear as pairs of nodes, whereas the corresponding leafs on the right appear as the Boolean # combinations between the leafs, e.g., R_5, S_4 in the middle part corresponds to $L(R_5)\#L(S_4)$ on the right. The size of the resulting BDD and the time complexity of the **apply** operator is limited to product of sizes of these BDDs.

The negation operator on BDDs is trivial; it requires reversing the labeling on the leafs, from 0 to 1 and from 1 to 0. Another useful operator is existential quantification over a Boolean variable, i.e., calculating $\exists x f$ for a BDD representing f. Since $\exists x f = f[0/x] \vee f[1/x]$, this operator can be implemented using **restrict** twice and then **apply** with $\# = \vee$ on the result.

Alternative representations *ZDDs*, for *Zero-suppressed Decision Diagrams*, were suggested by Minato [23]. ZDDs typically demonstrate better reduction than BDDs for Boolean functions in which the assignments that are satisfied are sparse. The reduction of ZDDs is slightly different than for BDDs. Reduction rules 1 and 2 remain the same. Reduction rule 3, which removes a node whose low and high edges point at the same node and redirects any incoming edge to its successor, is replaced with the following rule: a node v where its *high* successor is the constant 0, i.e., $L(h(v)) = 0$, is removed, and any incoming edge is redirected to $l(v)$.

Although ZDDs may produce a more compact representation than BDDs, the compaction that can be achieved is not exponential, but rather by a factor of the number of BDD variables $\mathcal{V}$.

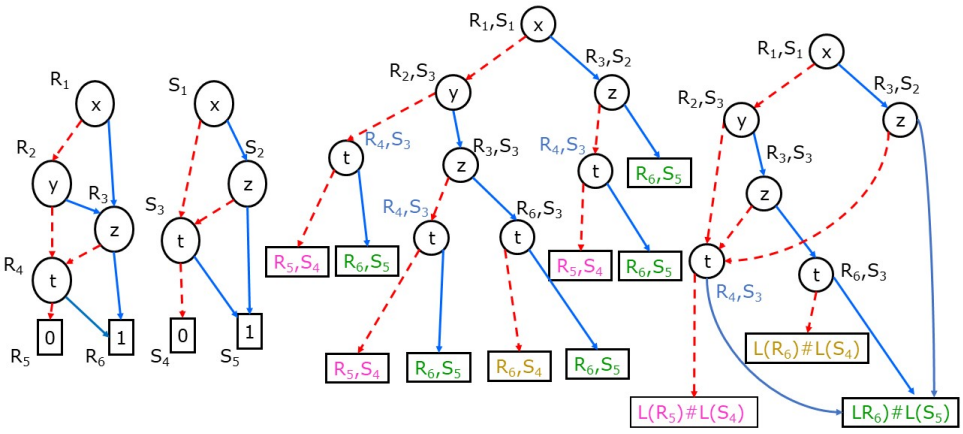


Fig. 4: *apply*# on BDDs (left) without (middle) and with (right) dynamic programming.

Multi Terminal Binary Decision Diagrams (MTBDDs) [3] extends the BDD notation to mappings from Boolean variables to a domain $\mathcal{D}$ that can be different than the Boolean values. Then, the **apply** operator can be used with, e.g., arithmetic operators like addition and multiplication instead of the Boolean operators. This is useful, for example, for the symbolic verification of probabilistic systems [1].

3 Representing Sets and Relations using BDDs

A Boolean function, and consequently a BDD, can represent a set of integer values. Each integer value i is, in turn, represented as a bit vector (i.e., as a binary number) $x_m \dots x_1$ where $i = x_1 \times 1 + x_2 \times 2 + \dots x_m \times 2^m$. For example, the integer 6 can be represented as the bit vector $x_3 x_2 x_1 = 110$. To represent a *set* of integers, the BDD returns *true* for any bit vector that represents an element in the set. For example, to represent the set $\{4, 6\}$, we first convert 4 into the bit vector $x_3 x_2 x_1 = 100$ and 6 into $x_3 x_2 x_1 = 110$. The Boolean function over x_1, x_2, x_3 is then $\overline{x_1} x_3$, which returns *true* exactly for these two bit vector combinations. To keep common conventions, we write a list of Boolean variables with the least indexed variable at the left but bit vectors and binary numbers with the least significant digit at the right.

This representation can be extended to represent relations, or, equivalently, a set of tuples over integers. The Boolean variables are partitioned into n bit vectors $x_1 = x^1_{k_1} \dots x^1_1$, $x_n = x^n_{k_n} \dots x^n_1$, each one of them representing an integer value. These bit vectors are then concatenated.

3.1 BDDs over integers

BDDs can represent a set of integers, where each value is kept as a bit vector, i.e., using its binary representation, using the BDD variables. This can be used, e.g., to represent integer values that an ALU processes or values of discrete timers. An advantage of this representation is that one can perform *arithmetic* operations and comparisons over sets of values, e.g., add a constant value Δ to each value in a set, or restrict a set to values that are bigger than a constant Δ, using BDD operations. We demonstrate how such operations are translated to BDDs.

The Boolean formula $addconst(t,t',\Delta)$ is satisfied by a triple of integer values t, t' and Δ, represented as the bit vectors $t_m \ldots t_1$, $t'_m \ldots t'_1$ and $\Delta_m \ldots \Delta_1$, respectively, such that $t' = t + \Delta$. The formula uses additional bits $r_1, \ldots, r_m$, where r_i is the *carry-over* from the i^{th} bits. Existential quantification is applied to remove the BDD variables $r_1, \ldots, r_m$. We ignore here the issue of addition overflow.

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

This Boolean function can be translated into a BDD over the variables $t_1, \ldots, t_m$, $t'_1, \ldots, t'_m$ and $\Delta_1, \ldots, \Delta_m$. It represents a relation on triples (t,t',Δ). When Δ is restricted to a fixed bit vector, the formula represents a relation on pairs (t,t').

Suppose that we want to update the set of values represented by a BDD B by adding a constant 3 to each value. We can do that by using *addconst* with the bit vector Δ set to the binary value $00 \ldots 011$. The BDD obtained by

$$\exists t_1 \ldots \exists t_m (B \wedge addconst(t,t',\Delta)) \tag{4}$$

is over the variables $t'_1, \ldots, t'_m$. It represents the values in B incremented by 3. Now we need to rename the variables $t'_1 \ldots t'_m$ back to $t_1, \ldots, t_m$. Renaming variables is a standard BDD operation, and we will denote this as $rename(C,t',t)$, where C is a BDD and t' and t are bit vectors. We obtain

$$rename(\exists t_1 \ldots \exists t_m (B \wedge addconst(t,t',\Delta)), t', t) \tag{5}$$

As another example, the formula $gtconst(t,\Delta)$ is satisfied by integers that are bigger than Δ (limited to the value $2^m - 1$, where the number of Boolean variables is m). Both t and Δ are integers represented as bit vectors, as before. Again, this is encoded as binary comparison, with Boolean variables $r_0, \ldots, r_m$ used to propagate the result of the comparison. As before, these variables are later removed using existential quantifiers

$gtconst(t,\Delta) = r_m$
where $r_0 = false$ **and**
for $1 \leq i \leq m$: $r_i = ((t_i \wedge \neg \Delta_i) \vee ((t_i \leftrightarrow \Delta_i) \wedge r_{i-1}))$

The functions *addconst* and *gtconst* can be adapted for signed integers as well.

3.2 BDDs over enumerations of values

A disadvantage of the representation suggested in Section 3.1 is that the number of BDD variables required can be very large. Representing integers requires $\lceil \log p \rceil$ bits, where p is the largest possible value. The problem can intensify when the represented data is over strings with varying lengths.

To alleviate this problem, sets of values and relations can be represented as BDDs over *enumerations* of values. When a value associated with a variable in the specification appears for the first time in the computation (e.g., during runtime verification, see Section 5), a new *enumeration* is associated with it. Enumeration values can be assigned consecutively according to their binary value; however, a refined algorithm can reuse enumerations that were used for values that can no longer affect subsequent results, see [13]. A hash table is used to point from the value to its enumeration so that in subsequent appearances of this value the same enumeration will be used. The use of enumerations instead of the actual values allows a representation with a smaller number of bits. In addition, enumerations of values that are not far apart often share large bit patterns, which can also contribute to the BDD compactness.

BDDs can represent relations over mixed domains, where some of them are encoded using enumerations, and others as binary numbers.

4 Using BDDs for Model Checking

BDDs have gained a huge popularity in the automated verification of finite state systems referred to as *model checking*. Comprehensive analysis of systems requires reasoning about their states and execution sequences, and the main bottleneck is state space explosion. A BDD can represent a Boolean function that encodes a set of states. Then, it is possible to apply operators on BDDs to process sets of states, rather than handling the states one at a time.

Consider a finite-state system with state space St and initial states $I \subseteq St$. The property we we want to check is that its execution must *never* arrive at states from $F \subseteq St$ (the *failure* states). Let $prec(s) \subseteq St$ be the set of states from which the system can move to $s \in St$ by performing one atomic transition, i.e., the *predecessor states* of s, and generalize it to $P(S) = \bigcup_{s \in S} prec(s)$. Checking that failure states cannot be reached from initial states is equivalent to checking that $I \cap P^*(F) \neq \emptyset$, where P^* denotes applying P repeatedly, 0 or more times. This can be described using the following pseudo-code:

$X1 := \emptyset; X2 := F;$
while $X1 \neq X2$ **do**
    $X1 := X2; X2 := X1 \cup P(X1);$
If $X1 \cap I \neq \emptyset$ **then** Return('failed');

Calculating $P(X1)$ and $X1 \cap I \neq \emptyset$ state by state is typically very expensive. This can severely limit the number of states that can processed. In *symbolic model checking* [9], all these operations are performed on BDDs, representing Boolean functions that encode sets of states. States are assignments a some fixed set of system (or program)

variables, and a set of states corresponds to a relation over the domains of these variables. Section Section 3.1 demonstrated how arithmetic operators can be applied to sets of integer values. This would work for the simple case where states consist of the value of a single integer variables but can be extended to operate on relations over mixed domains.

Let $I, F, X1$ and $X2$ be represented as the BDDs fI, fF, $fX1$, and $fX2$, respectively. A less trivial step is encoding P: instead of a function P, one can use a relation between the current states, represented as bit vectors, using the Boolean variables $x = x_m \dots x_1$, and the previous states, represented as bit vector using $x' = x'_m \dots x'_1$.The BDD R represents this relation over the BDD variables of x and x'. $R \wedge fX1$ restricts this relation so that the current state values satisfy $fX1$. Then $\exists x_1 \dots x_n (fX1 \wedge R)$ keeps only the state values of the predecessors to states satisfying $fX1$. The BDD operation *rename* is used to rename the variables of x' back to x. Finally, we apply disjunction to the obtained BDD and $fX1$ to obtain the union of the sets.

$fX1 := false$; $fX2 := fF$;
while $fX1 \neq fX2$ do
    $fX1 := fX2$; $fX2 := fX1 \vee rename(\exists x_1 \dots x_n (fX1 \wedge R), x', x)$;
If $(fX1 \wedge fI) \neq false$ then Return('failed');

5 Using BDDs for Runtime Verification

Runtime verification provides techniques for monitoring system executions against a formal specification. The monitored system is instrumented to report to the monitor on the occurrence of relevant events. The monitor observes the input events and keeps an internal *summary* of the prefix of the execution observed so far, which allows computing whether an evidence for a violation of the specification is already available.

Propositional Linear temporal logic (LTL) asserts about the evolution of an execution in time, using the future-time modalities $\Box$ (always), $\Diamond$ (sometimes), $\bigcirc$ (next-time) and $\mathcal{U}$ (until) [22]. It is possible to add to these modalities their corresponding past-time versions $\mathbf{H}$ (history), $\mathbf{P}$ (past), $\ominus$ (previous-time) and $\mathcal{S}$ (since), although adding them does not increase the expressive power [12].

RV often focuses on properties expressed in past-time Linear Temporal Logic (LTL), which includes the modalities $\mathbf{H}$, $\mathbf{P}$, $\ominus$ and $\mathcal{S}$, where it is implicitly assumed that the specification needs to hold for *all* the prefixes of the execution. This assumption is equivalent to prefixing each property with the $\Box$ operator. These properties correspond to temporal *safety properties* [2], where a failure can always be detected on a finite prefix as soon as it occurs [20].

First-order past-time LTL is obtained by adding predicates and quantification over data. An example of a first-order temporal specification is the following.

$$\forall f \, (close(f) \rightarrow \mathbf{P}\, open(f)) \tag{6}$$

It asserts that every file that is closed was opened before. Here, we need to keep in the summary a *set* of all the opened files so that we can compare them to the closing of files. In general, the summary in this case extends the one used for the propositional case by

keeping for each subformula the set of assignments, essentially a relation between the free variables occurring in a formula and the values that make the formula true.

Traces Assume a finite set of domains $D_1, \ldots, D_k$. Assume further that the domains are infinite, e.g., they can be the integers or strings[2]. Let P a set of *names* of unary predicates with typical instances p, q, r. Each predicate name p is associated with some domain $D_i = \mathit{domain}(p)$. A *ground predicate* is constructed from a predicate name and a constant of the same type. Thus, if the predicate name is p one can form ground predicates such as $p(\text{“gaga”})$ and $q(42)$. The restriction to unary predicates is not due to any principle limitation, but simplifies the presentation. An *event* is a finite set of ground predicates. For example, if $P = \{p, q, r\}$, then the set $\{p(\text{“gaga”}), q(42)\}$ is an event. A *trace* $\sigma = e_1 e_2 \ldots e_n$ is a finite sequence of events enumerated from 1. We denote the i^{th} event e_i in σ by $\sigma[i]$.

5.1 Syntax

Let V be a finite set of *variables*, with typical instances x, y, z. A predicate is constructed from a predicate name, and a variable or a constant (in which case it is a ground predicate) of the same type. Thus, if the predicate name p and the variable x are associated with the domain of strings, we have predicates like $p(\text{“gaga”})$ and $p(x)$. The syntax is as follows:

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

The formula $p(a)$, where a is a constant in $\mathit{domain}(p)$, means that the ground predicate $p(a)$ occurs in the most recent event. The formula $p(x)$, for a variable $x \in V$, holds with a binding of x to the value a if a ground predicate $p(a)$ appears in the most recent event. The formula $\exists x \, \varphi$ has the obvious meaning that there exists some x such that φ (in which x can appear free) holds. In addition, We can derive the universal quantification as $\forall x \, \varphi = \neg\exists x \neg\varphi$ and and other forms: $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\mathbf{P} \, \varphi = (\mathit{true} \, \mathcal{S} \, \varphi)$, and $\mathbf{H} \, \varphi = \neg\mathbf{P} \, \neg\varphi$.

5.2 Semantics

Assignments of values to variables are at the core of this semantics. An *assignment* over a set of variables $W \subseteq V$ maps each variable $x \in W$ to a value from its associated domain $\mathit{domain}(x)$. For example $[x \rightarrow 5, y \rightarrow \text{“abc”}]$ maps x to 5 and y to “abc”. By $\gamma[x \mapsto a]$ we mean the overriding of the assignment γ with the binding $[x \mapsto a]$. We denote by ε the empty assignment. Let $\mathit{free}(\varphi)$ be the set of free (i.e., unquantified) variables of a formula φ. Furthermore, let $\gamma|_{\mathit{free}(\varphi)}$ denote the restriction (projection) of an assignment γ to the free variables appearing in φ.

Predicate semantics We define a classic semantics for first-order past-time LTL. The assertion $(\gamma, \sigma, i) \models \varphi$ means that the trace $\sigma = e_1 e_2 \ldots e_n$ satisfies the formula φ for an assignment γ over $\mathit{free}(\varphi)$, where $1 \leq i \leq n$ (the relevant part of the execution is only the prefix $e_1 e_2 \ldots e_i$).

[2] For dealing with finite domains see [15].

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

For a finite trace σ, we write $\sigma \models \varphi$ to mean $\forall i (1 \leq i \leq \mathit{length}(\sigma) \rightarrow (\varepsilon, \sigma, i) \models \varphi)$.

Set semantics It helps the presentation of the BDD-based algorithm to first refine the semantics of the logic as a function that calculates the set of assignments satisfying a formula. Let $I[\varphi, \sigma, i]$ be the *interpretation function*, defined below, that returns a set of assignments such that $(\gamma, \sigma, i) \models \varphi$ iff $\gamma|_{\mathit{free}(\varphi)} \in I[\varphi, \sigma, i]$. The empty set of assignments $\emptyset$ behaves as the Boolean constant *false* and the singleton set $\{\varepsilon\}$ that contains the empty assignment behaves as the Boolean constant *true*. We define the union $\bigcup$ and intersection $\bigcap$ operators on sets of assignments, even if they are defined over non identical sets of variables. In this case, the assignments are extended to the union of the variables. Thus intersection between two sets of assignments A_1 and A_2 is defined like database "join" operator; i.e., it consists of the assignments whose projection on the *common* variables agrees with an assignment in A_1 and with an assignment in A_2. Union is defined as the dual operator of intersection.

Furthermore, let A be a set of assignments over the set of variables W; we denote by $\mathit{hide}(A, x)$ (for "hiding" the variable x) the set of assignments obtained from A after removing from each assignment the mapping from x to a value. In particular, if A is a set of assignments over only the variable x, then $\mathit{hide}(A, x)$ is $\{\varepsilon\}$ when A is nonempty, and $\emptyset$ otherwise. $A_{\mathit{free}(\varphi)}$ is the set of all possible assignments of values to the variables that appear free in φ. We add a 0 position for each sequence σ (an "initial state"), where I returns the empty set for each formula. The assignment-set semantics is shown in the following. For all occurrences of i, it is assumed that $i \geq 1$.

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

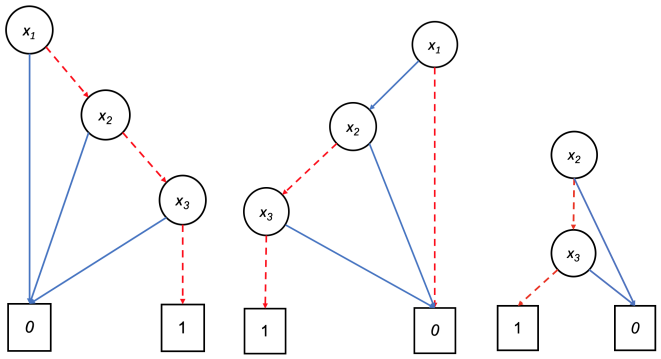
5.3 Algorithm

A runtime verification algorithm for first-order LTL was presented in [4], based on applying database operations to relations. We present here an RV algorithm that is based on BDDs [15].

BDDs for runtime verification We saw in Section 3 how a set of integers can be represented as a BDD: the BDD returning true for all bit-patterns corresponding to the binary encoding of the integers in the set. It was also explained how a value from an arbitrary value domain D, e.g. strings, can be represented as an integer while recording the mapping from the value to the integer in a hash map. E.g. the string "abc" can be represented as the number 6, which has the binary encoding 110. Consequently a set of values from the domain D can be represented as a BDD that is satisfied by the binary encodings of the corresponding integers.

First-order LTL formulas can contain multiple variables; a BDD can represent a set of assignments to variables as tuples of integers, each tuple position corresponding to a particular variable. This is the same as representing a relation over the domains of the variables. As shown in Section 3, such a tuples can be represented by concatenating the bit vectors of the individual tuple elements. For example, consider the assignment $[x \to 5, y \to \text{“abc”}]$. This can be thought of as the tuple $(5, \text{“abc”})$ if we associate the first tuple position with x and the second tuple position with y. If we map 5 and "abc" to integers, e.g. 1 and 2, the assignment can be thought of as being represented by the tuple $(1,2)$. This tuple can then finally be represented as the concatenation of the binary representations 001 and 010 of these integers: 001010. This insight is the core idea in the BDD representation, first presented in [15], and implemented in the tool DejaVu. Operations on BDDs, such as negation (corresponding to set complementation), conjunction (corresponding to set intersection), and disjunction (corresponding to set union) are very efficient. With k bits used for representing the enumerations for a variable, the BDD can represent 2^k values for each variable [8]. Furthermore, 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 during runtime.

Example Consider the formula $\exists x\, \mathbf{P} p(x)$ (there exists an x such that $p(x)$ occurred in the past). Consider furthermore the two-event trace $\langle \{p(\text{“ab”})\}, \{p(\text{“cd”})\} \rangle$. We will focus on the sub-formulas $p(x)$ and $\mathbf{P} p(x)$ and the BDDs that need to be calculated for them to keep a summary of the observed sequence of events during analysis of this trace. Figure 5 shows the generated BDDs. After the first event $p(\text{“ab”})$, when computing the BDD for $p(x)$, x is bound to "ab". We allocate an enumeration, an integer, in this case 0, and map "ab" to 0 in a hashmap. For the subformula $p(x)$ we create the BDD 5a that is satisfied exactly by its binary value 000. For The subformula $\mathbf{P} p(x)$, we need a BDD that is satisfied by all the binary encodings of enumerations for values seen so far. Since we only observed the value "ab" as an argument to p, the same BDD 5a is also used for $\mathbf{P} p(x)$. After the second event, the new value "cd" is mapped to the integer 1 (updating the hashmap), and the BDD 5b that is satisfied by its binary value 001 is created for the subformula $p(x)$. For the subformula $\mathbf{P} p(x)$ we build the BDD 5c that represents the set $\{\text{“ab”}, \text{“cd”}\}$, satisfied by the binary values 000 and 001. This BDD is obtained using the $\vee$-operation on the BDD 5b constructed at the current step for $p(\text{“cd”})$ and the BDD 5a, constructed in the previous step. Splitting the variable x into its bits: $x_3 x_2 x_1$, with x_1 the least significant bit, the figure shows the Boolean expressions over these bits corresponding to the BDDs.



(a) BDD for $p(x)$ and $\mathbf{P}p(x)$ after 1st event, "ab" maps to 0 (000), i.e.: $\overline{x_3}\overline{x_2}\overline{x_1}$

(b) BDD for $p(x)$ after 2nd event, "cd" maps to 1 (001), i.e.: $\overline{x_3}\overline{x_2}x_1$

(c) BDD for $\mathbf{P}p(x)$ after 2'nd event, 000 and 001, i.e.: $\overline{x_3}\overline{x_2}$

Fig. 5: BDDs for the sub-formulas $p(x)$ and $\mathbf{P}p(x)$ for the trace $\langle\{p(\text{“ab”})\},\{p(\text{“cd”})\}\rangle$, mapping "ab" to the integer 0 (binary 000), and mapping "cd" to the integer 1 (binary 001).

Some basic BDD operations We first introduce some basic functions used by the algorithm. Given some ground predicate $p(a)$ observed in the execution, matching with $p(x)$ in the monitored property, let $\mathbf{lookup}(x,a)$ be the enumeration of a in binary form. If this is the first occurrence of a, then it will be assigned a new enumeration. Otherwise, **lookup** returns the enumeration that a received before. We can use a counter[3], for each variable x, counting the number of different values appearing so far for x. When a new value appears, this counter is incremented, and the value is converted to the binary representation as discussed above. Enumerations that at any point in time have not yet been used represent the values not yet seen. In particular, we always leave one enumeration, $11\ldots11$ (all 1's), for this purpose. This enumeration is never allocated to represent observed data but represents all data not yet seen. This allows us to use a finite representation and quantify existentially and universally over *all* values in infinite domains, where $11\ldots11$ represents the infinite set of values not yet seen. 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.

The function $\mathbf{build}(x,A)$ returns a BDD that represents the set of assignments where x is mapped to (the enumeration of) v for $v \in A$. For example, assume that we use three Boolean variables (bits) x_1, x_2 and x_3 for representing enumerations over x (with x_1 being the least significant bit), and assume that $A = \{a,b\}$, $\mathbf{lookup}(x,a) = 011$ and

[3] In [13] a form of *garbage collection* is applied, where enumerations for values that no longer affect the checked property are reclaimed for later reuse. This involves a more complicated enumeration mechanism.

lookup$(x,b) = 001$. Then **build**(x,A) is a BDD representation of the Boolean function $x_1 \wedge \neg x_3$.

Intersection and union of sets of assignments are translated simply to conjunction and disjunction of their BDD representation, respectively, and complementation becomes BDD negation. We will denote the Boolean BDD operators as **and**, **or** and **not**. To implement the existential (universal, respectively) operators, we use the BDD existential (universal, respectively) operators over the Boolean variables that represent (the enumerations of) the values of x. Thus, if B_φ is the BDD representing the assignments satisfying φ in the current state of the monitor, then **exists**$(\langle x_1,\ldots,x_k\rangle, B_\varphi)$ is the BDD that represents the assignments satisfying $\exists x\ \varphi$ in the current state. Finally, BDD$(\bot)$ and BDD$(\top)$ are the BDDs that return always 0 or 1, respectively.

The BDD based algorithm The algorithm shown below extends the algorithm for the propositional case shown in [18]. It is based on the observation that the semantics of a formula in the current step can be cast in terms of the semantics of its subformulas in the current and the previous step. In particular, $\ominus\varphi$ holds in the current step if φ held in the previous step. The formula $(\varphi \mathcal{S} \psi)$ is equivalent to $(\psi \vee (\varphi \wedge \ominus(\varphi \mathcal{S} \psi)))$, which means that $(\varphi \mathcal{S} \psi)$ holds in the current step exactly if ψ holds now, or both φ holds now and $(\varphi \mathcal{S} \psi)$ held in the previous step. Thus, one only needs to look one step, or event, backwards in order to compute the new truth value of a formula. The algorithm operates on a summary of the sequence of events observed so far that consists of two vectors (arrays) of values indexed by subformulas: now calculated for the current event, and pre calculated for the the previous event. While in the propositional case [18] these vectors contain Boolean values, here they contain BDDs.

1. Initially, for each subformula φ of the specification η, $\mathsf{now}(\varphi) :=$ BDD$(\bot)$.
2. Observe a new event s (a set of ground predicates) 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 $\mathsf{now}(\psi)$.
   - $\mathsf{now}(\mathit{true}) :=$ BDD$(\top)$.
   - $\mathsf{now}(p(a)) :=$ if $p(a) \in s$ then BDD$(\top)$ else BDD$(\bot)$.
   - $\mathsf{now}(p(x)) :=$ **build**(x,A) where $A = \{a \mid p(a) \in s\}$.
   - $\mathsf{now}(\neg\varphi) :=$ **not**$(\mathsf{now}(\varphi))$.
   - $\mathsf{now}((\varphi \wedge \psi)) :=$ **and**$(\mathsf{now}(\varphi), \mathsf{now}(\psi))$.
   - $\mathsf{now}(\ominus\, \varphi) := \mathsf{pre}(\varphi)$.
   - $\mathsf{now}((\varphi\ \mathcal{S}\ \psi)) :=$ **or**$(\mathsf{now}(\psi),$ **and**$(\mathsf{now}(\varphi), \mathsf{pre}((\varphi \mathcal{S} \psi))))$.
   - $\mathsf{now}(\exists x\ \varphi) :=$ **exists**$(\langle x_1,\ldots,x_k\rangle, \mathsf{now}(\varphi))$.
5. if $\mathsf{now}(\eta) =$ BDD$(\bot)$ then report "error".
6. Goto step 2.

Example We shall illustrate the monitor generation using an example. Consider the following property stating that if a file f is closed, it must have been opened in the past with some access mode m (e.g. 'read' or 'write' mode):

$$\forall f\,(\mathit{close}(f) \longrightarrow \exists m\ \mathbf{P}\ \mathit{open}(f,m)) \tag{7}$$

Figure 6 (left) shows the monitor evaluation function generated by DEJAVU for this property. It relies on the enumeration of the subformulas shown in the Abstract Syntax Tree (AST) in Figure 6 (right). Two arrays are declared, indexed by subformula indexes: pre for the previous state and now for the current state, although here storing BDDs instead of Boolean values as in [18]. For each observed event, the function evaluate() computes the now array from highest to lowest index, and returns true (property is satisfied in this position of the trace) iff now(0) is not BDD($\bot$). 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) corresponding to the law: $\mathbf{P}\varphi = (\varphi \vee \ominus \mathbf{P}\varphi)$. As can be seen, for each new event, the evaluation of a formula results in the computation of a BDD for each subformula.

```
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)
    val result = !now(0).isZero
    tmp = now; now = pre; pre = tmp
    result
  }
}
```

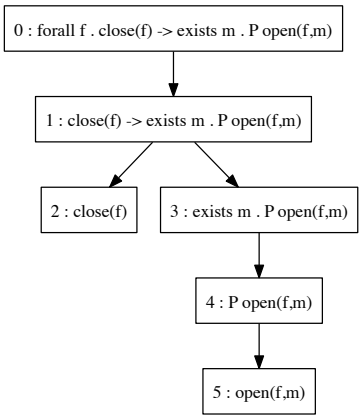
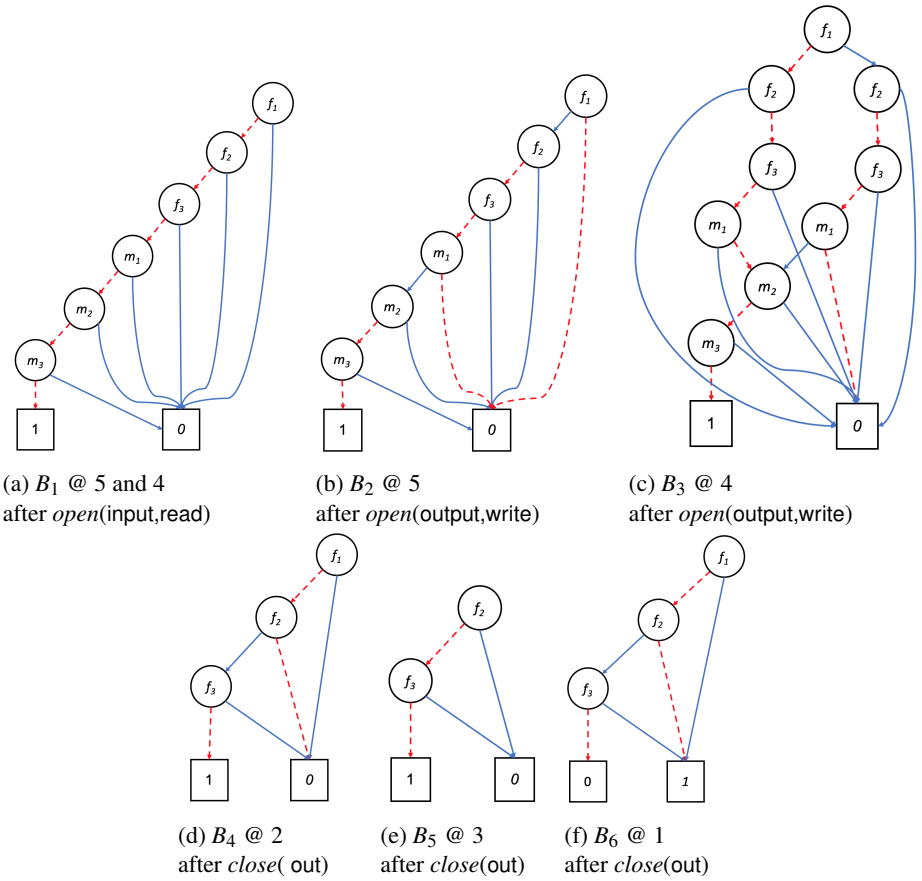


Fig. 6: Monitor (left) and AST (right) for the property.

We shall evaluate the example formula on a trace. Assume that each variable f and m is represented by three BDD bits: $f_3f_2f_1$ and $m_3m_2m_1$ respectively, with f_1 and m_1 being the least significant bits. Consider the input trace, consisting of three events:

$$\langle \{open(input, read)\}, \{open(output, write)\}, \{close(out)\} \rangle \tag{8}$$

When the monitor evaluates subformula 5 on the first event $open(\text{input}, \text{read})$, 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,.... For this first event the bit pattern $f_3f_2f_1$ is therefore mapped to 000 and the bit pattern $m_3m_2m_1$ is mapped to 000 as well. Hence, the assignment [f $\mapsto$ input, m $\mapsto$ read] is



(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. 7: Selected BDDs, named $B_1, \ldots, B_6$, computed after each event at various subformula nodes, indicated by B_i @ *node* (see Figure 6), during processing of the trace $\langle\{open(\mathsf{input,read})\}, \{open(\mathsf{output,write})\}, \{close(\mathsf{out})\}\rangle$.

represented by the concatenation of the two bit strings $m_3m_2m_1f_3f_2f_1 = 000000$, 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 7a shows the corresponding BDD B_1. In this BDD all the 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 observing that node 4 (all the seen values in the past) assumes the same BDD value as node 5, and conclude that since no *close*(...) event has been observed, the top-level formula (node 0) holds at this position in the trace.

Upon the second *open*(output,write) 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 for each variable). The new combined bit string for the assignments satisfying subformula 5 then becomes $m_3m_2m_1f_3f_2f_1 =$

001001, forming a BDD representing the assignment [f $\mapsto$ output, m $\mapsto$ write], and appearing in Figure 7b 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 assignments observed so far (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 $f_3 f_2 f_1$ = 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 m_1, m_2 and m_3 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 the possible bit patterns for f except for 010, which corresponds to the value out. This means, however, that the top-level formula in node 0 is not true (it is violated by bit pattern 010), and hence the formula is violated on the third event.

6 Using BDDs for Runtime Verification with Time

The last extension of the logic we shall consider in this paper is to allow properties to refer to the progress of time. The reported events are now assumed to appear with an integer timing value. We leave open the unit of measurement for time values (milliseconds, seconds, minutes, etc.). An example of such a specification is

$$\forall f\,(closed(f) \rightarrow \mathbf{P}_{\leq 20}\, open(f)) \tag{9}$$

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

6.1 Syntax

The syntax for first-order past-time LTL with time is as follows, where we have added two new formulas, each referring to a time constraint, a natural number $\delta \geq 0$:

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

The formula $(\varphi \mathcal{S}_{\leq\delta} \psi)$ has the same meaning as $(\varphi \mathcal{S} \psi)$, except that ψ must have occurred within δ time units. The formula $(\varphi \mathcal{S}_{>\delta} \psi)$ has the same meaning as $(\varphi \mathcal{S} \psi)$, except that ψ must have occurred more than δ time units ago. Other operators can be added, as shown in [14]. In addition to the previously defined derived operators we can define derived *timed* operators as follows: $\mathbf{P}_{\leq\delta}\varphi = (true \mathcal{S}_{\leq\delta} \varphi)$, $\mathbf{P}_{>\delta}\varphi = (true \mathcal{S}_{>\delta} \varphi)$, $\mathbf{H}_{\leq\delta}\varphi = \neg\mathbf{P}_{\leq\delta}\neg\varphi$, and $\mathbf{H}_{>\delta}\varphi = \neg\mathbf{P}_{>\delta}\neg\varphi$.

6.2 Semantics

A *timed event* is a pair (e,t) consisting of an event e and a time stamp t (a natural number). A *trace* $\sigma = (e_1,t_1)(e_2,t_2)\ldots(e_n,t_n)$ is a finite sequence of timed events, enumerated from 1. We denote the i^{th} timed event (e_i,t_i) in σ by $\sigma[i]$. We let $\sigma_e[i]$ denote the event e_i and we let $\sigma_t[i]$ denote the time t_i.

We define the predicate semantics for the two new timed operators below. The semantic equations for the remaining formulas are as shown in Section 5.2, although defined on timed traces, and where $\sigma[i]$ should be read as $\sigma_e[i]$.

- $(\gamma,\sigma,i) \models (\varphi\ \mathcal{S}_{\leq\delta}\ \psi)$ iff there exists $1 \leq j \leq i$ such that $\sigma_t[i] - \sigma_t[j] \leq \delta$ and $(\gamma,\sigma,j) \models \psi$, and for all $j < k \leq i$ it holds that $(\gamma,\sigma,k) \models \varphi$.
- $(\gamma,\sigma,i) \models (\varphi \mathcal{S}_{>\delta} \psi)$ iff there exists $1 \leq j < i$ such that $\sigma_t[i] - \sigma_t[j] > \delta$ and $(\gamma,\sigma,j) \models \psi$, and for all $j < k \leq i$ it holds that $(\gamma,\sigma,k) \models \varphi$.

6.3 Algorithm

We describe now *changes* to the algorithm in Section 5.3 for handling the two new formulas with the timing constraints. Recall that for each subformula φ of a formula, the algorithm in Section 5.3 updates two array positions: $\mathsf{now}(\varphi)$ and $\mathsf{pre}(\varphi)$. These BDDs represent assignments to free variables that occur in the formula, represented as a concatenated bit vector of the binary enumerations of the values assigned to the BDD variables: $x_k^n \ldots x_1^n \ldots x_k^1 \ldots x_1^1$. To keep track of the time, each such bit vector is augmented with a bit vector $t_m \ldots t_1$ being the binary code of the time that has passed since that assignment was observed, obtaining $t_m \ldots, t_1 x_k^n \ldots x_1^n \ldots x_k^1 \ldots x_1^1$. $x_k^n \ldots x_1^n \ldots x_k^1 \ldots x_1^1$
We add two new arrays, $\tau\mathsf{pre}(\varphi)$ and $\tau\mathsf{now}(\varphi)$, which for each subformula records the BDDs that includes these time values. These BDDs are then used to compute $\mathsf{now}(\varphi)$ and $\mathsf{pre}(\varphi)$ by removing the time values (by existential quantification over the time values).

Example We add a timing constraint to the formula (7), stating that when a file is closed it must have been opened within 3 time units in the past:

$$\forall f\,(close(f) \longrightarrow \exists m\ \mathbf{P}_{\leq 3}\ open(f,m)) \tag{10}$$

Let us apply this property to the the following trace, which is the trace (8) augmented with the time values 1, 2, and 3 respectively. We keep the time constraint and time values small and consecutive to keep the BDD small for presentation purposes:

$$\langle(\{open(input,read)\},1),(\{open(output,write)\},2),(\{close(out)\},3)\rangle \tag{11}$$

The BDD for the subformula $\mathbf{P}_{\leq 3}\ open(f,m)$ at the third event $close(out)$, shown in Figure 8, reflects that two (010 in binary) time units have passed since $open(input,read)$ occurred (follow leftmost path), and one time unit (001 in binary) has passed since $open(output,write)$ has occurred (follow rightmost path). The BDD is effectively an augmentation of the BDD in Figure 7c, with the additional three BDD variables t_1, t_2, and t_3 for the timer values, with t_1 being the least significant bit.

The BDD-based algorithm with time When a new event occurs, for a subformula with a timing constraint δ, we need to update the timers in $\tau\mathsf{now}$ that count the time that has passed since a tuple (assignment) of values satisfying the formula was observed. The difference between the clock value of the current event and the clock value of the previous one is Δ. In order to keep the representation of time small, $2\delta+1$ is the biggest

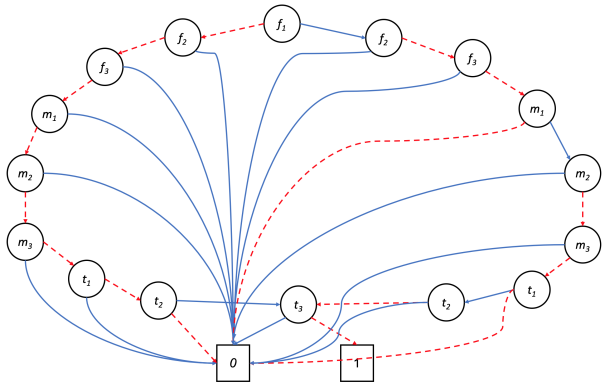


Fig. 8: The BDD for the formula $\mathbf{P}_{\leq 3}\ open(f,m)$ at the third event.

value of t that is stored. To see that this this is sufficient, and necessary, consider the following. First, during computation, when we observe a Δ that is bigger than δ, we cut it down to $\delta+1$ before we add to t. This is valid since we just need to know that it passed δ. Second, after we add Δ to t, we compare the new t against δ, and if now t goes beyond δ we can store just $\delta+1$. Finally, since adding $\Delta=\delta+1$ to a $t\leq\delta$ (since we only add Δ if $t\leq\delta$) gives max $2\delta+1$, then this is the biggest number we need to store in a BDD. Consequently, the number of bits needed to store time for a formula with a time constraint δ is $log_2(2\delta+1)$.

The algorithm for first-order past-time LTL with time is obtained by adding the statements below to step 4 of the algorithm shown in Section 5.3, and by adding the update $\tau\mathsf{pre} := \tau\mathsf{now}$ in step 2.

Algorithm for $(\varphi\ \mathcal{S}_{\leq\delta}\ \psi)$**:** We will use BDD0$(x)$ to denote the BDD where all the x_i bits are a constant 0, representing the Boolean expression $\neg x_1 \wedge \ldots \wedge \neg x_k$. The statements updating $\tau\mathsf{now}$ and now are as follows.

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

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

Algorithm for $(\varphi\mathcal{S}_{>\delta}\psi)$**:** We will use $EQ(t,c)$ to denote that the bit sting t is equal to c. This is technically defined as $EQ(t,c) = \exists z_1\ldots z_m\,(\mathrm{BDD0}(z)\wedge \mathit{addconst}(z,\ t,c))$, stating that $z=0$ added to c yields t. The updates to $\tau\mathsf{now}$ and now are as follows.

$$\begin{aligned}&\tau\mathsf{now}(\varphi\mathcal{S}_{>\delta}\psi) :=\\ &\quad(\mathsf{now}(\psi)\wedge(\neg\mathsf{pre}(\varphi\mathcal{S}_{>\delta}\psi)\vee\neg\mathsf{now}(\varphi))\wedge\mathrm{BDD0}(t))\ \vee\end{aligned}$$

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

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

References

1. L. de Alfaro, M. Z. Kwiatkowska, G. Norman, D. Parker, R. Segala, Symbolic Model Checking of Probabilistic Processes Using MTBDDs and the Kronecker Representation, TACAS, LNCS Volume 1785, Springer, 2000, 395-410.
2. B. Alpern, F. B. Schneider, Recognizing Safety and Liveness. Distributed Computing 2(3), 1987, 117-126.
3. R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, F. Somenzi, Algebraic Decision Diagrams and Their Applications. Formal Methods in System Design 10(2/3), 1997, 171-206 .
4. D. A. Basin, F. Klaedtke, S. Müller, E. Zalinescu, Monitoring Metric First-Order Temporal Properties, Journal of the ACM 62(2), 2015, 1-45.
5. S. Bensalem, K. Havelund, Dynamic Deadlock Analysis of Multi-threaded Programs, Haifa Verification Conference, LNCS Volume 3875, Springer, 2006, 208-223.
6. R. E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers 35(8), 1986, 677-691.
7. R. E. Bryant, Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, ACM Computing Survey 24(3), 1992, 293-318.
8. 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), 1991, 205-213.
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. E. M. Clarke, O. Grumberg, D. Peled, Model checking, MIT Press 2001.
11. E. M. Clarke, K. L. McMillan, X.g Zhao, M. Fujita, Jerry C.-Y. Yang, Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping. Formal Methods in System Design 10(2/3), 1997, 137-148.
12. D. M. Gabbay, A. Pnueli, S. Shelah, J. Stavi, On the Temporal Basis of Fairness. POPL, ACM Press 1980, 163-173.
13. K. Havelund, D. Peled, BDDs on the Run. ISoLA, LNCS Volume 11247, Springer, 2018, 58-69.
14. K. Havelund, D. Peled, First-Order Timed Runtime Verification using BDDs, ATVA, LNCS, Springer, to appear, 2020.
15. K. Havelund, D. Peled, D. Ulus, First-Order Temporal Logic Monitoring with BDDs, FMCAD, IEEE 2017, 116-123.

16. K. Havelund, D. Peled, D. Ulus, First-Order Temporal Logic Monitoring with BDDs. Formal Methods in System Design, published online 7 January, 2019, 1-21.
17. K. Havelund, G. Reger, D. Thoma, E. Zălinescu, Monitoring Events that Carry Data, book chapter, LNCS Volume 10457, Springer, 2018, 61-102.
18. K. Havelund, G. Rosu, Synthesizing Monitors for Safety Properties, TACAS, LNCS Volume 2280, Springer, 2002, 342-356.
19. JavaBDD, http://javabdd.sourceforge.net.
20. O. Kupferman, M. Y. Vardi, Model Checking of Safety Properties, Formal Methods in System Design 19(3), 2001, 291-314.
21. C. Y. Lee, Representation of Switching Circuits by Binary-Decision Programs, Bell Systems Technical Journal, 38, 1959, 985-999.
22. Z. Manna, A. Pnueli, Completing the Temporal Picture, Theoretical Computer Science 83, 1991, 91-130.
23. Shin-ichi Minato: Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. Design Automation Conference, ACM/IEEE, 1993, 272-277.