Monitoring First-Order Interval Logic*

Klaus Havelund[1], Moran Omer[2], and Doron Peled[2]

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

Abstract. Runtime verification is used for monitoring the execution of systems, e.g. checking sequences of reported events against formal specifications. Often, the specification refers to the monitored events. We take a different approach, and suggest a formalism akin to Allen's temporal logic. Allen's logic allows assertions about the relationship between concretely named intervals. Our formalism modifies Allen's logic into a first-order logic that allows quantification over intervals. Intervals can carry data as well. We allow referring to these explicitly, as well as comparing them with equality. We provide a monitoring algorithm and describe an implementation and experiments performed with it.

1 Introduction

Runtime verification allows monitoring of system executions, represented as execution traces, against a specification, either online as traces are generated, or offline after their generation. The monitored trace consists typically of events that can also carry data. The specification is often given using a temporal logic or as a state machine. The runtime algorithm checks for compatibility with the execution in an incremental way, where some summary of the reported execution prefix is updated upon the arrival of newly occurring event. This practice is aimed at both providing an early verdict, and at managing the incremental computational effort between consecutive events. The inclusion of data in the monitored events presents a challenge to the online monitoring algorithms, as the incremental computation performed between the arrival of events has to keep pace with the speed of the reported events.

While runtime verification, as described above, is concerned with monitoring specifications that refer to observed *events*, we study here monitoring specifications that refer to observed *intervals*. Writing specifications referring to intervals can be more succinct than writing specifications referring to events. *Allen's (temporal) logic* [1], also referred to as *Allen's interval algebra*, is a popular formalism for reasoning about the relation between intervals that occur on a timeline. It is often used for planning in AI. Allen's logic deals with a finite set of named intervals, referring directly to the interval names, e.g., $A < B$ means that the interval A must end before the interval B begins. This can be

* 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 and third author was partially funded by Israeli Science Foundation grant 1464/18: "Efficient Runtime Verification for Systems with Lots of Data and its Applications". © 2021. All rights reserved.

quite restrictive for describing the behavior of systems where many intervals with the same characteristics can occur, and where distinguishing specific intervals directly by name in the specification is inconvenient or even impossible.

We look at the more general problem of monitoring properties where we can *quantify over intervals*, as e.g. in the formula: $\exists A \exists B\ (A < B)$, stating that there exist (at least) two intervals A and B such that A ends before B begins. We also consider the problem where intervals may contain data. Note, however, that the logic we propose is not first-order w.r.t. data in that it does not support quantification over data. It does, however, allow to refer to data explicitly, as constants, as well as comparing different intervals with the same data values. Consequently, the logic allows expressing cases that involve relations between intervals that are embedded in the trace with many, sometimes irrelevant, intervals in between. The runtime verification allows "pattern matching" against these cases in a monitored trace.

We present a matching runtime verification algorithm. The algorithm decides whether any prefix of the execution (the currently observed trace) satisfies the specification. The runtime verification is based on updating a summary of the observed prefixes upon the arrival of each new interval *begin* and *end* event. The trick we employ is to maintain several sets of interval identifiers, and tuples of such, corresponding to the different Allen operators. These variables record those intervals and relations that have begun and not completed yet, as well as those intervals and relations that have been completed. For example, a *begin* event for one interval A followed by a *begin* event for another interval B, is stored (in a variable) as a potential for an A interval, as well as (in a different variable) a potential for an A interval overlapping with a B interval (where A starts, then B starts, then A ends and then B ends). An occurrence of an *end* event for A and then an *end* event for B will complete the picture to decide that A overlaped with B, as well as, of course, having seen completed A and a B intervals.

Our logic and runtime verification algorithm is implemented in the tool MonAmi[1]. The implementation encodes interval identifiers and data as bit vectors, which are then represented as BDDs. The bit vectors are obtained by a simple enumeration scheme. Such BDDs are useful for compacting interval identifiers and data when storing them in sets, and also makes negation (set complement) non-problematic.

Related Work The use of BDDs in runtime verification has been explored in [12] for the first-order past time temporal logic DejaVu, which is an event logic, in contrast to the interval logic explored here. However, the enumeration scheme for creating bit vectors from data and then converting them to BDDs is similar. Numerous event logics have been developed during the past two decades, including [15, 18, 4, 22, 10, 9, 5, 3, 12] to mention just a few.

Monitoring of Allen logic is explored in [23]. In that logic, however, intervals are referred to by explicit names, such as $A < B$. This means that one can only specify static patterns, one occurrence of a particular pattern: that there is one A and one B, such that $A < B$. This is in contrast to MonAmi, where we can quantify over such intervals. Specifically this means that we can specify repeated patterns in the trace e.g., that every

[1] **Mon**itoring **A**llen logic **m**odal **i**ntervals.

interval A with some specific data d is always followed by some other interval B with some data d'.

The most closely related monitoring system is nfer [14], also influenced by Allen's logic. But its specification formalism consists of Prolog-like interval-generating rules (see, e.g., Figure 5). The objective of nfer is to *generate* intervals from a trace of events, as an abstraction of the trace, to e.g. support trace comprehension by humans. Generated intervals can, for example, be visualized. nfer supports Boolean conditions over data as well as computations on data, resulting in new data being stored in the generated intervals. In order to reduce computational complexity, nfer operates in its default mode with a minimality principle, where the before-operator (MonAmi's $<$ operator) only matches the smallest intervals, whereas MonAmi matches all candidate intervals. Section 5 compares MonAmi with nfer further.

A different kind of extension to Allen's logic, where the various relations between operators are promoted into modalities was suggested by Halpern and Shoham [11].

2 Preliminaries

To motivate the study of interval-based specification, we first present the original *Allen Temporal Logic* (ATL).

Syntax. In its basic form, ATL has the following syntax:

$$\varphi ::= (\varphi \wedge \varphi) \mid \neg\varphi \mid A < B \mid A\,m\,B \mid A\,o\,B \mid A\,s\,B \mid A\,d\,B \mid A\,f\,B \mid A = B$$

where A and B are *intervals* from a finite set of intervals $\mathfrak{I}$, m stands for *meets*, o for *overlaps*, s for *starts*, d for *during*, and f for *finishes*. The original definition of the logic also includes the symmetric versions of these operators, e.g., an operator for $A\,mi\,B$ for $B\,m\,A$, etc.

Semantics. A *model* $M = \langle E, \prec, \asymp \rangle$ for Allen's logic, consists a finite set of events $E = \{begin(A) \mid A \in \mathfrak{I}\} \cup \{end(A) \mid A \in \mathfrak{I}\}$, a linear order $\prec \subseteq E \times E$, and an equivalence relation $\asymp \subseteq E \times E$, where $\preceq = (\prec \cup \asymp)^*$ (the transitive closure of the union of the two relations), such that:

- For each $A \in \mathfrak{I}$, $begin(A) \prec end(A)$.
- $\asymp$ is a partition of the set E into equivalence classes.
- $(\prec \cap \asymp) = \emptyset$.
- For every $a, b \in E$, either $a \preceq b$ or $b \preceq a$.

Thus, M is a linear order between equivalence classes. We call the relation $\prec$ *before*, and $\asymp$ *coincides*. The semantics is given as follows.

- $M \models (\varphi \wedge \psi)$ if $M \models \varphi$ and $M \models \psi$.
- $M \models \neg\varphi$ if $M \not\models \varphi$.
- $M \models A < B$ if $end(A) \prec begin(B)$.
- $M \models A\,m\,B$ if $end(A) \asymp begin(B)$.
- $M \models A\,o\,B$ if $begin(A) \prec begin(B) \prec end(A) \prec end(B)$.

- $M \models A\, s\, B$ if $begin(A) \asymp begin(A)$ and $end(A) \prec end(B)$.
- $M \models A\, d\, B$ if $begin(B) \prec begin(A)$ and $end(A) \prec end(B)$.
- $M \models A\, f\, B$ if $begin(B) \prec begin(A)$ and $end(A) \asymp end(B)$.
- $M \models A = Y$ if $begin(A) \asymp begin(B)$ and $end(A) \asymp end(B)$

As usual, we can define additional operators, in particular, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$ and $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$. The semantics is illustrated in Figure 1. As an example, consider then the ATL formula:

$$((B_1\, d\, L \wedge B_2\, d\, L) \wedge B_1 < B_2) \tag{1}$$

It asserts about three intervals B_1, B_2 and L, that B_1 appears before B_2 and both are embedded within L. Monitoring Allen's logic is described in [23].

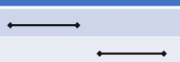
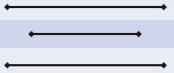
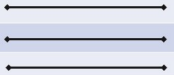
Relation	Visually	Read as
A<B		A precedes B
A m B		A meets B
A o B		A overlaps B
A s B		A starts B
A d B		A during B
A f B		A finishes B
A = B		A equals B

Fig. 1: Illustration of ATL's semantics.

3 A First-Order Interval Logic

We will explore now the monitoring of a first-order logic variant of Allen's temporal logic, which we term FoATL. While the original logic refers to a fixed set of named intervals, our variant allows quantification over the intervals that occur in the trace, which can optionally carry data. The logic allows also to relate different intervals with respect to their data values. The formalism supports monitoring of behaviors consisting of a large, perhaps unbounded, number of intervals, where patterns of behavior that consist of intervals are related in specified ways. For example, a relationship such as in Equation (1) can refer to any embedding within a sequence of intervals, matching this pattern, rather than referring to three particular intervals that appear in the input.

The setting. We monitor a sequence of events of the form $begin(z)$ and $end(z)$, where z is a sequence of parameters. The first parameter is an *interval enumeration* (also referred to as interval *id*), used to identify matching *begin* and *end* events and the rest of the parameters, which can be of different types, is optional. The second parameter can be e.g., a label representing the kind of interval, where a label *Boot* represents that it is a *boot* interval. For example, consider the sequence of events:

$$begin(1,Load), begin(2,Boot), end(2,Boot), begin(3,Boot), end(3,Boot), end(1,Load)$$

These events form three intervals, as depicted in Figure 2. These intervals correspond to the intervals L, B_1, and B_2 appearing in ATL Formula (1).

Our logic alters Allen's logic by adding quantification over the intervals. Hence, instead of fixed intervals, which can be referred to in a formula by their explicit name as constants, we allow interval *variables* $A, B, \ldots$ that can be instantiated to any of the intervals that appear in the model (the observed trace). Moreover, the intervals can carry data, and we write in the logic $A(d)$ to denote that the data of the interval assigned to the variable A has the constant value d. We can also verify whether two intervals A and B carry the same value using $same(A,B)$.

We make a few simplifying assumptions in order to concentrate on the main challenges of runtime verification of a first-order interval logic. However, the presented approach is extensible and the restrictions can be easily removed:

- We assume a matching unique integer value per interval, an *enumeration*, though it does not have to appear in consecutive order, is given for each related pair of events, e.g., $begin(5)$ and $end(5)$.
- Events can contain additional parameters besides the enumeration. For simplicity, we assume that there is only a *single* data value parameter, e.g., an integer or a string, and that it appears matched on both, e.g., $begin(5,abc)$ and $end(5,abc)$, events. In a more general setting, different number of parameters can appear for different intervals, and the parameters may appear only once, at the beginning or at the end.
- The monitored events appear one at a time. As there is no co-incidence of events, the relations are restricted to $A < B$ (before), $A\,o\,B$ (overlaps) and $A\,i\,B$ (for *includes*, which is the symmetric operator of Allen's *d during*). Hence, there is a total order between the events.
- As in Allen's logic, the specification does not refer to intervals that were opened with $begin(A)$ and were not closed yet with $end(A)$. The logic can of course be extended accordingly.
- We assume that as part of the monitoring, the restrictions on well formedness of the enumerations are checked. Multiple $begin(A)$ or $end(A)$ events cannot occur, and an $end(A)$ event cannot precede a $begin(A)$ event.
- We allow referring to the data elements in intervals, and also compare them. We offer in the syntax (and our implementation) the predicate *same* that relates intervals with the same data value. This can be extended to other relations that compare values.
- Quantification is applied to the (completed) intervals that have occurred. We do not allow adding "virtual" intervals that do not refer to $begin(A)$ and $end(A)$ events in the input.

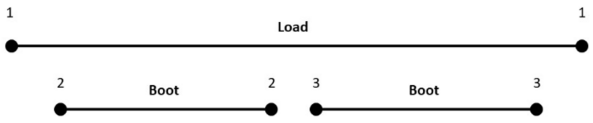


Fig. 2: Three intervals, with enumerations and an additional Boolean parameter.

Syntax of FoATL**.** The syntax is as follows.

$$\varphi ::= (\varphi \wedge \varphi) \mid \neg\varphi \mid A(d) \mid (A < B) \mid (A\, o\, B) \mid (A\, i\, B) \mid \exists A\varphi \mid same(A,B)$$

where A and B are variables (representing intervals) from a set of *interval* variables $\mathfrak{J}$, and d is a value from some fixed domain D of data values. Parentheses can be removed when clear from the context. Consider for example the following formula:

$$\exists A\, \exists B\, \exists C\, (A(Load) \wedge B(Boot) \wedge C(Boot) \wedge A\, i\, B \wedge A\, i\, C \wedge B < C).$$

This specification describes the existence of three intervals with the same relations between them as appearing in Figure 2.

Semantics of FoATL**.** Let $\mathfrak{J}$ be the finite set of *interval* variables over the enumerations in the observed execution prefix, and V the set of *data* variables, over the data domain D. We assume the following semantic components:

- $\sigma = e(1)e(2)\ldots e(n)$ is a sequence of events of form $begin(3)$ and $end(3)$ as described above.
- $\rho : \mathfrak{J} \mapsto Nat$ is a mapping from the interval variables $\mathfrak{J}$ to the natural numbers, representing interval enumerations.
- $data(j)$ is the data value associated with the interval whose enumeration is j.
- $start(j)$ is the number (position in the trace) of the event that starts the interval with enumeration j, i.e., the event $begin(j)$ (with an optional additional data value).
- $finish(j)$ is the number (position in the trace) of the event that ends the interval with enumeration j, i.e., the event $end(j)$ (with an optional additional data value).

We can now define the semantics of the logic inductively on the structure of the formula.

- $(\rho,\sigma) \models (\varphi \wedge \psi)$ if $(\rho,\sigma) \models \varphi$ and $(\rho,\sigma) \models \psi$
- $(\rho,\sigma) \models \neg\varphi$ if $(\rho,\sigma) \not\models \varphi$.
- $(\rho,\sigma) \models A(d)$ if $\rho(A) = j$ and $data(j) = d$.
- $(\rho,\sigma) \models (A < B)$ if $\rho(A) = j$ and $\rho(B) = k$ and $finish(j) < start(k)$.
- $(\rho,\sigma) \models (A\, o\, B)$ if $\rho(A) = j$ and $\rho(B) = k$ and $start(j) < start(k) < finish(j) < finish(k)$.
- $(\rho,\sigma) \models (A\, i\, B)$ if $\rho(A) = j$ and $\rho(B) = k$ and $start(j) < start(k) < finish(k) < finish(j)$.
- $(\rho,\sigma) \models \exists A\varphi$ if there exists j such that $\rho' = \rho[j/A]$ and $(\rho',\sigma) \models \varphi$.
- $(\rho,\sigma) \models same(A,B)$ if there exists $d \in D$ such that $\rho(A) = j$ and $\rho(B) = k$ and $data(j) = data(k)$.

Example properties.

- $\neg\exists A \exists B\,(A < B \wedge \mathit{same}(A,B))$.
  Totally ordered intervals cannot have the same data value.
- $\neg\exists A \exists B \exists C\,((A\,i\,B \wedge B\,i\,C))$.
  No double nesting of intervals.
- $\forall A \forall B\,((A < B \wedge (\neg\exists C(A < C \wedge C < B))) \rightarrow \neg(A(2) \wedge B(2)))$.
  No two adjacent intervals (one completely after the other without any interval in between) can have both the same value 2.
- $\forall A \forall B \forall C\,(((A\,o\,B) \wedge (B\,o\,C)) \rightarrow \neg(A\,o\,C))$.
  At no point there is an overlapping of three intervals.

Interpretation. One can interpret the semantics of a formula over finite or infinite sequences. As the logic is tailored with an application of runtime verification in mind, our intended use is to require that for a given (monitored observed) trace, the requirement will be that all prefixes will satisfy a given FoATL specification. This is a similar to the common use of temporal properties of the form $\Box\varphi$, where φ is restricted to past modalities, in runtime verification, see, e.g., [13, 12], i.e., to *safety properties* [2]. Nevertheless, our implementation returns a truth value for the inspected property for each prefix of the monitored trace. Note that satisfaction of a property over an infinite trace does not entail that it is satisfied by all finite prefixes, e.g., for $\varphi = \forall A \exists B\,(A < B)$, which asserts that there is no *biggest* interval. Conversely, $\neg\varphi$ is satisfied by every finite trace that includes at least one interval, but will not hold for a trace with infinitely linearly ordered intervals.

4 The Monitoring Algorithm

Calculating the Relations between Intervals. Recall that in our setting, we are restricted to three possible relations between intervals: $<$, o, and i. Let X and Y be different intervals, defined by *begin* and *end* events, that appeared in the current observed monitored prefix. We distinguish the following three sets of pairs (X,Y) of enumerations of intervals.

- $X < Y$ (*before*). Events appear in the order $\mathit{begin}(X), \mathit{end}(X), \mathit{begin}(Y), \mathit{end}(Y)$.
- $X\,o\,Y$ (*overlaps*). Events appear in the order $\mathit{begin}(X), \mathit{begin}(Y), \mathit{end}(X), \mathit{end}(Y)$.
- $X\,i\,Y$ (*includes*). Events appear in the order $\mathit{begin}(X), \mathit{begin}(Y), \mathit{end}(Y), \mathit{end}(X)$.

We maintain for each prefix of an execution three sets of pairs of enumerations, $XXYY$ for $X < Y$, $XYXY$ for $X\,o\,Y$ and $XYYX$ for $X\,i\,Y$. Further sets of pairs (X,Y) correspond to possible prefixes of the four events in the above three cases, namely XY, XYY, XYX and XXY. We further define the set X of enumerations for events $\mathit{begin}(X)$ where an $\mathit{end}(X)$ has not yet appeared and XX as the set of enumerations where both $\mathit{begin}(X)$ and $\mathit{end}(X)$ have occurred; thus latter is the set of completed intervals. Together, this defines two sets of enumerations, and seven sets of pairs. We define these sets inductively on the length i of the trace. For $i = 0$, all the sets are empty. Then the update of

these sets after ith event is defined according to Table 1. The rows correspond to the sets that are updated, and the columns to the ith event. The entry details how the set is updated after the ith event based on the values of the prior values of the sets. For example, for the set X (containing the open intervals), if the ith event is a $begin(Z)$ (or $begin(Z,d)$), then $X_i = X_{i-1} \cup \{Z\}$, and if the ith event is an $end(Z)$ (or $end(Z,d)$), then $X_i = X_{i-1} \setminus \{Z\}$. Our algorithm follows the updates in Table 1 upon arrival of any new event. We denote by $\mathcal{U}$ the universal set of enumerations. The empty set is denoted by $\emptyset$. We denote by $\overline{S}$ the complement of S, i.e., the set $\mathcal{U} \setminus S$. We will describe later how to implement these sets and operations using BDDs.

The following rules impose validity checks on the order of the $begin(Z,d)$ (d, the data value, is optional) and $end(Z,d)$ events, causing the system to halt when violated. Specifically, they disallow multiple $begin(Z,d)$, $end(Z,d)$, and an $end(Z,d)$ that appears before the corresponding $begin(Z,d)$.

$begin(Z,d)$: If $\{Z\} \cap (X \cup XX) \neq \emptyset$ then "multiple begin".
$end(Z,d)$: If $\{Z\} \cap XX \neq \emptyset$ then "multiple end".
If $\{Z\} \cap X = \emptyset$ then "intervals ends before it begins".

Set \ Event	$begin(Z,d)$	$end(Z,d)$
X (opened)	$X \cup \{Z\}$	$X \cap \overline{\{Z\}}$
XX (closed)		$XX \cup \{Z\}$
XY	$XY \cup ((X \times \{Z\}))$	$XY \cap (\mathcal{U} \times \overline{\{Z\}}) \cap (\overline{\{Z\}} \times \mathcal{U})$
XYY		$(XYY \cap \overline{XYYX}) \cup (XY \cap (\mathcal{U} \times \{Z\}))$
$XYYX$ ($X\,i\,Y$, includes)		$XYYX \cup (XYY \cap (\{Z\} \times \mathcal{U}))$
XYX		$(XYX \cap \overline{XYXY}) \cup (XY \cap (\{Z\} \times \mathcal{U})$
$XYXY$ ($X\,o\,Y$, overlaps)		$XYXY \cup (XYX \cap (\mathcal{U} \times \{Z\}))$
XXY	$XXY \cup (XX \times \{Z\})$	$XXY \cap (\mathcal{U} \times \overline{\{Z\}})$
$XXYY$ ($X < Y$, before)		$XXYY \cup (XXY \cap (\mathcal{U} \times \{Z\}))$
XD (X has data d)		$XD \cup (Z,d)$

Table 1: The update table.

The order of updating the sets is important: a set that is a prefix of another set, e.g., XY is a prefix set of XYX, is updated *after* the latter. Thus, upon arrival of a new event, the value of XYX is updated based on the *old value* of XY, *before* updating XY.

In order to handle intervals with data, we add another set XD of pairs of the form (Z,d), where Z is an enumeration and d is a data element. Then, upon the arrival of an event of the form $end(Z,d)$, we update $XD := XD \cup \{(Z,d)\}$. This construction can be easily extended to capture a different number of parameters n by keeping sets of $n+1$ tuples.

Using BDDs to represent relations. Our algorithm is based on representing relations between data elements using Ordered Binary Decision Diagrams (OBDD, although we

write BDD) [6]. A BDD is a compact representation for a Boolean function (arguments as well as result are Booleans) as a directed acyclic graph (DAG), see Figure 4.

A BDD is obtained from a binary tree that represents a Boolean formula with some Boolean variables $x_1 \ldots x_k$ by gluing together isomorphic subtrees. Each non-leaf node is labeled with one of the Boolean variables. A non-leaf node x_i is the source of two arrows leading to other nodes. A dotted-line arrow represents that x_i has the Boolean value *false* (i.e., 0), while a thick-line arrow represents that it has the value *true* (i.e., 1). The nodes in the DAG have the same order along all paths from the root (hence the letter 'O' in OBDD). Nodes may be absent along some paths, when the result of the Boolean function does not depend on the value of the corresponding Boolean variable. Each path leads to a leaf node that is marked by either *true* or *false*, corresponding to the Boolean value returned by the function for the Boolean values on the path.

A Boolean function, and consequently a BDD, can represent a set of integer values as follows. Each integer value is, in turn, represented using a bit vector: a vector of bits $x_1 \ldots x_k$ represents the integer value $x_1 \times 1 + x_2 \times 2 + \ldots x_k \times 2^k$, where the bit value of x_i is 1 for *true* and 0 for *false* and where x_1 is the *least* significant bit, and x_k is the *most* significant. For example, the integer 6 can be represented as the bit vector 110 (the most significant bit appears to the left) using the bits $x_1 = 0$, $x_2 = 1$ and $x_3 = 1$. To represent a *set* of integers, the BDD returns *true* for any combination of bits that represent an integer in the set. For example, to represent the set $\{4,6\}$, we first convert 4 and 6 into the bit vectors 100 and 110, respectively. The Boolean function over x_1, x_2, x_3 is $(\neg x_1 \wedge x_3)$, which returns *true* exactly for these two bit vector combinations.

This representation can be extended to relations, or, equivalently, a set of tuples over integers. The Boolean variables are partitioned into n bitstrings $x^1 = x^1_1, \ldots, x^1_{k_1}$, $\ldots$, $x^n = x^n_1, \ldots, x^n_{k_n}$, each representing an integer number, forming the bit string[2]:

$$x^1_1, \ldots, x^1_{k_1}, \ldots, x^n_1, \ldots, x^n_{k_n}.$$

Using BDDs over enumerations of values. Representing data values such as strings and integers, which appear within the observed trace of events, may not lead to a good compact representation. Instead, based on the limited ability to compare data values allowed by our logic, we represent in the BDD *enumerations* for these values, rather than the values themselves. When a value (associated with a variable in the specification) appears for the first time in an observed event, we assign to it a new *enumeration*. Values can be assigned consecutive enumeration values[3]. We use a hash table to point from the value to its enumeration so that in subsequent appearances of this value the same enumeration will be used. For example, if the runtime verifier sees the input events $begin(1,a)$, $begin(1,b)$, $begin(1,c)$, it may encode the data a, b, and c as the bit vectors 000, 001, and 010, respectively. The approach results in several advantages:

1. It allows a shorter representation of very big values in the BDDs; the values are compacted into a smaller number of bits.

[2] In the implementation the same number of bits are used for all variables: $k_1 = k_2 = \ldots = k_n$.

[3] A refined algorithm can reuse enumerations that were used for values that can no longer affect the verdict of the RV process, see [12].

2. It contributes to the compactness of the BDDs because enumerations of values that are not far apart often share large bit patterns.
3. The monitoring algorithm is simple; the Boolean operators over summary elements: conjunction, disjunction, and negation, are replaced by the same operators over BDDs.
4. Given an efficient BDD package, the implementation can be very efficient. On can also migrate between BDD packages.
5. Full use of negation follows easily.

BDD Operations. We list now the operators on BDDs representing sets of value tuples, used in evaluating the verdict of the specification on the currently inspected prefix. Each BDD is defined over a sequence that can be partitioned to several sections, e.g., $x_1^1 \ldots x_n^1 y_1^2 \ldots y_n^2$ for event parameter tuples of length 2.

conj$(\mathcal{B}, \mathcal{C})$ The conjunction (intersection) of the BDDs $\mathcal{B}$ and $\mathcal{C}$. Disjunction (union, or database co-join) can be defined similarly.

comp$(\mathcal{B})$ The complement of the BDD $\mathcal{B}$.

project$(\mathcal{B}, X)$ Projects out the Boolean variables $x_1 \ldots x_n$ that correspond to the parameter X of $\mathcal{B}$, obtaining $\exists x_1 \ldots \exists x_n B$.

singleton(z, X) Returns a BDD that returns a 1 (true) exactly when the bits $x_1 x_2 \ldots x_n$ encode the binary value of z.

restrict$(d, \mathcal{B})$ Returns a BDD with bits $x_1 \ldots x_n$ that results from the BDD $\mathcal{B}$ of the form XD, i.e., with bits $x_1 \ldots x_n d_1 \ldots d_m$, where the data value component $d_1 \ldots d_m$ is z. This involves applying the existential quantification $\exists d_1 \ldots \exists d_m \mathcal{B}$.

rename$(\mathcal{B}, X \leftarrow X', Y \leftarrow Y', \ldots)$ Replaces the bits $x_1 x_2 \ldots x_n$ with $x_1' \ldots x_n'$, the bits $y_1 \ldots y_n$ by $y_1' \ldots y_n'$, etc. in the BDD $\mathcal{B}$.

Completing the algorithm. The algorithm for the complete logic starts with setting all the sets in Table 1 to BDDs representing the empty sets of elements/pairs, according to their types. Upon the arrival of each new event of the type *begin*(z) or *end*(z), with or without an additional data parameter d, two steps are executed.

Step 1: The sets of values/pairs are updated according to Table 1.
Step 2: BDDs of the form B_φ for the subformulas φ of the monitored property are updated recursively as follows:

- $\mathcal{B}_{(\varphi \wedge \psi)} = conj(\mathcal{B}_\varphi, B_\psi)$
- $\mathcal{B}_{\neg\varphi} = comp(\mathcal{B}_\varphi)$
- $\mathcal{B}_{A(d)} = rename(restrict(d, XD), X \leftarrow A)$
- $\mathcal{B}_{A<B} = rename(XXYY, X \leftarrow A, Y \leftarrow B)$
- $\mathcal{B}_{A\,o\,B} = rename(XYXY, X \leftarrow A, Y \leftarrow B)$
- $\mathcal{B}_{A\,i\,B} = rename(XYYX, X \leftarrow A, Y \leftarrow B)$
- $\mathcal{B}_{\exists A\varphi} = project(\mathcal{B}_\varphi, A)$
- $\mathcal{B}_{same(A,B)} = project(conj(XD[X \leftarrow A], XD[X \leftarrow B]), D)$

5 Implementation

We implemented a prototype monitoring tool [19] for our logic FoATL, called MonAmi. It is a Python-based tool for monitoring intervals, formed by events, by checking them against a FoATL property. The tool works with Python 3.6 and above. It uses the 'dd' Python package [8] for generating and manipulating BDDs, which itself uses the CUDD BDD package [7] in C. MonAmi uses several input files that define the configuration of the initial parameters, the trace file, and the property file, for monitoring in offline mode. A trace $\mathcal{T}$ is a sequence of events $[\text{begin}, i, d]$ or $[\text{end}, i]$, where i is an interval enumeration, and d is the data. The tool can also be used for online monitoring while observing a trace of a program during execution. Our implementation uses the following three functions:

1. $lookup(z)$ - z is searched in a hash table. If this is its first occurrence, then it will be assigned a new enumeration. Otherwise, $lookup(z)$ returns the enumeration that z received before. z can represent either an interval id or data. We manage separate hash tables I, $\mathcal{D}$ for interval ids and data, respectively. For example, assume the trace[4] $[begin, 1, BOOT], [end, 1]$ and assume 3 bits[5] for enumerations (x_1, x_2, x_3). For the first event $[begin, 1, \text{BOOT}]$, $lookup(1) = 000$ (i.e., $x_1 = 0$, $x_2 = 0$, $x_3 = 0$) and $lookup(\text{BOOT}) = 000$ (enumerations of interval ids and data are allocated independently). For the second event $[end, 1]$, $lookup(1) = 000$, since the interval id, 1 is already assigned to 000 in the previous event in hash table I.
2. $update(t, i, d)$ – this is the main function that is responsible for updating the map $\mathcal{P} : \mathcal{B} \rightarrow BDD$, where $\mathcal{B} = \{X, XX, XY, XXY, XXYY, XYX, XYXY, XYY, XYYX, XD\}$, with enumerations of the parameters of the current event, where $t \in \{begin, end\}$, $i \in I$, and $d \in \mathcal{D}$. The BDDs in $\mathcal{B}$ are initially *false*, the empty BDD with no truth assignments. The updating of the BDDs is as described in Table 1. As an example, for the above trace, $update(t, i, d)$ will update the BDDs as follows (BDDs are shown as the set of bit vectors they map to *true*):
   (a) For the first event $[begin, 1, BOOT]$:
      i. $\mathcal{P}[X] := \{[x_1 = 0, x_2 = 0, x_3 = 0]\}$
   (b) For the second event $[end, 1]$:
      i. $\mathcal{P}[X] := false$
      ii. $\mathcal{P}[XX] := \{[x_1 = 0, x_2 = 0, x_3 = 0]\}$
      iii. $\mathcal{P}[XD] := \{[x_1 = 0, x_2 = 0, x_3 = 0, d_1 = 0, d_2 = 0, d_3 = 0]\}$
      (Note that XD is updated when an end event occurs.)
3. $eval(\varphi, \mathcal{P}, \mathcal{D})$ - evaluates the property φ. The property is represented as an Abstract Syntax Tree (AST), which is evaluated bottom up. $eval(\varphi, \mathcal{P}, \mathcal{D})$ returns *true* when the property is satisfied, and *false* otherwise. Figure 3 shows an example of a property and its AST.

Given these definitions, MonAmi works as follows:

– For each event:

[4] In the implementation each event is represented as a Python list $[v_1, \ldots, v_n]$ of values.
[5] The number of bits is dynamically extendable upon need.

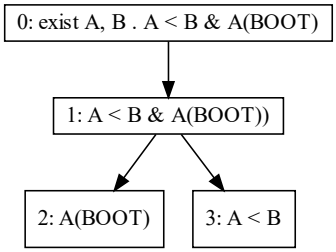


Fig. 3: Evaluation starts from the bottom (higher index) up (lower index).

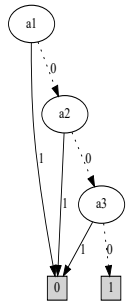
- Interval id and data are enumerated to bit-strings.
- The relevant BDDs in $\mathcal{P}$ are updated.
- The property is evaluated based on the current state defined by $\mathcal{P}$.

We demonstrate the algorithm with an example. Assume the trace $\mathcal{T} = [begin, 1, BOOT], [end, 1], [begin, 2, LOAD], [end, 2]$, and consider the property $\varphi = \exists A \exists B\ (A < B\ \wedge A(BOOT))$ (there is a boot, and some activity after that). For the first and the second event, the BDDs in $\mathcal{P}$ are updated as shown above, and the property evaluation returns *false*. Note that ':=' denotes an update, while '=' denotes the previous value of the BDD, which is also its current value.

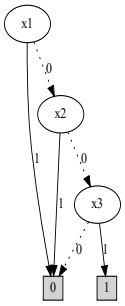
– For the third event [begin,2,LOAD] (BDDs for this step illustrated in Figure 4):
  - $lookup(2)$ returns 001 using the hash table I, as does $lookup(LOAD)$ using the hash table $\mathcal{D}$.
  - $update(begin, 001, 001)$
    * $\mathcal{P}[X] := \{[x_1 = 0, x_2 = 0, x_3 = 1]\}$
    * $\mathcal{P}[XX] = \{[x_1 = 0, x_2 = 0, x_3 = 0]\}$
    * $\mathcal{P}[XXY] := \{[x_1 = 0, x_2 = 0, x_3 = 0, y_1 = 0, y_2 = 0, y_3 = 1]\}$
    * $\mathcal{P}[XD] := \{[x_1 = 0, x_2 = 0, x_3 = 0, d_1 = 0, d_2 = 0, d_3 = 0]\}$
  - $eval(\varphi, \mathcal{P}, \mathcal{D})$
    * $A(BOOT) = \{[a_1 = 0, a_2 = 0, a_3 = 0]\}$
    * $A < B = false$
    * $A < B\ \&\ A(BOOT) = false$
    * $\exists A\ \exists B\ (A < B\ \wedge A(BOOT)) = false$
– For the last event $[end, 2]$:
  - $lookup(2)$ returns 001 since its assigned before. At *begin* event, the mapping between 2 and *LOAD* was stored.
  - $update(end, 001, 001)$
    * $\mathcal{P}[X] := false$
    * $\mathcal{P}[XX] := \{[x_1 = 0, x_2 = 0, x_3 = 0], [x_1 = 0, x_2 = 0, x_3 = 1]\}$
    * $\mathcal{P}[XXY] := false$

    - $\mathcal{P}[XXYY] := \{[x_1 = 0, x_2 = 0, x_3 = 0, y_1 = 0, y_2 = 0, y_3 = 1]\}$
    - $\mathcal{P}[XD] := \{[x_1 = 0, x_2 = 0, x_3 = 0, d_1 = 0, d_2 = 0, d_3 = 0], [x_1 = 0, x_2 = 0, x_3 = 1, d_1 = 0, d_2 = 0, d_3 = 1]\}$
- $eval(\varphi, \mathcal{P}, \mathcal{D})$
    - $A(BOOT) = \{[a_1 = 0, a_2 = 0, a_3 = 0]\}$
    - $A < B = \{[a_1 = 0, a_2 = 0, a_3 = 0, b_1 = 0, b_2 = 0, b_3 = 1]\}$
    - $A < B \;\&\; A(BOOT) = \{[a_1 = 0, a_2 = 0, a_3 = 0, b_1 = 0, b_2 = 0, b_3 = 1]\}$
    - $\exists A \exists B\, (A < B \;\&\; A(BOOT)) = true$

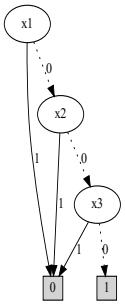
We see, that after the last event, the property is satisfied.



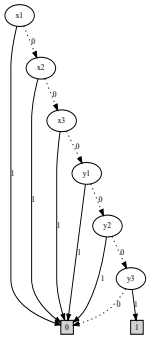
(a) $A(BOOT)$



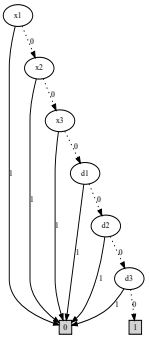
(b) $\mathcal{P}[X]$



(c) $\mathcal{P}[XX]$



(d) $\mathcal{P}[XXY]$



(e) $\mathcal{B}[XD]$

Fig. 4: BDDs after the third event [begin,2,LOAD].

Experiments. To evaluate MonAmi, we performed a comparison with the interval-based nfer tool [14], mentioned in the related work section on page 3. We formulated

four properties using the formalisms of the two systems, all related to receiving data from a planetary rover, and evaluated tool performances (time and memory) on traces of different sizes. The planetary rover scenario is inspired by realistic properties of the Curiosity Mars rover [17]. The rover's behavior is reported to ground via the following simplified intervals (amongst many): DL_IMAGE (downlink an image), DL_MOBPRM (downlink mobility parameter values), DL_ARMPRM (downlink robotic arm parameter values), DL_FAIL (downlink fails), INS_ON (instrument power turned on), INS_FAIL (instrument powering fails), INS_RECOVER (instrument recovers), GET_CAMDATA (reading camera data), STARVE (thread starves), and BOOT (re-boot rover, e.g. after a failure).

The four properties expressed in the formalisms of MonAmi and nfer are shown in Figure 5. In nfer we state a property as as a collection of Prolog-like interval-generating rules of the form id :– body, where the rule body contains Allen operators applied to events and intervals generated by other rules. The result of a match of the body is a new interval with the name id, as specified by the rule head. Events and intervals can carry data, which can be used e.g. in **where**-conditions. The IVAL rule (used by all the four properties) generates intervals for all matching (same interval identifier) BEGIN and END events in the trace, and stores (**map**) their interval and data values in the generated IVAL event. The FOUND interval in each nfer property is generated when an error is detected.

```
1. !exist B1, B2, D .
     B1('BOOT') & B2('BOOT') & D('DL_IMAGE') &
     B1 < B2 &
     (B1 i D |
      B2 i D |
      (B1 < D & D < B2) |
      (B1 o D & !D i B2) |
      (D o B2 & !D i B1)
     )

2. !exist D, F .
     (D('DL_MOBPRM') | D('DL_ARMPRM')) &
     F('DL_FAIL') &
     D i F

3. !exist O, F, R .
     O('INS_ON') & F('INS_FAIL') & R('INS_RECOVER') &
     O < F & F < R &
     !exist X . (X('INS_ON') | X('INS_RECOVER')) & O < X & X < R

4. !exist D, G, S .
     D('DL_IMAGE') & G('GET_CAMDATA') & S('STARVE') &
     D i S & G i S
```

```
IVAL :– BEGIN before END
  where BEGIN.interval = END.interval
  map {interval → BEGIN.interval, data → BEGIN.data}

1. BOOT :– IVAL where IVAL.data = "BOOT"
   DL:– IVAL where IVAL.data = "DL_IMAGE"
   DBOOT :– BOOT before BOOT
   FOUND :– DL during DBOOT

2. DL :– IVAL where IVAL.data = "DL_MOBPRM" | IVAL.data = "DL_ARMPRM"
   FAIL :– IVAL where IVAL.data = "DL_FAIL"
   FOUND :– FAIL during DL

3. ON :– IVAL where IVAL.data = "INS_ON"
   FAIL :– IVAL where IVAL.data = "INS_FAIL"
   RECOVER :– IVAL where IVAL.data = "INS_RECOVER"
   EXEC :– ON before RECOVER
   FOUND :– FAIL during EXEC

4. DL :– IVAL where IVAL.data = "DL_IMAGE"
   GET :– IVAL where IVAL.data = "GET_CAMDATA"
   STARVE :– IVAL where IVAL.data = "STARVE"
   FOUND :– STARVE during (GET slice DL)
```

Fig. 5: Evaluated propertiesin MonAmi (left) and nfer (right).

Property 1 states that there is no DL_IMAGE during two BOOT intervals (after the start of the first and before the end of the second), causing a potential loss of information. Property 2 states that there is no DL_FAIL during a DL_MOBPRM or DL_ARMPRM interval. Property 3 states that there is no INS_FAIL in between an INS_ON and a subsequent closest INS_RECOVER. Note how in the MonAmi speci-

fication we need to express the concept of *closest* as an additional constraint (that there is no INS_ON or INS_RECOVER in between). In nfer this is the default semantics, also referred to as the *minimality* principle, see discussion below. Property 4 states that there is no STARVE during a period where both an DL_IMAGE interval and a GET_CAMDATA interval are active. The nfer **slice** operator produces the intersection between two intervals. As mentioned, nfer's default execution mode uses a principle of *minimality*, where nfer's A **before** B operator (analog to MonAmi's A $<$ B operator) searches the closest right-most B from a given A. The minimality principle, however, can be switched off; so it behaves like MonAmi. Properties 1, 2, and 4 are in nfer evaluated with minimality switched off. nfer was originally designed to run with minimality switched on. However, the C version of nfer offers the option of switching off minimality, while the Scala version was extended with this option in order to perform the experiment.

We created 5 trace files for each property of different sizes, with 1000, 2000, 4000, 8000, and 16000 events. The traces were generated to evaluate the natural execution mode of MonAmi (stop on first violation) for these properties, by creating the traces to be violated only at the last event. These were generated with a trace generator, guided by one rule for each property. The maximal number of overlapping intervals was also controlled by a parameter (we chose as limit of 3). To ensure that violation will not occur in the middle of the trace we set the data to be different from the ones that appears in the property, except for the violating events. MonAmi is compared to two versions of nfer, a first prototype version in Scala [21], and a later developed version in C [20]. In addition, MonAmi is run in two modes: small step (*S*), the normal mode, where the formula is re-evaluated for each new event; and big step (*B*), where the formula is evaluated only at the end of the trace. The latter mode is a form of optimization, where we reduce the number of times the formula is evaluated. Table 2 shows the result of the evaluation. The experiments were carried out on a Dell Latitude 5401 laptop (Intel Core I7-9850H 9th Gen, 32GB RAM, 512GB SSD) with Ubuntu 20.04.2 LTS OS.

Wrt. memory, nfer/C overall performs the best and nfer/Scala the worst. MonAmi/CB and MonAmi/CS both perform very close to the good performance of nfer. Wrt. time, again nfer/C performs the best. MonAmi/CB performs as well as or close to nfer/C, whereas MonAmi/CS generally performs least well wrt. time. nfer/Scala is somewhere in the middle.

6 Conclusion

We described an extension to Alan's temporal logic, termed FoATL, that allows quantification over the intervals that occur in a monitored trace. We presented an efficient algorithm for runtime-verification and implemented a prototype tool in Python. The implementation is based on representing sets of tuples of enumerations over the intervals and their data values as BDDs using the 'dd' package.

The closest tool related to MonAmi is nfer and we comment on the relations between these two tools and their capabilities. The FoATL logic allows for a very convenient form of quantification over events, which resembles first-order quantification over time points on a time line. This approach becomes feasible due to the fact that we only "quantify" only over the time points where things actually happen in the trace, and due to the

Property	Tool	1000	2000	4000	8000	16000
1	MonAmi/*S*	1.89 s 51.86 MB	9.46 s 52.43 MB	22.00 s 54.19 MB	72.93 s 78.02 MB	250.55 s 90.50 MB
	MonAmi/*B*	0.31 s 51.74 MB	0.60 s 52.48 MB	1.25 s 54.56 MB	3.82 s 58.94 MB	6.82 s 86.47 MB
	nfer/*Scala*	0.19 s 140.41 MB	0.35 s 164.09 MB	1.28 s 395.83 MB	4.42 s 365.73 MB	17.32 s 385.23 MB
	nfer/*C*	0.03 s 11.03 MB	0.05 s 11.48 MB	0.15 s 12.70 MB	0.52 s 15.15 MB	1.96 s 19.85 MB
2	MonAmi/*S*	0.37 s 51.71 MB	0.83 s 52.65 MB	2.88 s 54.35 MB	7.98 s 57.30 MB	10.65 s 63.39 MB
	MonAmi/*B*	0.17 s 51.67 MB	0.30 s 52.27 MB	0.61 s 54.34 MB	1.20 s 57.06 MB	2.47 s 64.27 MB
	nfer/*Scala*	0.25 s 147.85 MB	0.41 s 196.26 MB	1.19 s 352.84 MB	4.32 s 392.45 MB	18.73 s 662.18 MB
	nfer/*C*	0.02 s 11.00 MB	0.04 s 11.48 MB	0.14 s 12.75 MB	0.52 s 15.12 MB	1.98 s 19.89 MB
3	MonAmi/*S*	1.20 s 51.69 MB	3.89 s 52.62 MB	13.06 s 54.30 MB	61.25 s 59.08 MB	385.18 s 86.24 MB
	MonAmi/*B*	0.19 s 51.82 MB	0.36 s 52.48 MB	0.82 s 54.35 MB	1.69 s 57.09 MB	3.58 s 66.90 MB
	nfer/*Scala*	0.24 s 142.16 MB	0.44 s 191.50 MB	1.29 s 332.99 MB	4.78 s 391.98 MB	19.82 s 562.61 MB
	nfer/*C*	0.02 s 11.05 MB	0.05 s 11.49 MB	0.15 s 12.77 MB	0.54 s 15.18 MB	2.12 s 19.91 MB
4	MonAmi/*S*	0.51 s 51.85 MB	1.49 s 52.55 MB	4.74 s 53.91 MB	17.31 s 57.21 MB	54.80 s 64.79 MB
	MonAmi/*B*	0.18 s 51.70 MB	0.32 s 52.25 MB	0.72 s 53.88 MB	1.30 s 57.09 MB	2.74 s 65.87 MB
	nfer/*Scala*	0.20 s 150.56 MB	0.39 s 199.01 MB	1.23 s 402.66 MB	4.86 s 361.00 MB	18.29 s 531.94 MB
	nfer/*C*	0.02 s 11.10 MB	0.05 s 11.63 MB	0.15 s 13.01 MB	0.54 s 15.67 MB	2.16 s 21.08 MB

Table 2: MonAmi's S and B modes versus nfer's Scala and C versions.

storage of data as BDDs. nfer, in contrast, has the flavor of rule-based programming. FoATL allows free negation, and consequently implication, which is only allowed in a limited sense in the C version of nfer, and not in the Scala version; the properties on page 7 show very natural uses of inner negation and implication. MonAmi can also be extended with new interval types. nfer relies as default on the minimal interpretation of the before-operator, choosing the closests rightmost interval. MonAmi can be extended to also to allow this mode, since it does provide performance improvements, and in some cases properties do rely naturally on minimality. Extending the logic to be first-order also wrt. data is considered for future work.

References

1. James F. Allen, Maintaining Knowledge About Temporal Intervals, Communications of the ACM, 26 (11), 832–843.
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, 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.
5. D. A. Basin, F. Klaedtke, S. Müller, E. Zalinescu, Monitoring Metric First-Order Temporal Properties, Journal of the ACM 62(2), 45, 2015.
6. R. E. Bryant, Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, ACM Comput. Surv. 24(3), 293-318, 1992.
7. CUDD BDD package [https://davidkebo.com/cudd]
8. The 'dd' Python package for manipulating Binary decision diagrams (BDDs) and Multi-valued decision diagrams (MDDs) [https://github.com/tulip-control/dd]
9. N. Decker, M. Leucker, D. Thoma, Monitoring Modulo Theories, Journal of Software Tools for Technology Transfer, Volume 18, Number 2, 2016.
10. S. Hallé, R. Villemaire, Runtime Enforcement of Web Service Message Contracts with Data, IEEE Transactions on Services Computing, Volume 5 Number 2, 2012.
11. J. Y. Halpern, Y. Shoham, A Propositional Modal Logic of Time Intervals, Journal of ACM 38(4), 935-962, 1991.
12. K. Havelund, D. Peled, D. Ulus, First-order Temporal Logic Monitoring with BDDs. FM-CAD 2017, 116-123
13. K. Havelund, G. Rosu, Synthesizing Monitors for Safety Properties, TACAS'02, LNCS Volume 2280, Springer, 342-356, 2002.
14. S. Kauffman, K. Havelund, R. Joshi, S. Fischmeister, Inferring Event Stream Abstractions. Formal Methods System Design 53(1): 54-82, 2018.
15. 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.
16. O. Kupferman, M. Y. Vardi. Model Checking of Safety Properties. Formal Methods System Design, 19(3), 291-314, 2001.
17. Mars Curiosity Rover [https://mars.nasa.gov/msl]
18. 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.
19. MonAmi tool source code [https://github.com/moraneus/MonAmI]
20. nfer in C [http://nfer.io]
21. nfer in Scala [https://github.com/rv-tools/nfer]. Awaiting JPL's permission for open sourcing.
22. 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.
23. G. Rosu, S. Bensalem, Allen Linear (Interval) Temporal Logic - Translation to LTL and Monitor Synthesis. CAV 2006: 263-277.
24. A. P. Sistla, Theoretical Issues in the Design and Analysis of Distributed Systems, Ph.D Thesis, Harvard University, 1983.
25. A. P. Sistla, M. Y. Vardi, P. Wolper, The Complementation Problem for Büchi Automata with Applications to Temporal Logic (Extended Abstract), ICALP 1985, 465-474, 1984.

26. L. J. Stockmeyer, A. R. Meyer: Word Problems Requiring Exponential Time: Preliminary Report, STOC, 1973: 1-9.
27. W. Thomas, Automata on Infinite Objects,. Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B) 1990: 133-191.