Online Monitoring of Distributed Systems with a Five-valued LTL

Ming Chai and Bernd-Holger Schlingloff
Humboldt University of Berlin
Rudower Chaussee 25, 12489 Berlin, Germany
Email: {ming.chai, hs}@informatik.hu-berlin.de

***Abstract*—In this paper, we deal with two kinds of uncertainties in distributed systems. On one hand, the order of causally unrelated executions is not determined because we cannot assume that there exists a global clock. On the other hand, in a finite amount of time, the behaviour can be observed only up to a certain moment, and the future behaviour is unknown. In this paper, we investigate a monitoring approach based on linear temporal logic (LTL) specifications. We propose a five-valued semantics for LTL to deal with both kinds of uncertainties. We develop an efficient runtime verification algorithm using formula rewriting, and show the feasibility of our approach with a case study in the railway domain.**

***Index Terms*—runtime verification; distributed system; five-valued logic; LTL**

I. Introduction

Runtime verification is a lightweight validation technique that checks whether a run of a system satisfies a correctness property or not. Online monitoring is implemented while the system is running by a *monitor*. This is a device or a piece of software that observes a behaviour of the system under monitoring, and gives a verdict as the result. Runtime verification does not require a model of the system, and is able to perform ongoing testing [1]. It is effective for providing correctness assurance for distributed systems, which are notoriously hard to verify or test. When monitoring a distributed system, the following problems arise.

A behaviour observed by a monitor is an execution trace, which is built according to time stamps of executions. For a distributed system, the order of causally unrelated executions of different components is not always determined when the system does not have a global clock. This can cause race conditions, which are difficult to catch and eliminate by testing or model checking. A race condition impacts monitoring results as well. It may result in inconsistency between an observed trace and the actual behaviour of a distributed system. Therefore, the monitoring results are not always certain. For instance, given a distributed system consisting of two components, each of which concurrently writes a data to a file. A monitor reads, e.g., a trace ($write_1$, $write_2$). However, due to asynchronicity, the actual behaviour of this system could be ($write_2$, $write_1$). Since we don't know which component writes the file first, the monitoring result for the property "$write_1$ is executed before $write_2$" is uncertain.

This work was supported by the State Key Laboratory of Rail Traffic Control and Safety (Contract No.: RCS2012K001), Beijing Jiaotong University.

Linear temporal logic (LTL) is a well accepted and expressive formal language used for specifying correctness properties. Unfortunately, monitoring results can be misleading with LTL. This is because LTL is usually defined on infinite traces, whereas a behaviour can be observed by a monitor only up to a certain moment. For instance, let τ = (*open*, *read*, *write*, *write*) be an observed trace. A correctness property "if a file is open, it will be closed in the future" is expressed by the LTL formula ($open \rightarrow \mathbf{F}\ close$). The monitoring result with standard LTL is false. This is an inappropriate verdict because there exists a suffix *close* leading to a trace which satisfies the formula. Since we do not know whether *close* will be actually executed, the monitoring result is not adequately expressed by a standard truth value (true or false) at this point in time.

One solution for the first problem could be to restrict properties for race conditions. If a behaviour is a critical race, a monitor for such correctness properties would only accept the correct execution order; otherwise, it would accept all possible execution orders. Unfortunately, such restrictions will make it difficult to build a monitor. In this paper, we use standard LTL to express correctness properties, and faithfully present satisfaction relations in monitoring results. The second problem is essentially caused by unknown further behaviours. A solution would be to restrict the evaluation to completed traces, e.g. postmortem dumps. However, in many contexts intermediate results are desirable. Thus, we introduce additional truth values for such intermediate results.

For the satisfaction relation between a behaviour of a distributed system $\mathfrak{b}$ and an LTL property φ, we consider the following five possibilities.

- $\mathfrak{b}$ satisfies or violates φ, no matter how the system will behave in the future;
- $\mathfrak{b}$ satisfies or violates φ, but there exists an extension $\mathfrak{b}'$ of $\mathfrak{b}$ which may change the verdict;
- the satisfaction relation between $\mathfrak{b}$ and φ can not be determined according to any of the above categories, since the actual order of independent events in $\mathfrak{b}$ is unknown.

We define a five-valued logic $E_5 = \{tt, ff, pt, pf, uk\}$ to express these five possible monitoring results. Truth values *tt* (*true*) and *ff* (*false*) indicate that the monitor has observed the system sufficiently, and can stop monitoring because the result will not change in the future. The values *pt* (*possibly true*) and *pf* (*possibly false*) indicate that the monitoring result may

change if further events are observed, therefore the monitoring has to continue. Truth value *uk* indicates that the monitoring result is unknown since the execution order is non-determined. In this case, the system could keep on running or terminate with an exception (if the behaviour causes a critical race); or the monitor could raise an alarm and start a further checking procedure.

Related work.

There are several approaches that express uncertainties in executions of systems. Bauer et. al. propose LTL_3 to express uncertainties caused by unknown future behaviours [2]. LTL_3 is defined by introducing a third truth value "?" into LTL. The truth value "?" indicates that it cannot be determined whether an LTL formula satisfies a trace because there are different possible suffixes of the trace. Furthermore, they extend their work by extending ? to $\top^p$ and $\bot^p$. The value $\top^p$ (or $\bot^p$) means that the satisfaction relation is unknown, meanwhile the existing trace does not violate (or satisfy, respectively) the property [3]. Morgenstern et al. [4] argue that even the four-valued semantics is not sufficient for defining the satisfaction relation between a LTL formula and an infinite trace. They therefore consider several subclasses of LTL formulae to express safety, liveness, persistence and recurrence properties, respectively, and propose semantics for each of these subclasses.

To express the occurrence probability of an event, fuzzy temporal logic has been proposed [5]. Lamine et. al. [6] and Pasquale et. al. [7] use this logic to monitor systems in unpredictable environments. To our knowledge, uncertainties caused by non-determined execution orders have not been studied in the runtime verification literature. However, uncertainties have been investigated in system testing. Kahsai et. al [8] use three-valued test oracles, which are colors green, red and yellow, to evaluate a test case with respect to a specification. A test case is evaluated to green, if it reflects a required behaviour of the specification; it is evaluated to red, if it reflects a forbidden behaviour of the specification; and it is evaluated to yellow, if it reflect a behaviour which is neither required nor forbidden by the specification.

The rest of the paper is organized as follows. Section II gives the definition of five-valued LTL for finite traces with inaccurate time values. Section III presents our runtime verification algorithm for monitoring distributed systems. Section IV shows a case study of the RBC/RBC handover process in the railway domain. Section V contains a conclusion and ideas for future work.

II. A Five-valued LTL for Monitoring distributed Systems

A. Truth Values

For a many valued logic, a truth value can be defined as a set of possible answers for a query from some assumptions [9]. In a three-valued logic $E_3 = \{Y, N, \{Y, N\}\}$, the truth value Y and N are singletons, indicating that only this answer can be concluded from the given assumptions; and $\{Y, N\}$ indicating that both answers are possible.

In three valued logic, the classical boolean operations need to be extended, and new connections can be defined. For disjunction ($\vee$) and negation ($\neg$), we adopt Kleene's three valued truth tables. In addition, we introduce a new binary operation ($\amalg$), which compares two arguments. In the following truth tables, the truth value $\{Y, N\}$ is denoted with ?.

$\vee$	Y	?	N
Y	Y	Y	Y
?	Y	?	?
N	Y	?	N

$\neg$	
Y	N
?	?
N	Y

$\amalg$	Y	?	N
Y	Y	?	?
?	?	?	?
N	?	?	N

When monitoring a system, the following two items are interesting:

1) whether the observed behaviour meets the correctness property, and
2) whether the monitoring result is conclusive or provisional.

Unfortunately, traditional many-valued logics give little knowledge about these items. Therefore, we define a truth value by an evolution operation $\rightsquigarrow$.

Given any truth values c, $\mathsf{c}' \in E_3$, we define a new truth value as $(\mathsf{c} \rightsquigarrow \mathsf{c}')$. The truth value indicates that the set of answers from given assumptions is c (i.e., the truth value at present is c), and the union of all possible answers when introducing more assumptions is c' (i.e., the possible truth value in the future is c').

Notice we cannot assume there always exists a new assumption to be introduced. The answer at present would still be a possible answer in the future. Therefore, for any $\mathsf{c} \rightsquigarrow \mathsf{c}'$ it holds that $\mathsf{c} \subseteq \mathsf{c}'$.

Formally, nine truth values can be defined by $\rightsquigarrow$ from E_3, however four of them (Y $\rightsquigarrow$ N, N $\rightsquigarrow$ Y, ? $\rightsquigarrow$ Y and ? $\rightsquigarrow$ N) violate the above rule. A five-valued logic $E_5 = \{tt, ff, pt, pf, uk\}$ can be obtained from E_3 with $\rightsquigarrow$ as follows.

- $tt \triangleq$ Y $\rightsquigarrow$ Y
- $ff \triangleq$ N $\rightsquigarrow$ N
- $uk \triangleq$? $\rightsquigarrow$?
- $pt \triangleq$ Y $\rightsquigarrow$?
- $pf \triangleq$ N $\rightsquigarrow$?

Given e_1, $\mathsf{e}_2 \in E_5$ with $\mathsf{e}_1 = \mathsf{c}_1 \rightsquigarrow \mathsf{c}_1'$ and $\mathsf{e}_2 = \mathsf{c}_2 \rightsquigarrow \mathsf{c}_2'$, we define $\mathsf{e}_1 \circ \mathsf{e}_2 \triangleq (\mathsf{c}_1 \circ \mathsf{c}_2) \rightsquigarrow (\mathsf{c}_1' \circ \mathsf{c}'_2)$ with $\circ \in \{\vee, \amalg\}$, and $\neg\ \mathsf{e}_1 \triangleq (\neg \mathsf{c}_1) \rightsquigarrow (\neg \mathsf{c}_1')$. Form this, the five-valued truth tables for the operations $\vee$, $\neg$ and $\amalg$ can be calculated, and are shown in Fig. 1.

B. The Five-valued LTL

Let AP be a finite set of atomic propositions and $\Sigma = 2^{AP}$ a finite alphabet. An event is any element of Σ. We define a finite trace over Σ to be an element of Σ^*, whereas an infinite trace is an element of Σ^ω.

Definition 1. (**LTL syntax**) Given the finite set AP of atomic propositions, LTL formulae are formed according to the following grammar, where $p \in AP$

$\varphi ::= \bot \mid p \mid \neg\ \varphi \mid (\varphi_1 \vee \varphi_2) \mid (\varphi_1\ \mathbf{U}\ \varphi_2) \mid \mathbf{X}\ \varphi.$

$\vee$	*tt*	*pt*	*uk*	*pf*	*ff*
tt	*tt*	*tt*	*tt*	*tt*	*tt*
pt	*tt*	*pt*	*pt*	*pt*	*pt*
uk	*tt*	*pt*	*uk*	*uk*	*uk*
pf	*tt*	*pt*	*uk*	*pf*	*pf*
ff	*tt*	*pt*	*uk*	*pf*	*ff*

$\amalg$	*tt*	*pt*	*uk*	*pf*	*ff*
tt	*tt*	*pt*	*uk*	*uk*	*uk*
pt	*pt*	*pt*	*uk*	*uk*	*uk*
uk	*uk*	*uk*	*uk*	*uk*	*uk*
pf	*uk*	*uk*	*uk*	*pf*	*pf*
ff	*uk*	*uk*	*uk*	*pf*	*ff*

$\neg$	
tt	*ff*
pt	*pf*
uk	*uk*
pf	*pt*
ff	*tt*

Figure 1. Truth tables for five-valued logic

In addition, we use the following shorthands: $\varphi_1 \wedge \varphi_2$ standards for $\neg\ (\neg\varphi_1 \vee \neg\ \varphi_2)$, $\varphi_1 \rightarrow \varphi_2$ stands for $\neg\ \varphi_1 \vee \varphi_2$, $\mathbf{F}\ \varphi$ stands for $\top\ \mathbf{U}\ \varphi$ and $\mathbf{G}\ \varphi$ stands for $\neg\mathbf{F}\ \neg\ \varphi$. The semantics for LTL on an infinite trace is as follows.

Definition 2. (**Semantics for standard (two valued) LTL**) Given an infinite trace $\tau = e_0 e_1 \cdots \in \Sigma^\omega$ with $i \geq 0$ being a position of τ. The semantics for LTL is defined inductively as follows:

$(\tau,\ i) \not\models \bot$;
$(\tau,\ i) \models p$ iff $p \in e_i$;
$(\tau,\ i) \models \neg\ \varphi$ iff $(\tau,\ i) \not\models \varphi$;
$(\tau,\ i) \models (\varphi_1 \vee \varphi_2\ )$ iff $(\tau,\ i) \models \varphi_1$ or $(\tau,\ i) \models \varphi_2$;
$(\tau,\ i) \models (\varphi_1\ \mathbf{U}\ \varphi_2)$ iff there exists $k \geq i$ such that $(\tau,\ k) \models \varphi_2$, and for all $i \leq j < k$ it holds that $(\tau,\ j) \models \varphi_1$;
$(\tau,\ i) \models \mathbf{X}\ \varphi$ iff $(\tau,\ i + 1) \models \varphi$.

We denote $(\tau,\ 0) \models \varphi$ with $\tau \models \varphi$. When monitoring a distributed system, an observed execution may actually be executed at a different global time (according to the observer) then a local time recorded in the time stamp. We assume that the global time and the local time deviate only by a fixed amount of time. An execution of a distributed system is defined with an *uncertain time event* (*ue*), which is a tuple $ue \triangleq (e,\ t,\ \Delta t)$ from the set $(\Sigma \times \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0})$. Here, t is a discrete time stamp to identify the local time of execution; and Δt standards for the maximal delay to the global time of *ue*. With the respect to the global clock, an event *ue* can be actually executed at any time within the interval $[t,\ (t\ +\ \Delta t)]$, which is denoted by $\mathcal{I}\ (ue)$. We define $Event\ (ue) \triangleq e$, $Time\ (ue) \triangleq t$ and *Delay* $(ue) \triangleq \Delta t$.

The temporal relations of two executions are uncertain if the intersection of their time intervals is not empty. An observed behaviour is a set of uncertain time events. From an observed behaviour, a set of event traces can be concluded. Given an observed behaviour $\mathfrak{b} \triangleq \{ue_1,\ ue_2,\ ...,\ ue_n\}$, let ρ be a permutation function on 1 to n. A trace τ is consistent with this behaviour if

- $\tau \triangleq Event(ue_{\rho(1)}) \cdots Event(ue_{\rho(n)})$, and
- for any i and j with $\rho(i) < \rho(j)$, either
  - $Time\ (ue_i) < Time\ (ue_j)$, or
  - $\mathcal{I}\ (ue_i) \cap \mathcal{I}\ (ue_j) \neq \emptyset$.

The concluded trace set ($\mathcal{T}_\mathfrak{b}$) for the behaviour $\mathfrak{b}$ is the set of all traces that are consistent with $\mathfrak{b}$. For instance, let $\mathfrak{b} = \{ue_a,\ ue_b,\ ue_c\}$ be an observed behaviour, where ue_a = (a, 0, 3), ue_b = (b, 2, 3) and ue_c = (c, 4, 3). Since *Time*(ue_c) > *Time*(ue_a), and the intersection of $\mathcal{I}(ue_c)$ and $\mathcal{I}(ue_a)$ is an empty set, *Event*(ue_c) comes after *Event*(ue_a) for all $\tau \in \mathcal{T}_\mathfrak{b}$. Because $\mathcal{I}(ue_a) \cap \mathcal{I}(ue_b) \neq \emptyset$, there exists $\tau,\ \tau' \in \mathcal{T}_\mathfrak{b}$ such that *Event*(ue_a) comes before *Event*(ue_b) in τ, meanwhile *Event*(ue_a) comes after *Event*(ue_b) in τ'. The event trace set $\mathcal{T}_\mathfrak{b}$ therefore is $\{abc,\ bac,\ acb\}$.

Whether a behaviour $\mathfrak{b}$ meets an LTL property φ is determined by the satisfaction relation between the trace set $\mathcal{T}_\mathfrak{b}$ and φ.

Definition 3. (**Semantics of five valued LTL on a finite trace set**) Let $\mathcal{T} \triangleq \{\tau_1,\ \tau_2,\ ...,\ \tau_n\}$ be a set of finite traces with $\tau_i \in \Sigma^*$ for all $1 \leq i \leq n$, and $\varepsilon \triangleq (\emptyset,\ \emptyset,\ ...) \in \Sigma^\omega$ an infinite empty trace. The truth value of the satisfaction relation between τ and a LTL formula φ, denoted with $[\tau \models \varphi]$, is defined as follows:

$$[\tau \models \varphi] = \begin{cases} tt & \text{if for all } \upsilon \in \Sigma^\omega\text{: } \tau\upsilon \models \varphi \\ pt & \text{if } \tau\varepsilon \models \varphi \text{ and there exists } \upsilon \in \Sigma^\omega\text{: } \tau\upsilon \not\models \varphi \\ pf & \text{if } \tau\varepsilon \not\models \varphi \text{ and there exists } \upsilon \in \Sigma^\omega\text{: } \tau\upsilon \models \varphi \\ ff & \text{if for all } \upsilon \in \Sigma^\omega\text{: } \tau\upsilon \not\models \varphi \end{cases}.$$

Here we only use four truth values because we define the satisfaction with respect to a single trace. The satisfaction relation between a trace set $\mathcal{T}$ and φ, denoted with $[\mathcal{T} \models \varphi]$, is then defined as

$[\mathcal{T} \models \varphi] = [\tau_1 \models \varphi]\ \amalg\ [\tau_2 \models \varphi]\ \amalg \cdots \amalg\ [\tau_n \models \varphi]$.

For the above example $\mathcal{T}_\mathfrak{b}$ = {*abc*, *bac*, *acb*}, let a = {p, q}, b = {p}, and c = {q} with $p,\ q \in AP$. Then we have $[\mathcal{T}_\mathfrak{b} \models (p \rightarrow \mathbf{F}\ q)]$ = tt, $[\mathcal{T}_\mathfrak{b} \models (p \rightarrow \mathbf{G}\ q)]$ = ff, $[\mathcal{T}_\mathfrak{b} \models \mathbf{G}\ \mathbf{F}\ (p \vee q)]$ = pt, $[\mathcal{T}_\mathfrak{b} \models \mathbf{G}\ (p \rightarrow \mathbf{F}\ q)]$ = pf, and $[\mathcal{T}_\mathfrak{b} \models ((p \wedge q)\ \mathbf{U}\ (p \wedge \neg\ q))]$ = uk.

III. A Forward Monitoring Algorithm

In runtime verification, an execution is observed by a monitor directly upon its occurrence. This may cause the following problems when monitoring a distributed system. On one hand, to find whether there exists uncertainties caused by non-determined execution orders, a set of traces concluded from the observed behaviour needs to be checked. All traces in the set have the same executions but different temporal relations. The size of the set grows heavily with the growing length of the observed behaviour. This increases the monitoring complexity seriously. On the other hand, future executions may change the temporal relations of existing ones. A monitor needs to re-check the existing trace if such an execution occurs. Therefore, the whole observed behaviour needs to be stored. This limits the implementation of runtime verification, especially online monitoring, when the storage space is limited. To overcome these drawbacks, we extend the traditional runtime verification approach by adding a buffer. The buffer collects executions from the system, and sends them to a monitor when future executions will not overlap with any of the collected ones. In this approach, a monitor reads a sequence of execution sets, where temporal relations of the sets are determined. It does

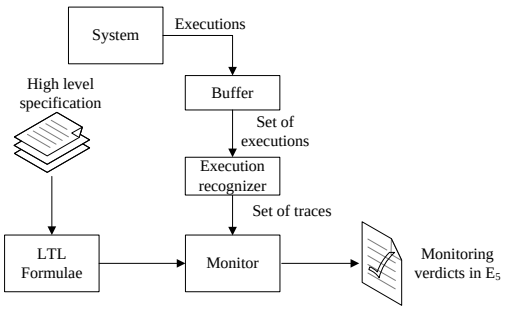

Figure 2. Framework for Monitoring a system

not re-check the existing trace when receiving a new execution set, and does not store the entire observed trace as well.

The framework of our runtime verification approach for distributed systems is shown in Fig. 2. In this framework, correctness properties are from the high level specification, and are expressed with LTL formulae. The buffer collects low level executions from the system. It has a timer, which resets to 0 when a new execution is observed. If the timer equals to the maximal time delay, then the buffer sends the set of collected executions to the execution recognizer. The execution recognizer converts received executions into a set of high-level event traces, which can be recognized by correctness properties. The high-level event traces are sent to the monitor, which consists of LTL formulae and five-valued LTL checking algorithms. The monitor presents satisfaction verdicts as monitoring results.

Let $\mathfrak{b} \triangleq \mathfrak{b}_1\mathfrak{b}_2 \ldots \mathfrak{b}_n$ be a finite behaviour sequence collected by the buffer. The corresponding trace set sequence $\mathcal{T}_\mathfrak{b} \triangleq \mathcal{T}_1\mathcal{T}_2 \ldots \mathcal{T}_n$ is generated by the execution recognizer, where for all $1 \leq i \leq n-1$ it holds that $\max_{ue \in \mathcal{T}_i}(Time\ (ue) + Delay\ (ue)) < \min_{ue \in \mathcal{T}_{i+1}}(Time\ (ue))$. A monitor checks the satisfaction relation between $\mathcal{T}$ and an LTL property. According Definition 3, the checking time for $(\mathcal{T}_\mathfrak{b} \models \varphi)$ is $|\mathcal{T}_1| \cdot |\mathcal{T}_2| \cdot \cdots \cdot |\mathcal{T}_n| \cdot \mathcal{O}(\tau_1\tau_2 \ldots \tau_n, \varphi)$, where $\mathcal{O}(\tau_1\tau_2 \ldots \tau_n, \varphi)$ is the time of checking $(\tau_1\tau_2 \ldots \tau_n \models \varphi)$ with $\tau_i \in \mathcal{T}_i$ for all $1 \leq i \leq n$. When a new behaviour $\mathcal{T}'$ is received, the definition requests to check the entire trace set sequence $\mathcal{T}\mathcal{T}'$. This causes backtracking in the checking process.

To reduce the complexity of monitoring, we develop a forward rewriting algorithm (Alg. 1). In this algorithm, subformulae of φ are stored into a list *FList* from the bottom up, and the formulae after rewriting are stored into a list *RewF*. A formula after rewriting consists of satisfaction relations, which are connected with operations "*and*", "*or*" and "*not*". the operation "$[\chi]$" assigns a formula χ to a truth value in E_5, where satisfaction relations are assigned to a truth value in E_5 and connected operations are assigned to $\wedge$, $\vee$ and $\neg$ respectively. The function *Rewrite*$(\mathcal{T}_i, \psi)$ assigns a formula $\mathcal{T}_i \models \psi$ to a truth value $\coprod_{\tau \in \mathcal{T}_i} [\tau \models \psi]$. The truth values for subformulae against a trace set are stored into a list *Eva*. The present result for $[\mathcal{T}_i \models \varphi]$ equals to Eva$[|FList\ (\varphi)|]$. Here we omit the rewriting rules for $\tau \models \varphi$, which can be developed directly from the definition of LTL.

Algorithm 1 A rewriting algorithm of $[\mathcal{T} \models \varphi]$

Function Five-valued LTL checking ($\mathcal{T}$, φ)
/* initialization of the checking process */
for j = 1 **to** $|FList\ (\varphi)|$ **do {**
$\psi \leftarrow$ FList[j];
RewF $[j] \leftarrow$ Rewrite ($\mathcal{T}_1$, ψ); **}**
for $i = 2$ **to** $|\mathcal{T}|$ **do {**
If φ is a temporal operation free formula
then print "$[\mathcal{T}_i \models \varphi] =$" $[\mathcal{T}_1 \models \varphi]$;
else print "$[\mathcal{T}_i \models \varphi] =$" SubFC ($\varphi$, $\mathcal{T}_i$, RewF[1]);**}**
Function SubFC (φ, $\mathcal{T}_i$, RewF $[j]$)
/* rewriting algorithm for subformulae */
for j = 1 **to** $|FList\ (\varphi)|$ **do {**
$\psi \leftarrow$ FList [j];
case ψ is a propositional logic formula
RewF $[j] \leftarrow (\mathcal{T}_i \models \psi)$;
Eva $[j] \leftarrow [\mathcal{T}_i \models \psi]$;
case $\psi = \neg\psi_1$
RewF $[j] \leftarrow$ not $(\mathcal{T}_i \models \psi)$;
Eva $[j] \leftarrow \neg\, [\mathcal{T}_i \models \psi]$;
case $\psi = \psi_1\ \mathcal{U}\ \psi_2$
RewF $[j] \leftarrow$ RewF $[j]$ or
(Rewrite ($\mathcal{T}_i$, $\mathbf{G}\psi_1$);
Eva $[j] \leftarrow$ [RewF $[j]$] $\vee$
([Rewrite ($\mathcal{T}_i$, $\mathbf{G}\ \psi_1$)] $\wedge$ *pf*);
case $\psi = \mathbf{X}\ \psi_1$
if $|\mathcal{T}_i| > 1$ then RewF $[j] \leftarrow (\mathcal{T}_i \models \mathbf{X}\ \psi_1)$;
else Eva $[j] \leftarrow$ ***pf***;
RewF $[j] \leftarrow \psi_1$;**}**
return Eva $[|FList\ (\varphi)|]$;

For a set of traces $\mathcal{T}$ and an LTL formula, it can be proved that $[\mathcal{T} \models \neg\, \varphi] = \neg\, [\mathcal{T} \models \varphi]$. Therefore, the algorithm for checking the negation formula is straightforward. However, $[\mathcal{T} \models \varphi_1 \vee \varphi_2]$ is not equivalent to $([\mathcal{T} \models \varphi_1] \vee [\mathcal{T} \models \varphi_2])$. For instance, given $\mathcal{T} = \{abccd, bacdc\}$, it holds that $[\mathcal{T} \models a] = uk$, $[\mathcal{T} \models b] = uk$. The truth value for $([\mathcal{T} \models \varphi_1] \vee [\mathcal{T} \models \varphi_2])$ is $(uk \vee uk) = uk$, whereas $[\mathcal{T} \models (a \vee b)] = tt$. To develop the forward rewriting algorithm for $(\varphi_1 \vee \varphi_2)$, we first present the precise relations between $([\mathcal{T} \models \varphi_1] \vee [\mathcal{T} \models \varphi_2])$ and $[\mathcal{T} \models (\varphi_1 \vee \varphi_2)]$ in the following lemma.

Lemma 4. *Given* $\mathsf{e}_1, \mathsf{e}_2 \in E_5$, $[\mathcal{T} \models \varphi_1] = \mathsf{e}_1$ *and* $[\mathcal{T} \models \varphi_2] = \mathsf{e}_2$*, it holds that:*
($\mathsf{e}_1 \vee \mathsf{e}_2$*) = ff iff* $[\mathcal{T} \models (\varphi_1 \vee \varphi_2)] =$ *ff;*
if ($\mathsf{e}_1 \vee \mathsf{e}_2$*) = tt, then* $[\mathcal{T} \models (\varphi_1 \vee \varphi_2)] =$ *tt;*
if ($\mathsf{e}_1 \vee \mathsf{e}_2$*) = pf then* $[\mathcal{T} \models (\varphi_1 \vee \varphi_2)] =$ *pf;*
if ($\mathsf{e}_1 \vee \mathsf{e}_2$*) = uk then* $[\mathcal{T} \models (\varphi_1 \vee \varphi_2)] =$ *uk, or* $[\mathcal{T} \models \varphi_1 \vee \varphi_2] =$ *tt, or* $[\mathcal{T} \models (\varphi_1 \vee \varphi_2)] =$ *pt;*
if ($\mathsf{e}_1 \vee \mathsf{e}_2$*) = pt then* $[\mathcal{T} \models (\varphi_1 \vee \varphi_2)] =$ *pt,* *or* $[\mathcal{T} \models \varphi_1 \vee \varphi_2] =$ *tt.*

The proof can be achieved easily via identities of first-order logic.

For traces τ and τ', let $\tau_{j..i}$ be the segment consisting of

events from the j^{th} event to the i^{th} event of τ, with $0 < j < i$. A safety/liveness property is defined as follows.

Definition 5. (Safety/liveness property) An LTL formula φ is called a safety property, iff for any trace τ,
$\tau \models \varphi$ if for all $i < |\tau|$, there exists $\tau' \in \Sigma^\omega$ such that $\tau_{[..i]}\tau' \models \varphi$; and

φ is called a liveness property, iff for any trace τ,
for all $i < |\tau|$, there exists $\tau' \in \Sigma^\omega$ such that $\tau_{[..i]}\tau' \models \varphi$.

For any property φ, there exists a safety property φ_S and a liveness property φ_L such that $\varphi = \neg(\neg\varphi_S \vee \neg\varphi_L)$ (decomposition theorem) [10].

The following theorem shows that the satisfaction relation between $\mathcal{T}_1\mathcal{T}_2...\mathcal{T}_n$ and $(\varphi_1 \vee \varphi_2)$ can be obtained by checking the satisfaction relation at each position (i.e., $\mathcal{T}_1 \models (\varphi_1 \vee \varphi_2)$, ..., $\mathcal{T}_n \models (\varphi_1 \vee \varphi_2)$) separately. According to this theorem, a forward rewriting algorithm for the formula $(\varphi_1 \vee \varphi_2)$ can be achieved.

Theorem 6. *Given a finite sequence of trace sets $\mathcal{T} \triangleq \mathcal{T}_1\mathcal{T}_{2...n}$, if $[\mathcal{T}_1 \models (\varphi_1 \vee \varphi_2)] =$* tt *(or* ff*), then $[\mathcal{T} \models (\varphi_1 \vee \varphi_2)] =$* tt *(or* ff*, respectively); else if $[\mathcal{T}_1 \models (\varphi_1 \vee \varphi_2)] \neq$* tt *(or* ff*), then $[\mathcal{T} \models (\varphi_1 \vee \varphi_2)]$ (denoted with $\mathsf{e} \in E_5$) is as follows.*

- *if φ_1 and φ_2 are safety properties, then* $\mathsf{e} = [[\mathcal{T}_1 \models (\varphi_1 \vee \varphi_2)] \wedge [\mathcal{T}_{2...n} \models (\varphi_1 \vee \varphi_2)]]$*;*
- *if φ_1 and φ_2 are liveness properties, then* $\mathsf{e} = [[\mathcal{T}_1 \models (\varphi_1 \vee \varphi_2)] \vee [\mathcal{T}_{2...n} \models (\varphi_1 \vee \varphi_2)]]$*;*
- *if φ_1 is a safety property and φ_2 is a boolean formula (without temporal operations), then* $\mathsf{e} = [[\mathcal{T}_1 \models \varphi_1] \vee ([\mathcal{T}_1 \models (\varphi_1 \vee \varphi_2)] \wedge [\mathcal{T}_{2...n} \models \varphi_1])]$*;*
- *if φ_1 is a liveness property and φ_2 is a boolean operation then*
- $\mathsf{e} = [[\mathcal{T}_1 \models \varphi_1] \vee ([\mathcal{T}_1 \models (\varphi_1 \vee \varphi_2)] \vee [\mathcal{T}_{2...n} \models \varphi_1])]$*;*
- *if φ_1 is a safety property and φ_2 is a liveness property, and*
  - $[[\mathcal{T}_1 \models \varphi_1] \vee [\mathcal{T}_1 \models \varphi_2]] =$ pf*, then* $\mathsf{e} = [\mathcal{T}_{2...n} \models \varphi_2]$*;*
  - $[\mathcal{T}_1 \models \varphi_1] \vee [\mathcal{T}_1 \models \varphi_2] =$ pt*, then* $\mathsf{e} = [[\mathcal{T}_1 \models \varphi_2] \vee [\mathcal{T}_{2...n} \models (\varphi_1 \vee \varphi_2)]]$*;*
  - $[[\mathcal{T}_1 \models \varphi_1] \vee [\mathcal{T}_1 \models \varphi_2]] =$ uk*, then* $\mathsf{e} = [[\mathcal{T}_1 \models \varphi_2] \vee [\mathcal{T}_{2...n} \models \varphi_2]]$*.*

Proof: We illustrate the proof of the case "φ_1 is a safety property and φ_2 is a liveness property", others can be proved in the same way.

Denote $[\mathcal{T}_1 \models \varphi_1]$, $[\mathcal{T}_1 \models \varphi_2]$ and $[\mathcal{T} \models \varphi_1 \vee \varphi_2]$ with e_1, e_2 and e, respectively. According to the lemma 4, if $\mathsf{e} \neq$ *tt* and $\mathsf{e} \neq$ *ff*, then the only possibilities of $(\mathsf{e}_1 \vee \mathsf{e}_2)$ are $(\mathsf{e}_1 \vee \mathsf{e}_2) =$ *pf*, $(\mathsf{e}_1 \vee \mathsf{e}_2) =$ *pt* and $(\mathsf{e}_1 \vee \mathsf{e}_2) =$ *uk*. According to definition 5, a liveness property φ can be understand as "if a trace satisfies φ, then it will not violates φ with any suffix". Therefore,
$[\mathcal{T} \models \varphi] \neq$ *pt* if φ is a liveness property. Meanwhile, as $\mathcal{T}$ is a finite sequence, $[\mathcal{T} \models \varphi] \neq$ *ff* since there always exists $\mathcal{T}$' such that $[\mathcal{T}\mathcal{T}' \models \varphi] =$ *tt*. Similar, $[\mathcal{T} \models \varphi] \neq$ *pf* and $[\mathcal{T} \models \varphi] \neq$ *tt* if φ is a safety property, which is "if a trace violates φ, then it will not satisfies φ again". Therefore $\mathsf{e} \neq$ *ff*.

If $(\mathsf{e}_1 \vee \mathsf{e}_2) =$ *pf*, then the only possibility is $\mathsf{e}_1 =$ *ff* and $\mathsf{e}_2 =$ *pf*. That is for all $\tau \in \mathcal{T}_1$, it holds that $\tau \not\models \varphi_1$. Therefore, for any $\tau' \in \mathcal{T}_{2...n}$, it holds that $(\tau\tau' \not\models \varphi_1)$. Then for all τ and τ', $(\tau\tau' \models (\varphi_1 \vee \varphi_2))$ iff $\tau\tau' \models \varphi_2$. Since $[\mathcal{T}\mathcal{T}' \models \varphi] \neq$ *tt*, the suffix $\mathcal{T}$' decides the satisfaction relation between $\mathcal{T}\mathcal{T}'$ and φ, that is $\mathsf{e} = [\mathcal{T}_{2...n} \models \varphi_2]$.

If $(\mathsf{e}_1 \vee \mathsf{e}_2) =$ *pt*, then the possibilities are $\mathsf{e}_1 =$ *pt* and $\mathsf{e}_2 =$ *pf* or *uk*. For the case $\mathsf{e}_1 =$ *pt* and $\mathsf{e}_2 =$ *pf*, for all $\tau \in \mathcal{T}_1$, it holds that $\tau \models \varphi_1$ and $\tau \not\models \varphi_2$. Therefore, the value of e is decided by the value of $[\mathcal{T}_{2...n} \models (\varphi_1 \vee \varphi_2)]$. For the case $\mathsf{e}_1 =$ *pt* and $\mathsf{e}_2 =$ *uk*, for all $\tau \in \mathcal{T}_{2...n}$, there always exists some $\tau \in \mathcal{T}_1$ such that $(\tau\tau' \models \varphi_2)$. Therefore, it is not the case that $\mathsf{e} =$ *pf* (if for all $\tau' \in \mathcal{T}_{2...n}$ with $\tau' \not\models (\varphi_1 \vee \varphi_2)$, $[\mathcal{T}\mathcal{T}' \models \varphi] =$ *uk*). Hence, the value of e equals
$[\mathcal{T}_1 \models \varphi_2] \vee [\mathcal{T}_{2...n} \models (\varphi_1 \vee \varphi_2)]$, which is also the case of $\mathsf{e}_2 =$ *pf*.

If $(\mathsf{e}_1 \vee \mathsf{e}_2) =$ *uk*, then it is possible that $\mathsf{e}_1 =$ *uk* and $\mathsf{e}_2 =$ *uk*, or $\mathsf{e}_1 =$ *ff* and $\mathsf{e}_2 =$ *uk*, or $\mathsf{e}_1 =$ *uk* and $\mathsf{e}_2 =$ *pf*. For all three cases, for all $\tau \in \mathcal{T}_{2...n}$, there always exists some $\tau \in \mathcal{T}_1$ such that $\tau\tau' \not\models \varphi_1$. Therefore,
$\tau\tau' \models \varphi_1 \vee \varphi_2$ iff $\tau\tau' \models \varphi_2$. For the case of $\mathsf{e}_2 =$ *uk*, e equals $[\mathcal{T}_1 \models \varphi_2] \vee [\mathcal{T}_{2...n} \models \varphi_1 \vee \varphi_2]$. ■

With this algorithm, the checking time for $(\mathcal{T}_1\mathcal{T}_2 \ ... \ \mathcal{T}_n \models \varphi)$ is reduced to $(|\mathcal{T}_1| \cdot \mathcal{O}(\tau_1, \varphi) + |\mathcal{T}_2| \cdot \mathcal{O}(\tau_2, \varphi) + \cdots + |\mathcal{T}_n| \cdot \mathcal{O}(\tau_n, \varphi))$, where $\mathcal{O}(\tau_i, \varphi)$ is the checking time for $(\tau_i \models \varphi)$ with $\tau_i \in \mathcal{T}_i$ for all $1 \leq i \leq n$.

IV. Case Study: the RBC/RBC Handover Process for Two Trains

In European Train Control System (ETCS) level 2, the RBC is responsible for providing movement authorities to allow the safe movement of trains. If a train requests to enter a new RBC area, the RBC of the leaving area (i.e., the handing over RBC, denoted with RBC_{HOV}) sends a request message (denoted with *Req*) to the RBC of the entering area (i.e., the accepting RBC, denoted with RBC_{ACC}). If the entering area is not occupied by another train (the route state is "clear"), then RBC_{ACC} permits the request by sending a route related information (denoted with *RRI*), and set the route state to "occupied". After the train has been running a safety distance, the accepting RBC set the route state to "clear" again.

Details of the RBC/RBC handover process can be found in **[11]**. In this section, we illustrate our monitoring approach with a case that two trains from different routes try to enter the same RBC area (Fig. 3). If the two trains request to enter the accepting RBC area at almost the same time, a race condition arises.

Let e be an event, we write $e(i)$ to indicate that e is sent or received by the i^{th} handing over RBC (i.e. $\text{RBC}_{HOV}i$). For example, the event $RRI(1)$ means the route related information is sent to $\text{RBC}_{HOV}(1)$, *Req*(2) means the request is sent by $\text{RBC}_{HOV}(2)$. We assume that trains do not have a global clock, and the maximal time delay of an event is $\Delta t = 5$. We denote the route state "clear" with an event C, and the route state "occupied" with $\neg C$.

The monitor starts monitoring the system whenever a train request to enter the accepting RBC area, and the area is

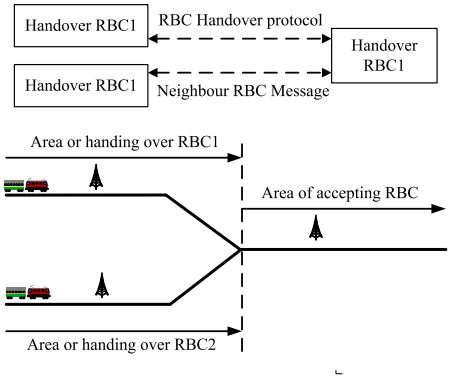

Figure 3. A case study: two trains from different routes try to enter a same RBC area

clear. The following behaviours are collected by the buffer, according the clock of each RBC$_{HOV}$.

$\mathfrak{b}_1$ = {({*Req*(1), C}, 0, 5), ({*Req*(2), C}, 1, 5), ({*Req*(1), C}, 2, 5) ({*Req*(2), C}, 3, 5)};

$\mathfrak{b}_2$ = {(*RRI*(2), 10, 5), *RRI*(2), 12, 5), *RRI*(2), 13, 5), *RRI*(2), 14, 5), (*Req*(1), 16, 5), (*Req*(1), 17, 5) };

$\mathfrak{b}_3$ = {(C, 40, 5)}.

A sequence of behaviours $\mathfrak{b} = \mathfrak{b}_1\mathfrak{b}_2\mathfrak{b}_3$ is sent to the execution recognizer, which converts it to a sequence of trace sets $\mathcal{T} = \mathcal{T}_1\mathcal{T}_2\mathcal{T}_3$. We consider the following properties.

- Property 1: an RBC$_{HOV}$ sends a request to the RBC$_{ACC}$, and if the route is clear, then the RBC$_{ACC}$ sends *RRI* to the RBC$_{HOV}$, and sets the route occupied, i.e., $\varphi_1 = (\mathit{Req}(i) \wedge C) \wedge \mathbf{F}\ (\mathit{RRI}(i) \wedge \neg C)$; and
- property 2: if RBC$_{ACC}$ sends an RRI to a RBC$_{HOV}$, it can not send it to another RBC$_{HOV}$ until the route is clear, i.e., $\varphi_2 = \mathbf{G}\ (\mathit{RRI}(i) \rightarrow (\neg \mathit{RRI}(i')\ \mathcal{U}\ C))$, with $i \neq i'$.

We also inject some errors into the executions, and get behaviours as follows.

$\mathfrak{b}'_1$ = {({*Req*(1), C}, 0, 5), ({*Req*(2), C}, 1, 5)};

$\mathfrak{b}'_2$ = {(*RRI*(1), 8, 5), (*Req*(2), 10, 5)};

$\mathfrak{b}'_3$ = {(*RRI*(2), 17, 5)}.

The trace set sequence for this run is denoted with $\mathcal{T}' = \mathcal{T}'_1\mathcal{T}'_2\mathcal{T}'_3$. The online monitoring results of $\mathcal{T}$ and $\mathcal{T}'$ with respect to φ_1 and φ_2 are as follows.

	$\mathcal{T}$			$\mathcal{T}'$		
	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}'_1$	$\mathcal{T}'_2$	$\mathcal{T}'_3$
φ_1	*pf*	*uk*	*uk*	*pf*	*uk*	*tt*
φ_2	*pt*	*pf*	*pt*	*pt*	*pf*	*ff*

For trace $\mathcal{T}$ and φ_1, the monitoring result *uk* indicates the race condition. Since the run of the system does not violate the correctness properties, this race condition is a non-critical race. The monitoring result *ff* for $\mathcal{T}'$ indicates the run of the system violates φ_2 (possibly caused by the race condition). The system should stop running although it satisfies φ_1. We developed a basic algorithm for monitoring such traces directly from the definition of five-value LTL. Then we implemented the forward algorithm in Maude [12]. This is a high performance system providing a rewriting environment, and is able to execute 2 millions of rewrites per second. With the forward algorithm, the rewriting complexity is reduced significantly. Experiment results show that the approach is feasible. We created a long trace set sequence by repeating 100 times $\mathcal{T}$, and checking it against φ_2. The basic algorithm uses 2.7 million rewrites, whereas the forward algorithm uses only 0.28 million rewrites. Furthermore, since the forward algorithm does not store the entire sequence of trace sets, it is also feasible for online monitoring.

V. Conclusions and Future Work

This paper presented a five valued LTL based runtime verification approach for distributed systems. On one hand, this approach reduces the difficulty of building correctness properties for non-determined temporal relations of causally unrelated executions. These uncertainties are indicated in monitoring results by the truth value *uk*. On the other hand, this approach avoids the problem that a violation of a requirement can only be detected after the monitored system is stopped. Intermediate results are expressed by truth values *pt* and *pf*. In addition, a forward rewriting algorithm was developed. It is able to reduce LTL checking time, and does not suffer from the trace storing problem. Furthermore, the algorithm was used to test a case study in the railway domain, and evaluated on several benchmarks. The results are very encouraging and show the feasibility of our approach.

Future work will show how to reduce uncertainties in monitoring results. Furthermore, we are planning to consider different specification languages for system properties, such as UML diagrams.

References

[1] M. Leucker and C. Schallhart, "A brief account of runtime verification," *Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[2] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for ltl and tltl," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, p. 14, 2011.

[3] ——, "Comparing ltl semantics for runtime verification," *Journal of Logic and Computation*, vol. 20, no. 3, pp. 651–674, 2010.

[4] A. Morgenstern, M. Gesell, and K. Schneider, "An asymptotically correct finite path semantics for ltl," in *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 2012, pp. 304–319.

[5] S. Dutta, "An event based fuzzy temporal logic," in *Multiple-Valued Logic, 1988., Proceedings of the Eighteenth International Symposium on*. IEEE, 1988, pp. 64–71.

[6] K. B. Lamine and F. Kabanza, "Using fuzzy temporal logic for monitoring behavior-based mobile robots," in *Proc. of IASTED Int. Conf. on Robotics and Applications*, 2000, pp. 116–121.

[7] L. Pasquale and P. Spoletini, "Monitoring fuzzy temporal requirements for service compositions: Motivations, challenges and experimental results," in *Requirements Engineering for Systems, Services and Systems-of-Systems (RESS), 2011 Workshop on*. IEEE, 2011, pp. 63–69.

[8] T. Kahsai, M. Roggenbach, and B.-H. Schlingloff, "Specification-based testing for refinement," in *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*. IEEE, 2007, pp. 237–246.

[9] M. Fitting, "Kleene's logic, generalized," *Journal of Logic and Computation*, vol. 1, no. 6, pp. 797–810, 1991.

[10] A. J. Robinson and A. Voronkov, *Handbook of automated reasoning*. Elsevier, 2001, vol. 2.

[11] M. Chai and B.-H. Schlingloff, "A rewriting based monitoring algorithm for tptl," 2013, pp. 61–72. [Online]. Available: http://ceur-ws.org/Vol-1032/paper-06.pdf

[12] P. C. Olveczky, "Real-time maude 2.3 manual," *Research report*, 2004.