Runtime Verification of Traces under Recording Uncertainty*

Shaohui Wang, Anaheed Ayoub, Oleg Sokolsky, and Insup Lee

Department of Computer and Information Science
University of Pennsylvania
{shaohui,anaheed}@seas.upenn.edu,{sokolsky,lee}@cis.upenn.edu

Abstract. We present an on-line algorithm for the runtime checking of temporal properties, expressed as past-time Linear Temporal Logic (LTL) over the traces of observations recorded by a "black box"-like device. The recorder captures the observed values but not the precise time of their occurrences, and precise truth evaluation of a temporal logic formula cannot always be obtained. In order to handle this uncertainty, the checking algorithm is based on a three-valued semantics for past-time LTL defined in this paper. In addition to the algorithm, the paper presents results of an evaluation that aimed to study the effects of the recording uncertainty on different kinds of temporal logic properties.

1 Introduction

Data recorders are very important in the design of safety-critical systems. They allow system manufacturers and government regulators to collect data that help to diagnose the problem in case of a system failure. The best known example of a data recorder is the flight data recorder (FDR), also known as the "black box," that most aircraft are equipped with.

There is much interest in incorporating similar technology into medical devices. Adverse events—that is, cases where the patient was harmed during the application of the device—have to be reported to regulators. However, without data recording capability, analysis of adverse events becomes very difficult or even impossible. Thus, we are seeing the same kinds of adverse events repeated over and over again.

A preliminary design of a data recorder, called *life data recorder* (LDR) for medical devices has been proposed by Bill Spees, safety researcher at the U.S. Food and Drug Administration [18]. The LDR would collect updates of device state variables and relevant event occurrences and periodically transfer recorded snapshots to non-volatile storage. In doing so, the information about exact ordering of events within the recording period is lost. We can thus view a recorded trace as an abstraction of a concrete execution trace, so that the same abstract trace may arise from a number of concrete traces.

* Research is supported in part by the National Science Foundation grants CNS-0834524, CNS-0930647, and CNS-1035715.

In this paper, we are concerned with checking past-time LTL properties of system executions, that is, concrete traces. We assume, however, that all observations become available only after a snapshot is recorded. Thus we have only the abstract trace of the execution to work with. We therefore reinterpret LTL formulas in a way that reflects uncertainty in abstract traces. We introduce a three-valued semantics, under which a formula evaluates to true on an abstract trace Tr only if the same formula would evaluate to true on every concrete trace that is consistent with Tr. Dually, a formula is false on an abstract trace only if it is false on every consistent concrete trace. Otherwise, the outcome is uncertain.

We extend the algorithm of [10] to handle our three-valued semantics. The interesting aspect of the extension is that the algorithm operates on abstract traces; however, the formulas express properties of concrete traces, and there may be multiple concrete states between two abstract state. Thus, in each abstract state we need to reason about the segments of possible concrete traces since the previous abstract state, as well as refer to the truth values of subformulas calculated in the previous abstract state.

The paper is organized as follows. Section 2 defines abstract and concrete traces and describes the LDR recording scheme. Section 3 defines the three-valued semantics of past-time LTL over abstract traces and presents our runtime checking algorithm according to the semantics. Section 4 presents the evaluation of our checking algorithm on randomly generated traces. We conclude with an overview of related work in Section 5 and a discussion on possible future work in Section 6.

2 The Trace Model

In temporal logic based runtime verification, the primary task is to check a temporal logic formula on a given trace. A trace is usually regarded as a sequence of states, while the contents of states vary in different settings or domains. In this section, we describe the recording scheme of the LDR[18], and define two notions of traces, namely concrete traces and abstract traces.

2.1 LDR Recording Scheme

An LDR collects updates to a set of variables generated by a medical device and periodically records snapshots of their values in permanent memory. Three types of variables are recorded by the LDR: (a) process variables, (b) synchronized events, and (c) asynchronized events. The latter two together are called fast changers. At the time-out of every period, called a *frame*, a vector of 32-bit words is recorded to some non-volatile external storage. Recorded values are put to the vector slots according to a scheme specified by a *dictionary*, described as follows.

Process variables represent essential state information for the medical device, and are assumed not to change more than once during every frame. Each process variable is assigned one slot in the snapshot vector. It may be either empty, if the

value did not change during the frame, or contain the new value for the process variable.

Synchronized events are fast changers that may occur multiple times within a frame. They are recorded according to the time of their occurrences relative to the beginning of the current frame. One frame is divided into a fixed number (S, throughout the paper) of *subframe*s of equal intervals, for all synchronized events. We have $F = S \times I$, where F is the snapshot period (frame interval length), and I is the subframe interval length. Each synchronized event consumes S consecutive slots in the snapshot vector, starting from a designated base slot b. Assuming the beginning of the current frame is at time t, then an occurrence of a synchronized event at time t' ($t' < t + F$) is recorded to the slot numbered $b + \lfloor (t' - t)/I \rfloor$. Similar to process variables, we assume that there are no more than one occurrence per subframe for each synchronized event.

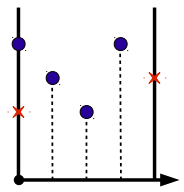
Asynchronized events are fast changers which exhibit bursty behavior: occasionally, they may change more than once per subframe, but the number of changes within a frame is bounded. Similar to synchronized events, a fixed number (A, throughout the paper) of consecutive slots are assigned to each asynchronized event. In one frame, at most A occurrences of an asynchronized event may happen. They are sequentially recorded one slot per occurrence, starting from the designated base slot b. No timing constraints with regards to subframes are imposed on asynchronized events. Only that they arrived in the order they are recorded in a frame is known.

Additional specifications of the LDR recording scheme in [18], such as encryption of data, external storages, etc., are tangential to our focus and omitted.

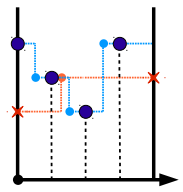
An example LDR recording. Fig. 1(a) shows an example segment of an LDR recording for a process variable x and a synchronized event y with at most four occurrences per frame ($S = 4$). The shaded cells in Column 0 represent the initial values for x and y. Each of the following columns is a snapshot vector for one frame in the recording session from the LDR. Frame 1 (shaded) is depicted in Fig. 1(b). The variable x (marked 'x') changes from 2 to 3, and y (marked 'o') changes from 4 to 3, to 2, and to 4, in the first, second, and third subframes, respectively. A dash entry in the snapshot vector means no events recorded.

	0	1	2	3
x	2	3	2	1
y^1		3	2	–
y^2		2	1	3
y^3		4	3	–
y^4	4	–	2	4

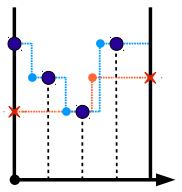
(a) A sample recording



(b) Depicting Frame 1 in (a)



(c) One event interleaving



(d) Another event interleaving

Fig. 1. Sample segment of an LDR recording.

Recorded traces may exhibit uncertainties in capturing system executions. Fig. 1(c) shows one possible system event interleaving which produces the shaded recording in Fig. 1(a). The smaller dots represent the actual events which alter the values of the relative variables. In this case, the change of x occurs in between the first and the second changes of y. Fig. 1(d) shows another, where the change of x occurs in between the second and the third changes of y. It can be seen that in this example there are four possible different system event interleavings which produce the shaded recording in Column 1 in Fig. 1(a).

2.2 Concrete Traces and Abstract Traces

In this paper, we differentiate two notions of traces, the concrete and the abstract. Informally, a concrete trace is a sequence of concrete states, where each of them is a mapping of variables to their values. An abstract trace is a sequence of abstract states, where each of them is, in our setting, an LDR recorded vector. We assume that concrete and abstract traces are finite.

Definition 1 (Concrete State and Concrete Trace). *Assuming a set V of variables and a domain D for their values, a concrete state is a mapping $f : V \to D$ of variables to their values. A concrete trace $p = p_0 \dots p_m$ of length m is a sequence of concrete states $p_0, \dots, p_m$.*

Definition 2 (Abstract State and Abstract Trace). *An abstract trace Tr is the sequence of snapshot vectors recorded by an LDR. Each snapshot vector is an abstract state.*

Concrete traces are not observed directly, but are captured by recordings from the LDR component, i.e., abstract traces. For example in Fig. 1, the snapshot vector for Frame 1 represents four concrete traces for (x, y) below, with the second and third depicted in Fig. 1(c) and Fig. 1(d), respectively:

$$(2,4) \xrightarrow{x} (3,4) \xrightarrow{y} (3,3) \xrightarrow{y} (3,2) \xrightarrow{y} (3,4),$$
$$(2,4) \xrightarrow{y} (2,3) \xrightarrow{x} (3,3) \xrightarrow{y} (3,2) \xrightarrow{y} (3,4),$$
$$(2,4) \xrightarrow{y} (2,3) \xrightarrow{y} (2,2) \xrightarrow{x} (3,2) \xrightarrow{y} (3,4),$$
$$(2,4) \xrightarrow{y} (2,3) \xrightarrow{y} (2,2) \xrightarrow{y} (2,4) \xrightarrow{x} (3,4).$$

We particularly note that an abstract state in our setting is essentially an acyclic transition system and captures a set of concrete traces. We say that any concrete trace that an abstract state captures is consistent with the abstract state.

As can be seen from the above example, the end states for each concrete trace in a frame are the same. This is due to the fact that at the end of each frame, all of the variables have been changed to their respective last values.

We use $Tr(0:n)$, or simply Tr, to represent the abstract trace of length n, $Tr(i)$ (a snapshot vector) to represent the i^{th} abstract state of Tr, and $Tr(i)_e$ to denote the concrete state at the end of the i^{th} abstract state ($1 \leq i \leq n$). $Tr(i)_e$ is computed from $Tr(i)$ by simply scanning through the vector $Tr(i)$ and establishing the mapping from each variable to its last value in the vector $Tr(i)$.

If nothing is recorded in $Tr(i)$ for a variable, its value from $Tr(i-1)_e$ is used. $Tr(0)$, which gives an initial value to every monitored variable, is a special case: it is, in effect, a concrete state, and fills exactly one slot in the vector for each variable.

We use $Path(Tr(i))$ to represent the set of concrete traces consistent with the abstract state $Tr(i)$, and the variable p^i to range over elements in $Path(Tr(i))$. When necessary, a concrete trace p^i of length m_i is written as $p^i = p^i_0 \dots p^i_{m_i}$. Note that for a given i, all concrete traces in $Path(Tr(i))$ are of the same length, which is equal to the number of variable changing events recorded in Frame i, so a single i is subscripted to m. The superscript i is often omitted when the context is clear. The following notations refer to the same concrete state for a given i: $Tr(i)_e$, $p^i_{m_i}$, and p^{i+1}_0.

Without loss of generality, we assume that all concrete traces for a given frame are not zero-length, since zero-length concrete traces result from abstract states where no changes to variable values occur, in which case the abstract state can be removed from our considerations.

For a span of n frames, the concrete traces are constructed by sequentially concatenating one concrete trace from each of the n frames. The concatenations at the boundaries of frames are consistent since the end values of variables in one frame are the same as their initial values in the next. We generalize the notation $Path(Tr(n))$ to $Path(Tr(0:n))$ to denote the set of concatenated concrete traces from abstract trace $Tr(0:n)$. We also generalize the concept of consistency between a concrete trace and an abstract trace naturally.

3 Syntax and Semantics of Past-time LTL

In real time systems, we often need to specify system properties with past-time LTL formulas and monitor system variables to check if the formulas are satisfied. The semantics for past-time LTL formulas on a concrete trace is standard [13, 15]. Also, to facilitate efficient runtime checking, it is convenient to define the semantics in a recursive fashion so that it is unnecessary to keep the history trace [10].

Checking past-time LTL properties on abstract traces, however, is different in that uncertainty arises when events are gathered in batch mode—a snapshot of a frame capturing a magnitude of events in the system—with their interleavings only partially known.

It is our main concern in this paper to both continue using the past-time LTL to describe system properties due to their succinctness and familiarity to the verification community, and handle the uncertainty in checking properties on the recorded abstract traces due to the unknown event interleavings.

Our approach is to keep the syntax for past-time LTL but introduce a new three-valued semantics based on standard semantics for concrete traces. A formula φ evaluates to true on an abstract trace Tr only if φ evaluates to true on all concrete traces consistent with Tr; φ evaluates to false on Tr only if it is false on every concrete trace consistent with Tr; otherwise it is undecided.

In this section, we first review the syntax of past-time LTL and its standard semantics and runtime checking algorithm, and then define the new semantics and extend the runtime checking algorithm to our three-valued semantics.

3.1 Syntax and Standard Semantics for Past-Time LTL

We assume all predicates on a set V of variables are the atomic formulas. We use the variable a to range over the set of atomic formulas, and $a(p_j)$ to represent the truth value of predicate a evaluated on concrete state p_j. The syntax rules for building formulas from atomic ones are as follows.

Definition 3 (Syntax for Formulas).

$$\varphi := \mathit{true} \mid \mathit{false} \mid a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \odot \varphi \mid \diamond\!\!\!\cdot\, \varphi \mid \boxdot \varphi \mid \varphi \,\mathcal{S}\, \varphi$$

For example, $(x = 14)\,\mathcal{S}\,(x \geq y)$ is a well formed formula. Intuitively, $\odot\,\phi$ reads "previously ϕ", meaning ϕ was true at the immediately previous state; $\diamond\!\!\!\cdot\, \phi$ reads "once ϕ", meaning there was some time in the past when ϕ was true; $\boxdot\,\phi$ reads "always in the past ϕ", meaning that ϕ was always true in the past; and $\phi\,\mathcal{S}\,\psi$ reads "ϕ (weakly) since ψ", meaning that either ϕ was always true in the past, or ψ held somewhere in the past and since then ϕ has always been true. The formal definition of the semantics is as follows.

Definition 4 (Standard Semantics for Past-Time LTL[10, 13, 15]). *A concrete trace $p = p_0 \ldots p_m$ of length m satisfies a past-time LTL formula φ, written $p \models \varphi$, is inductively defined on the structure of φ as follows.*

$$\begin{array}{lll}
p \models \mathit{true} & & \mathit{is\ always\ true} \\
p \models \mathit{false} & & \mathit{is\ always\ false} \\
p \models a & \mathit{iff} & a(p_m)\ \mathit{holds} \\
p \models \neg\psi & \mathit{iff} & p \not\models \psi \\
p \models \phi \wedge \psi & \mathit{iff} & p \models \phi\ \mathit{and}\ p \models \psi \\
p \models \phi \vee \psi & \mathit{iff} & p \models \phi\ \mathit{or}\ p \models \psi \\
p \models \odot\,\phi & \mathit{iff} & m > 0\ \mathit{and}\ p_0 \ldots p_{m-1} \models \phi,\ \mathit{or}\ m = 0\ \mathit{and}\ p_0 \models \phi \\
p \models \diamond\!\!\!\cdot\, \phi & \mathit{iff} & p_0 \ldots p_j \models \phi\ \mathit{for\ some}\ 0 \leq j \leq m \\
p \models \boxdot\,\phi & \mathit{iff} & p_0 \ldots p_j \models \phi\ \mathit{for\ all}\ 0 \leq j \leq m \\
p \models \phi\,\mathcal{S}\,\psi & \mathit{iff} & \mathit{either}\ p \models \boxdot\,\phi,\ \mathit{or}\ \big(p_0 \ldots p_j \models \psi\ \mathit{for\ some}\ 0 \leq j \leq m \\
 & & \qquad\qquad \mathit{and}\ p_0 \ldots p_k \models \phi\ \mathit{for\ all}\ j < k \leq m\big)
\end{array}$$

The Runtime Checking Algorithm The verification of a formula φ on a concrete trace p is based on the fact that the semantics in Definition 4 can be stated in a recursive fashion. For example, the semantics for the "since" operator $\mathcal{S}$ can be equivalently stated as

$$p \models \phi\,\mathcal{S}\,\psi \quad \text{iff} \quad p \models \psi, \text{ or } (p \models \phi \text{ and } (m > 0 \text{ implies } p_0 \ldots p_{m-1} \models \phi\,\mathcal{S}\,\psi))\,. \quad (1)$$

A runtime formula checker can cache the intermediate result of checking $\phi\,\mathcal{S}\,\psi$ on trace $p_0 \ldots p_{m-1}$ to use in the checking of $\phi\,\mathcal{S}\,\psi$ on trace p, according to the recursive semantics. In general, the checker iterates through all concrete states from p_0 through p_m. In each concrete state p_i, the checker keeps the satisfaction

results of all subformulas of φ on the trace $p_0 \dots p_{i-1}$ (which we call the *checker state*). The checker updates its state based on the values in p_i, as defined in [10].

We illustrate the algorithm with an example before we provide an extension in the next subsection, where we define three-valued semantics for past-time LTL. To check the truth value of $(x = 3)\,\mathcal{S}\,(x \geq y)$ in the trace for (x, y):

$$p = p_0 \dots p_4 = (2,5) \to (3,5) \to (3,3) \to (3,4) \to (3,6),$$

we follow the procedure of evaluating the subformulas $\phi \equiv (x = 3)$, $\psi \equiv (x \geq y)$, and $\phi\,\mathcal{S}\,\psi$.

Step	$\models$	$\phi \equiv (x = 3)$	$\psi \equiv (x \geq y)$	$\phi\,\mathcal{S}\,\psi$
0.	p_0	F	F	F
1.	$p_0 p_1$	T	F	F
2.	$p_0 p_1 p_2$	T	T	T
3.	$p_0 p_1 p_2 p_3$	T	F	T
4.	$p_0 p_1 p_2 p_3 p_4$	T	F	T

Each line in the table is the checker state for use in its next line. In deciding that $p_0 p_1 p_2 p_3 \models \phi\,\mathcal{S}\,\psi$ is true, for example, the facts that $p_0 p_1 p_2 p_3 \models \phi$ is true (from the current state) and that $p_0 p_1 p_2 \models \phi\,\mathcal{S}\,\psi$ is true (from the checker state) are used against the alternative semantics for the "since" operator $\mathcal{S}$ defined in (1).

3.2 Three-Valued Semantics for Past-Time LTL

Inspired by [12], we define a new semantics for the past-time LTL formulas against abstract traces. A formula φ is true on an abstract trace Tr only if φ evaluates to true on all concrete traces consistent with Tr; φ evaluates to false on Tr only if it is false on every concrete trace consistent with Tr; otherwise it is undecided. We use the semantic notions $[\![Tr \models \varphi]\!] = \top$, $[\![Tr \models \varphi]\!] = \bot$, and $[\![Tr \models \varphi]\!] = ?$ to indicate the three cases, respectively, where $\top$, $\bot$, and ? are truth values in three-valued logics to represent true, false, and unknown. The truth table for a commonly accepted variant of three-valued logics, namely the Kleene logic[11], is shown in Definition 5.

Definition 5 (Truth Table for Kleene Logic). *The following is the truth table for Kleene logic. (A and B are truth values.)*

A	$\top$			$\bot$			?		
B	$\top$	$\bot$	?	$\top$	$\bot$	?	$\top$	$\bot$	?
$A \vee_3 B$	$\top$	$\top$	$\top$	$\top$	$\bot$	?	$\top$	?	?
$A \wedge_3 B$	$\top$	$\bot$	?	$\bot$	$\bot$	$\bot$	?	$\bot$	?
$\neg_3 A$	$\bot$			$\top$			?		

We now consider the three-valued semantics for an abstract trace Tr of length n and a past-time LTL formula φ. We assume Tr and φ is fixed in the sequel.

We define the semantics in a recursive fashion, assuming the checking for the partial trace $Tr(0:i)$ is finished and the checking result of $[\![Tr(0:i) \models \psi]\!]$ for any (proper) subformula ψ of φ is available. We denote such information with a so called *subformula value mapping* $SV_i : \texttt{SubFormulas}(\varphi) \to \{\top, \bot, ?\}$ which, for a subformula ψ of φ, $SV_i(\psi) = [\![Tr(0:i) \models \psi]\!]$.

To establish the recursive semantic definition from $[\![Tr(0:i) \models \varphi]\!]$ to $[\![Tr(0:i+1) \models \varphi]\!]$, we use an auxiliary semantic function `checkOne` which takes a subformula value mapping, a concrete trace, and a formula, and returns a result from $\{\top, \bot, ?\}$. The intended use of function `checkOne` is that, when called with $\texttt{checkOne}(SV_i, p, \varphi)$, where SV_i is the subformula value mapping for φ on trace $Tr(0:i)$, and p is one concrete trace from $Path(Tr(i+1))$, the function returns whether φ is satisfied on all, none, or some (neither all nor none) concrete traces formed by concatenating any concrete trace in $Path(Tr(0:i))$ with p.

Definition 6. *Given a subformula value mapping SV, a formula φ, and a concrete trace $p = p_0 \dots p_m$, the function* $\texttt{checkOne}(SV, p, \varphi)$ *is defined inductively on the structure of φ, as follows.*

$$
\begin{aligned}
\texttt{checkOne}(SV, p, \mathit{true}) &= \top \\
\texttt{checkOne}(SV, p, \mathit{false}) &= \bot \\
\texttt{checkOne}(SV, p_0, \dots p_m, a) &= \begin{cases} \top, & \text{if } a(p_m) \text{ holds,} \\ \bot, & \text{if } a(p_m) \text{ does not hold,} \end{cases} \\
\texttt{checkOne}(SV, p, \neg\psi) &= \neg_3 \texttt{checkOne}(SV, p, \psi) \\
\texttt{checkOne}(SV, p, \phi \wedge \psi) &= \texttt{checkOne}(SV, p, \phi) \wedge_3 \texttt{checkOne}(SV, p, \psi) \\
\texttt{checkOne}(SV, p, \phi \vee \psi) &= \texttt{checkOne}(SV, p, \phi) \vee_3 \texttt{checkOne}(SV, p, \psi) \\
\texttt{checkOne}(SV, p_0 \dots p_m, \odot\, \phi) &= \begin{cases} SV(\odot\, \phi), & \text{if } m = 0, \\ SV(\phi), & \text{if } m = 1, \\ \texttt{checkOne}(SV, p_0 \dots p_{m-1}, \phi), & \text{if } m > 1. \end{cases} \\
\texttt{checkOne}(SV, p_0 \dots p_m, \diamond\, \phi) &= \begin{cases} \top, & \text{if } SV(\diamond\, \phi) = \top, \text{ or } \big(m > 0 \text{ and} \\ & \quad (\texttt{checkOne}(SV, p_0 \dots p_{m-1}, \diamond\, \phi) = \top \\ & \quad\quad \text{or } \texttt{checkOne}(SV, p_0 \dots p_m, \phi) = \top)\big), \\ \bot, & \text{if } SV(\diamond\, \phi) = \bot, \text{ and } \big(m > 0 \text{ implies} \\ & \quad (\texttt{checkOne}(SV, p_0 \dots p_{m-1}, \diamond\, \phi) = \bot \\ & \quad\quad \text{and } \texttt{checkOne}(SV, p_0 \dots p_m, \phi) = \bot)\big), \\ ?, & \text{otherwise.} \end{cases} \\
\texttt{checkOne}(SV, p_0 \dots p_m, \boxdot\, \phi) &= \begin{cases} \top, & \text{if } SV(\boxdot\, \phi) = \top, \text{ and } \big(m > 0 \text{ implies} \\ & \quad (\texttt{checkOne}(SV, p_0 \dots p_{m-1}, \boxdot\, \phi) = \top \\ & \quad\quad \text{and } \texttt{checkOne}(SV, p_0 \dots p_m, \phi) = \top)\big), \\ \bot, & \text{if } SV(\boxdot\, \phi) = \bot, \text{ or } \big(m > 0 \text{ and} \\ & \quad (\texttt{checkOne}(SV, p_0 \dots p_{m-1}, \boxdot\, \phi) = \bot \\ & \quad\quad \text{or } \texttt{checkOne}(SV, p_0 \dots p_m, \phi) = \bot)\big), \\ ?, & \text{otherwise.} \end{cases} \\
\texttt{checkOne}(SV, p_0 \dots p_m, \phi\, \mathcal{S}\, \psi) &= \begin{cases} SV(\phi\, \mathcal{S}\, \psi), & \text{if } m = 0, \\ \top, & \text{if } m > 0 \text{ and } \big(\texttt{checkOne}(SV, p_0 \dots p_m, \psi) = \top \\ & \quad \text{or } (\texttt{checkOne}(SV, p_0 \dots p_{m-1}, \phi\, \mathcal{S}\, \psi) = \top \\ & \quad\quad \text{and } \texttt{checkOne}(SV, p_0 \dots p_m, \phi) = \top)\big), \\ \bot, & \text{if } m > 0, \ \texttt{checkOne}(SV, p_0 \dots p_m, \psi) = \bot \\ & \quad \text{and } \big(\texttt{checkOne}(SV, p_0 \dots p_{m-1}, \phi\, \mathcal{S}\, \psi) = \bot \\ & \quad\quad \text{or } \texttt{checkOne}(SV, p_0 \dots p_m, \phi) = \bot\big), \\ ?, & \text{otherwise.} \end{cases}
\end{aligned}
$$

It is worthwhile to note that the `checkOne` function is recursive in terms of the length of the concrete trace p, and thus can be turned into an efficient algorithm using the idea from the runtime checking algorithm illustrated in Subsection 3.1.

Definition 7 (Three-Valued Semantics for Past-Time LTL). *An abstract trace Tr of length n satisfying a past-time LTL property φ, written $[\![Tr \models \varphi]\!]$,*

is inductively defined on the structure of φ, as follows.

$$
\begin{aligned}
[\![Tr \models \mathit{true}]\!] &= \top \\
[\![Tr \models \mathit{false}]\!] &= \bot \\
[Tr(0:n) \models a] &= \begin{cases} \top, & \text{if } a(Tr(n)_e) \text{ holds,} \\ \bot, & \text{if } a(Tr(n)_e) \text{ does not hold,} \end{cases} \\
[\![Tr \models \neg\psi]\!] &= \neg_3 [\![Tr \models \psi]\!] \\
[\![Tr \models \phi \wedge \psi]\!] &= [\![Tr \models \phi]\!] \wedge_3 [\![Tr \models \psi]\!] \\
[\![Tr \models \phi \vee \psi]\!] &= [\![Tr \models \phi]\!] \vee_3 [\![Tr \models \psi]\!]
\end{aligned}
$$

if φ is $\odot\,\phi$, $\diamondsuit\,\phi$, or $\boxdot\,\phi$,

$$
[\![Tr(0:n) \models \varphi]\!] = \begin{cases} \top, & \text{if } (n = 0 \text{ implies } Tr(0) \models \varphi) \text{ and } (n > 0 \text{ implies} \\ & \quad \forall p \in Path(Tr(n)) : \texttt{checkOne}(SV_{n-1}, p, \varphi) = \top), \\ \bot, & \text{if } (n = 0 \text{ implies } Tr(0) \not\models \varphi) \text{ and } (n > 0 \text{ implies} \\ & \quad \forall p \in Path(Tr(n)) : \texttt{checkOne}(SV_{n-1}, p, \varphi) = \bot), \\ ?, & \text{otherwise.} \end{cases}
$$

$$
[\![Tr(0:n) \models \phi\,\mathcal{S}\,\psi]\!] = \begin{cases} \top, & \text{if } (n = 0 \text{ implies } Tr(0) \models \phi \vee \psi) \text{ and } (n > 0 \text{ implies} \\ & \quad \forall p \in Path(Tr(n)) : \texttt{checkOne}(SV_{n-1}, p, \phi\,\mathcal{S}\,\psi) = \top), \\ \bot, & \text{if } (n = 0 \text{ implies } Tr(0) \not\models \phi\,\mathcal{S}\,\psi) \text{ and } (n > 0 \text{ implies} \\ & \quad \forall p \in Path(Tr(n)) : \texttt{checkOne}(SV_{n-1}, p, \phi\,\mathcal{S}\,\psi) = \bot), \\ ?, & \text{otherwise.} \end{cases}
$$

The definition is also recursive in the length n of the abstract trace $Tr(0:n)$, since the satisfaction results of subformulas of φ on trace $Tr(0:n-1)$ are encapsulated in the subformula value mapping SV_{n-1}. Note that the recursion stops at $n = 0$, where $Tr(0)$ is a concrete state and $SV_0(\psi)$ for a subformula ψ of φ is defined to be $\top$ if $\psi(Tr(0))$ holds, and $\bot$ otherwise.

3.3 An Example

In this section we provide an example illustrating the runtime checking algorithm which translates the recursive definitions of our three-valued semantics into an iterative procedure, and in the next section we present the algorithm.

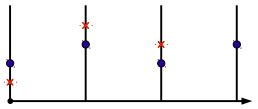
Consider the formula $\varphi \equiv \odot \odot \odot \diamondsuit\,(x = y)$ on the abstract trace Tr of length 3 shown in Fig. 2, where x and y are both process variables. The iterative steps are shown in Fig. 2(c), explained below.

Starting from the initial (concrete) state $(1, 2)$, all subformulas of φ are checked and the subformula value mapping SV_0 is updated. Then for Frame i $(i = 1, 2, 3)$, each box labeled $\#j$ $(j = 1, 2)$ is checked with a call to the auxiliary function $\texttt{checkOne}(SV_{i-1}, p^{\#j}, \varphi)$. Since `checkOne` is recursively defined on the length of the concrete trace $p^{\#j}$, inside each box labeled $\#j$, the entries are computed column by column. For example, the $(4, 2)$ column of box #1 in Frame 2 is the result of checking the initial segment $(4, 3) \to (4, 2)$ of concrete trace $p^{\#1} = (4, 3) \to (4, 2) \to (3, 2)$, which is an intermediate step in the call to $\texttt{checkOne}(SV_1, p^{\#1}, \varphi)$.

The values are computed according to Definition 6, except that recursive calls to $\texttt{checkOne}(SV, p_0 \dots p_{m-1}, \_)$ from $\texttt{checkOne}(SV, p_0 \dots p_m, \_)$, and the calls to $\texttt{checkOne}(SV, p_0 \dots p_m, \psi)$ from $\texttt{checkOne}(SV, p_0 \dots p_m, \varphi)$, where ψ is a subformula of φ, are replaced with table lookups.

	0	1	2	3
x	1	4	3	–
y	2	3	2	3

(a) Abstract Trace



(b) Depicting the Abstract Trace

	Initial State		Frame 1					Frame 2					Frame 3	
	(1, 2)	SV_0	$p^{\#1}$: →(1,3)→(4,3) $p^{\#2}$: →(4,2)→(4,3)				SV_1	$p^{\#1}$: →(4,2)→(3,2) $p^{\#2}$: →(3,3)→(3,2)				SV_2	$p^{\#1}$: →(3,3)	SV_3
			#1 (1,3)	(4,3)	#2 (4,2)	(4,3)		#1 (4,2)	(3,2)	#2 (3,3)	(3,2)		#1 (3,3)	
$x = y$	F	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊤	⊤
$\diamondsuit\,(x = y)$	F	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊤	?	⊤	⊤
$\odot\,\diamondsuit\,(x = y)$	F	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊤	?	?	?
$\odot\,\odot\,\diamondsuit\,(x = y)$	F	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	?	?
$\odot\,\odot\,\odot\diamondsuit\,(x = y)$	F	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥

(c) Runtime Checking Algorithm Steps

Fig. 2. Example for Runtime Checking Algorithm

After each box $\#j$ in Frame i is computed, for any subformula ψ of φ, $SV_i(\psi)$ is updated to $\top$ if all entries in the last columns of each box $\#j$ (shaded in Fig. 2(c)) and row ψ is $\top$; to $\bot$ if they are all $\bot$; and to ? otherwise. Checking for Frame $i+1$ begins after the update of SV_i. When the algorithm finishes, $SV_3(\varphi)$ is the result of checking $\varphi \equiv \odot\,\odot\,\odot\diamondsuit\,(x = y)$ on the abstract trace Tr.

This example shows the "hybrid" nature of the `checkOne` calculation: some subformulas of φ are evaluated on the concrete states of p, while others are looked up in the checker state of the preceding abstract state.

3.4 Checking Past-time LTL Formulas Against Abstract Traces

In this subsection, we present the algorithm, shown in Algorithm 1, for checking the truth value of a given past-time LTL formula φ and a given abstract trace Tr of length n, based on our recursive semantics in Definition 7. It is an extended version of the runtime checking algorithm based on the recursive definition for past-time LTL formulas on concrete traces [10].

We use the notation `SubFormulas`(φ) for the list of subformulas of φ, and assume the *enumeration invariant* in the algorithm: for any formula ψ at position j in `SubFormulas`(φ), all subformulas of ψ are at positions smaller than j.

4 Experiments

We implemented, in Python, a prototype of the past-time LTL checker described in the preceding sections. To evaluate our implementation and gain insights into the utility of the three-valued semantics, we also built a test environment that generates random abstract traces for a given LDR configuration file, and random past-time LTL formulas from a set of formula templates. Having generated sets of abstract traces and formulas, we evaluated each formula on every trace. This section summarizes the obtained results.

Algorithm 1: Runtime Checking for Past-time LTL on Abstract Traces

```
input  : abstract trace Tr(0 : n), past-time LTL formula φ
output: checking result for [[Tr |= φ]]
initialization: sf⟵SubFormulas(φ); Pre⟵ {} (empty mapping); Now⟵ {};
for j = 1 to length(sf) do
    if sf[j] is true then Pre[true]⟵ ⊤;
    if sf[j] is false then Pre[false]⟵ ⊥;
    if sf[j] is atomic formula a then
        if a(Tr(0)) holds then Pre[a] ⟵ ⊤ else Pre[a] ⟵ ⊥;
    if sf[j] is ¬ψ then Pre[¬ψ] ⟵ ¬3Pre[ψ];
    if sf[j] is φ ∨ ψ then Pre[φ ∨ ψ] ⟵Pre[φ]∨3Pre[φ];
    if sf[j] is φ ∧ ψ then Pre[φ ∧ ψ] ⟵Pre[φ]∧3Pre[φ];
    if sf[j] is ⊙φ, ⟐φ, or ⊡φ then Pre[sf[j]] ⟵Pre[φ];
    if sf[j] is φ S ψ then Pre[φ S ψ] ⟵Pre[ψ]∨3Pre[φ];
for i = 1 to n do
    for j = 1 to length(sf) do
        if sf[j] is true then Now[true]⟵ ⊤;
        if sf[j] is false then Now[false]⟵ ⊥;
        if sf[j] is atomic formula a then
            if a(Tr(i)_e) holds then Now[a] ⟵ ⊤ else Now[a] ⟵ ⊥;
        if sf[j] is ¬ψ then Now[¬ψ] ⟵ ¬3Now[ψ];
        if sf[j] is φ ∨ ψ then Now[φ ∨ ψ] ⟵Now[φ]∨3Now[φ];
        if sf[j] is φ ∧ ψ then Now[φ ∧ ψ] ⟵Now[φ]∧3Now[φ];
        if sf[j] is ⊙φ, ⟐φ, ⊡φ, or φ S ψ then
            forall p ∈ Path(Tr(i)) do
                check[p] ← checkOne(Pre, p, φ);
                result[p] ← check[p](sf[j]);
            if each element of result is ⊤ then Now[sf[j]] ⟵ ⊤;
            else if each element of result is ⊥ then Now[sf[j]] ⟵ ⊥;
            else Now[sf[j]] ⟵?;
    Pre⟵Now;
return Now[φ];
```

The formula templates were taken from common LTL specifications from the Spec Patterns project at Kansas State University [17]. For each of our five chosen categories, five temporal templates are specified: globally, before, after, between/and, and after/until. We altered the formulas for the twenty-five chosen templates from future-time to past-time, by replacing the future-time operators $\Box$ (globally), $\Diamond$ (eventually), $\bigcirc$ (next state), and $\mathcal{W}$ (weak until) with their past-time counterparts $\boxdot$, $\diamondsuit$, $\odot$, and $\mathcal{S}$, respectively. The strong until operator $\mathcal{U}$ was replaced with the strong since operator $\mathcal{S}_s$, and then transformed according to the equivalence $P\,\mathcal{S}_s\,Q \equiv (P\,\mathcal{S}\,Q) \wedge \diamondsuit\, Q$.

Table 1 lists all the specification templates used in our experiments. Note that the natural language description for each category has also been changed accordingly. For instance, the category "S precedes P" in future-time logic refers to traces where P cannot be true until an S happens ($(\neg P)\ \mathcal{W}\ S$). Its past-time counterpart ($(\neg P)\,\mathcal{S}\,S$) states that "S concluded P", i.e., if a P was observed, there must have later been an observation of S after which P was always false.

Twenty-five instances for each of the twenty-five formula templates were generated, with atomic symbols (P, Q, R, etc.) in the templates replaced with randomly generated atomic formulas, in our case predicates involving LDR recorded variables, e.g., $a + 42 \leq b$. Forty abstract traces all of length 20 were also ran-

		1. globally	2. after R	3. before Q
A.	absence (P was false)	$\boxdot \neg P$	$\diamondsuit R \rightarrow (\neg P \, \mathcal{S}_s \, R)$	$\boxdot (Q \rightarrow \boxdot \neg P)$
B.	existence (P became true)	$\diamondsuit P$	$(\neg R) \, \mathcal{S} \, (P \wedge \neg R)$	$\boxdot (\neg Q) \vee \diamondsuit (Q \wedge \diamondsuit P)$
C.	universality (P was true)	$\boxdot P$	$\diamondsuit R \rightarrow (P \, \mathcal{S}_s \, R)$	$\boxdot (Q \rightarrow \boxdot P)$
D.	conclusion (S concluded P)	$(\neg P) \, \mathcal{S} \, S$	$\diamondsuit R \rightarrow ((\neg P) \, \mathcal{S}_s \, (S \vee R))$	$\boxdot (\neg Q) \vee (Q \wedge ((\neg P) \, \mathcal{S} \, S))$
E.	cause (S weakly caused P)	$\boxdot (P \rightarrow \diamondsuit S)$	$\diamondsuit R \rightarrow ((P \rightarrow ((\neg R) \, \mathcal{S}_s \, (S \wedge \neg R))) \, \mathcal{S}_s \, R)$	$\boxdot (Q \rightarrow \boxdot (P \rightarrow \diamondsuit S))$

		4. between R and Q	5. before Q since R
A.	absence (P was false)	$\boxdot ((Q \wedge \neg R \wedge \diamondsuit R) \rightarrow ((\neg P) \, \mathcal{S}_s \, R))$	$\boxdot ((Q \wedge \neg R) \rightarrow ((\neg P) \, \mathcal{S} \, R))$
B.	existence (P became true)	$\boxdot ((Q \wedge R) \rightarrow ((\neg R) \, \mathcal{S} \, (P \wedge \neg R)))$	$\boxdot ((Q \wedge R) \rightarrow ((\neg R) \, \mathcal{S}_s \, (P \wedge \neg R)))$
C.	universality (P was true)	$\boxdot ((Q \wedge \neg R \wedge \diamondsuit R) \rightarrow (P \, \mathcal{S}_s \, R))$	$\boxdot ((Q \wedge \neg R) \rightarrow (P \, \mathcal{S} \, R))$
D.	conclusion (S concluded P)	$\boxdot ((Q \wedge \neg R \wedge \diamondsuit R) \rightarrow ((\neg P) \, \mathcal{S}_s \, (S \vee R)))$	$\boxdot ((Q \wedge \neg R) \rightarrow ((\neg P) \, \mathcal{S} \, (S \vee R)))$
E.	cause (S weakly causesd P)	$\boxdot ((Q \wedge \neg R \wedge \diamondsuit R) \rightarrow ((P \rightarrow ((\neg R) \, \mathcal{S}_s \, (S \wedge \neg R))) \, \mathcal{S}_s \, R))$	$\boxdot ((Q \wedge \neg R) \rightarrow ((P \rightarrow ((\neg R) \, \mathcal{S}_s \, (S \wedge \neg R))) \, \mathcal{S} \, R))$

Table 1. Past-time LTL Formula Templates

domly generated, according to the dictionary for a process variable a and two synchronous events b and c, with at most four recordings per frame ($S = 4$).

Therefore in our experiments, a total of 40(# traces) $\times$ 25(# templates) $\times$ 25(# instances / template) $= 25,000$ trace-formula combinations were tested. For each given abstract trace Tr and given formula instance φ, we collected the results for $[\![Tr(0) \models \varphi]\!]$, $[\![Tr(0:1) \models \varphi]\!]$, ..., $[\![Tr(0:20) \models \varphi]\!]$ as a sequence of 21 values from $\{\top, \bot, ?\}$, which we call a *result sequence* for Tr and φ. So a total of $25,000 \times 21$(length of a result sequence) $= 525,000$ values of $\top$, $\bot$, or ? were collected.

The experiments were run on a Windows XP desktop with 2.8GHz Intel Core Duo CPU and 2Gb memory and finished within 7 hours. Profiling shows that 97.7% of the running time was spent on executing the `checkOne` function, due to the exponential number of concrete traces corresponding to an abstract state. A few of our observations are discussed below.

Frequency of uncertain outcomes. We first evaluated how often the uncertain result (?) happens. Table 2 lists our two measurements: (a) how many of the trace-formula combinations give uncertain checking results (the number of result sequences whose the last value is ?), and (b) how many of all the 525,000 results are uncertain (the total number of ? in all result sequences).

We see from Table 2 that, the uncertain results do not occur as often as one may expect. To explain this observation, we note that most of the temporal operators are insensitive to the uncertainty, and also the scope of uncertainty is bounded within one abstract state.

Propagation of uncertainties. We then consider that, given a trace Tr and a formula φ, if an observation of an uncertain result happened at abstract state i, i.e., $[\![Tr(0:i) \models \varphi]\!] =?$, whether it will be the case that all following outcomes in the result sequence are uncertain, i.e, $[\![Tr(0:j) \models \varphi]\!] =?$, for all $i \leq j \leq n$.

We identified 7,129 out of all the 25,000 result sequences where ? occurred at least once somewhere in the sequence, 3,660 of which (51.34%) exhibit outcomes

Measurement	Uncertain Results	Total Cases	Percentage
(a)	3903	25,000	15.61%
(b)	63359	525,000	12.07%

Table 2. Chance of Uncertainty in Three-Valued Logic

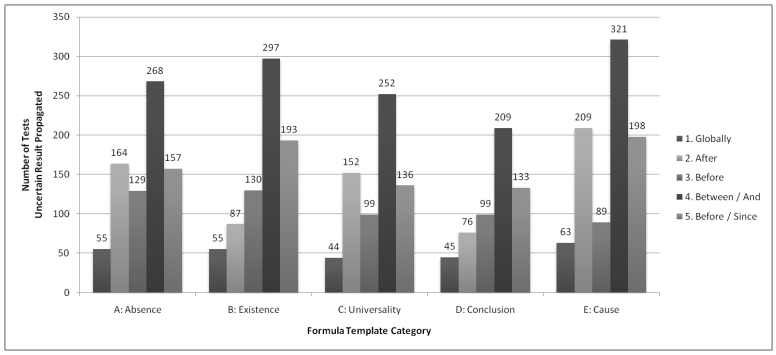


Fig. 3. Uncertainty Propagation

with trailing uncertain values (?) up to the end of the respective sequence. Above, we saw that uncertain values do not occur often. However, once occurred, they tend to persist. This is consistent with the intuition that an uncertain result in one abstract state pollutes the checker state and affects all subsequent states.

Fig. 3 plots which formula templates these 3,660 result sequences belong to. It is observed that templates in the "between/and" group are more likely to propagate uncertainty. This is partly due to the complex formula templates in the "between/and" group, which make the checker less likely to exit the uncertain state, compared to simpler templates in the "globally" group.

Impact of formula patterns. Another observation from the collected results is that, certain groups of formula templates exhibit patterned checking results.

The first group of formula templates includes $\{A.1, B.1, C.1\}$ in Table 1. The formula templates share the form that either $\diamond$ or $\boxdot$ quantifies over an atomic formula or its negation. The common patterns are either (a) all $\bot$ or all $\top$, or (b) a consecutive number of $\top$ (or $\bot$, respectively), followed by a consecutive number of ?, and then all $\bot$ (or $\top$, respectively).

This observation shows that, once a property with the $\boxdot$ operator has been falsified, it continues to be false; before this, it underwent being (probably trivially) true on *all* concrete traces, *some* concrete traces, and finally *none*. A dual result can be stated for the $\diamond$ operator.

A second group of templates involve the formulas where $\mathcal{S}$ is the main operator ($\{B.2, D.1\}$). The respective result sequences for formulas in this group show no obvious pattern, where the values $\top$, $\bot$, and ? almost randomly appear. This shows that randomly generated formulas with $\mathcal{S}$ as the main operator are more often determined locally in one abstract state.

5 Related Works

There are many runtime verification systems that formalize correctness properties in LTL, as seen for example in [12, 5, 8]. Different LTL variants have been defined based on semantics for finite traces [3, 4, 9]. The three-valued logic LTL_3 as an LTL logic with a semantics for finite traces has been used in [4]. The LDR traces in this paper are special cases of Mazurkiewicz traces[14] where the independence relation is defined by the LDR recording scheme. Alternative semantics of LTL formulas on Mazurkiewicz traces were studied (e.g., [**?**,7]) but were not based on a three-valued interpretation.

Compared to [4], this work used three-valued semantics for past-time LTL on traces with the LDR recording scheme, where the uncertainty in our case comes from unknown event interleavings; in [4], the uncertainty for LTL_3 was due to all possible unknown future suffixes of a finite trace. Although past-time LTL is not more expressive than LTL, it is exponentially more succinct and more convenient for specifying correctness properties for runtime verification over finite traces[9].

The technique of defining recursive semantics for checking temporal logic properties is standard to model checking [6] and runtime verification. [1, 10], as well as the algorithm presented in this paper, are based on this technique.

[16, 19] provide different approaches to randomly generating LTL formulas. We used templates from [17] in our experiments as the formula categorization helps study the relationship between satisfaction of formulas and their patterns.

6 Conclusion and Future Work

We considered a problem of runtime verification of past-time LTL properties over recorded traces, in which some information about the order of observations may be lost. We showed that a three-valued interpretation of the formulas is needed to reflect this uncertainty. We developed the appropriate semantics for past-time LTL and implemented the checking algorithm. Finally, we conducted an evaluation of checking several formula patterns over randomly generated traces and discussed the effects on uncertainty on checking outcomes.

We intend to extend this work in several directions. Extending the new semantics to the full LTL will require a non-trivial effort, and we also plan to tackle the effect of uncertainty on real-time properties. In the recorded traces, abstract states are timestamped when the state is recorded, but the time of actual observations is lost, resulting in additional uncertainty for the timed operators.

We believe that the implementation of the checker can be substantially improved by treating the set of concrete trace segments symbolically. A naive idea may be to simply run an LTL model checker on the transition system that represents the LDR trace model; however, we also need to construct the checker state in the previous abstract state for the right subformulas. Thus a more elaborate approach is needed.

Finally, we would like to consider a more precise semantic definition, so that a formula evaluates to $\top$ *if and only if* it is true on every concrete trace. The

current semantics satisfies just the "only if" condition. Indeed, suppose we are checking the formula $\phi \vee \psi$, and ϕ holds exactly on those traces where ψ does not hold. Both ϕ and ψ evaluate to ?, but $\phi \vee \psi$ should evaluate to $\top$. One way to achieve this is to forego the Kleene logic and define the semantics of a formula directly in terms of the set of concrete paths on which the formula holds. Then, we can assign the truth value to each formula depending on whether this set is empty, or is equal to the set of all traces.

Acknowledgement. We would like to thank Klaus Havelund and Grigore Roşu for their insightful input and making their tools in [10] available. We also thank the anonymous reviewers for their comments to improve the paper.

References

1. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Program monitoring with LTL in EAGLE. Parallel and Distributed Processing Symposium (2004)
2. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: FSTTCS'06. LNCS, vol. 4337. Springer-Verlag (Dec 2006)
3. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL Semantics for Runtime Verification. Journal of Logic and Computation (JLC) 20, 651–674 (June 2010)
4. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology (2011)
5. Bodden, E.: J-LO—A tool for runtime-checking temporal assertions. Diploma thesis, RWTH Aachen University (Nov 2005)
6. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
7. Genest, B., Kuske, D., Muscholl, A., Peled, D.: Snapshot verification. In: TACAS'05. pp. 510–525. No. 3440 in LNCS, Springer-Verlag (2005)
8. Havelund, K., Rosu, G.: Monitoring programs using rewriting. International Conference on Automated Software Engineering (2001)
9. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: TACAS'02. pp. 342–356. Springer-Verlag (2002)
10. Havelund, K., Rosu, G.: Efficient monitoring of safety properties. Int. J. Softw. Tools Technol. Transf. 6(2), 158–173 (Aug 2004)
11. Kleene, S.C.: Introduction to Metamathematics. D. Van Nostrand (1950)
12. Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. In: PDPTA'99. pp. 279–287 (1999)
13. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems: specification, vol. 1. Springer Verlag (1992)
14. Mazurkiewicz, A.: Concurrent program schemes and their interpretations. Tech. rep., DAIMI Rep. PB 78, Aarhus University (1977)
15. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. IEEE (1977)
16. Rozier, K., Vardi, M.: LTL satisfiability checking. In: 14th Workshop on Model Checking Software (SPIN '07). LNCS, vol. 4595. Springer-Verlag (2007)
17. SAnToS Lab, Kansas State University: Property pattern mappings for LTL, http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml
18. Spees, W.S.: Functional Requirement for LDR Component. Center for Devices and Radiological Health, FDA (2010)
19. Tauriainen, H., Heljanko, K.: Testing LTL formula translation into Büchi automata. STTT 4(1), 57–70 (2002)